Compaq NonStop™ Pathway/iTS SCREEN COBOL Reference Manual

Abstract

This manual describes the SCREEN COBOL programming language, which Compaq *NonStop*™ Pathway/iTS application programmers use to write programs that communicate with operator terminals or intelligent devices and send data to users.

Product Version

Pathway/iTS 1.0

Part Number	Published
426750-001	October 2000

Document History

Part Number	Product Version	Published
127341	Pathway/TS D42	August 1996
136664	Pathway/TS D42+	October 1997
139453	Pathway/TS D42+	January 1998
426750-001	Pathway/iTS 1.0	October 2000

Ordering Information

For manual ordering information: domestic U.S. customers, call 1-800-243-6886; international customers, contact your local sales representative.

Document Disclaimer

Information contained in a manual is subject to change without notice. Please check with your authorized representative to make sure you have the most recent information.

Export Statement

Export of the information contained in this manual may require authorization from the U.S. Department of Commerce.

Examples

Examples and sample programs are for illustration only and may not be suited for your particular purpose. The inclusion of examples and sample programs in the documentation does not warrant, guarantee, or make any representations regarding the use or the results of the use of any examples or sample programs in any documentation. You should verify the applicability of any example or sample program before placing the software into productive use.

U.S. Government Customers

FOR U.S. GOVERNMENT CUSTOMERS REGARDING THIS DOCUMENTATION AND THE ASSOCIATED SOFTWARE:

These notices shall be marked on any reproduction of this data, in whole or in part.

NOTICE: Notwithstanding any other lease or license that may pertain to, or accompany the delivery of, this computer software, the rights of the Government regarding its use, reproduction and disclosure are as set forth in Section 52.227-19 of the FARS Computer Software—Restricted Rights clause.

RESTRICTED RIGHTS NOTICE: Use, duplication, or disclosure by the Government is subject to the restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 52.227-7013.

RESTRICTED RIGHTS LEGEND: Use, duplication or disclosure by the Government is subject to restrictions as set forth in paragraph (b)(3)(B) of the rights in Technical Data and Computer Software clause in DAR 7-104.9(a). This computer software is submitted with "restricted rights." Use, duplication or disclosure is subject to the restrictions as set forth in NASA FAR SUP 18-52 227-79 (April 1985) "Commercial Computer Software—Restricted Rights (April 1985)." If the contract contains the Clause at 18-52 227-74 "Rights in Data General" then the "Alternate III" clause applies.

U.S. Government Users Restricted Rights — Use, duplication or disclosure restricted by GSA ADP Schedule Contract.

Compaq NonStop™ Pathway/iTS SCREEN COBOL Reference Manual

Index | Examples | Figures | Tables

2. SCREEN COBOL Source Program (continued)

5. Data Division (continued)

6. Procedure Division

7. Compilation

8. Pathway Application Example

Examples

Figures

Tables

Tables (continued)

What's New in This Manual

Manual Information

Abstract

This manual describes the SCREEN COBOL programming language, which Compaq *NonStop*™ Pathway/iTS application programmers use to write programs that communicate with operator terminals or intelligent devices and send data to users.

Product Version

Pathway/iTS 1.0

Part Number	Published
426750-001	October 2000

Document History

Part Number	Product Version	Published
127341	Pathway/TS D42	August 1996
136664	Pathway/TS D42+	October 1997
139453	Pathway/TS D42+	January 1998
426750-001	Pathway/iTS 1.0	October 2000

New and Changed Information

The Compaq *NonStop*™ Pathway/iTS product was formerly called Pathway/TS. For the Pathway/iTS 1.0 independent product release, the product was renamed to conform to current Compaq product naming standards and to reflect the new internet (web client) capabilities of the product. After the first reference to the product name in each section of this manual, subsequent references use the shortened form of the name, Pathway/iTS.

Product Changes

This manual edition adds information about the CONVERT command, which converts a SCREEN COBOL object program to a web client consisting of Java code and HTML pages. This new information includes:

This manual edition reflects the following changes to Pathway/iTS:

- The new capability of Pathway/iTS to convert SCREEN COBOL object files to web clients through the SCUP CONVERT command, as described in Managing Object Files With SCUP on page 1-10.

- Filtering of 3161 timeout error messages when the ON ERROR clause is used with a SEND MESSAGE statement, as described under TIMEOUT timeout-value on page 6-82 in the description of the SEND MESSAGE statement.

Corrections and Enhancements to the Manual

The following corrections and enhancements have been made to the material in this manual:

- The discussion of Modifying or Replacing the Advisory Message Routine on page A-4 has been corrected to reflect the use of pTAL and the `nld` utility, and also enhanced to clarify how to change message text in or add messages to the standard ADVISORY^MESSAGE routine.
- Omissions were corrected in the boxed syntax for Field-Characteristic Clauses on page 5-33.
- References to Compaq trademarks and the Pathway/iTS product name have been updated.
- References to obsolete products have been removed.
- Miscellaneous terminology changes and editorial corrections have been made.

About This Manual

This manual describes the SCREEN COBOL programming language. This language is used for writing programs that define and control terminal displays or intelligent devices for online transaction processing applications running in a PATHMON environment.

Who Should Read This Manual

This manual is for programmers who are responsible for developing SCREEN COBOL programs to define and control terminal displays or intelligent devices in a PATHMON environment. Readers are assumed to have experience in writing, compiling, and running programs on Compaq *NonStop™ Himalaya* systems.

Related Documentation

In addition to this manual, information about Pathway/iTS appears in the following publications:

Compaq NonStop™ Pathway/iTS SCUP Reference Manual	Describes managing a SCREEN COBOL library with the SCREEN COBOL Utility Program (SCUP).
Compaq NonStop™ Pathway/iTS Web Client Programming Manual	Describes how to convert SCREEN COBOL requesters to web clients, explains how to build and deploy those clients, and also provides the information Java developers and web designers need to to modify and enhance the Java and HTML portions of the converted clients.
Compaq NonStop™ Pathway/iTS TCP and Terminal Programming Guide	A guide for programmers who are writing SCREEN COBOL requesters to be used in Pathway applications.
Compaq NonStop™ Pathway/iTS System Management Manual	Describes the interactive management interface to the Pathway/iTS product and describes how to manage Pathway/iTS objects.
Compaq NonStop™ Pathway/iTS Management Programming Manual	Describes the management programming interface for Pathway/iTS objects in the PATHMON environment.
Compaq NonStop™ Pathway Products Glossary	Defines technical terms used in this manual and in other manuals for the Pathway products: Pathway/iTS, NonStop™ TS/MP, and Pathway/XM.
Operator Messages Manual	Describes all messages that are distributed by the Event Management Service (EMS), including those generated by NonStop™ TS/MP and Pathway/iTS processes.

Your Comments Invited

After using this manual, please take a moment to send us your comments. You can do this by returning a Reader Comment Card or by sending an Internet mail message.

A Reader Comment Card is located at the back of printed manuals and as a separate file on the Compaq CD Read disc. You can either FAX or mail the card to us. The FAX number and mailing address are provided on the card.

Also provided on the Reader Comment Card is an Internet mail address. When you send an Internet mail message to us, we immediately acknowledge receipt of your message. A detailed response to your message is sent as soon as possible. Be sure to include your name, company name, address, and phone number in your message. If your comments are specific to a particular manual, also include the part number and title of the manual.

Many of the improvements you see in Compaq manuals are a result of suggestions from our customers. Please take this opportunity to help us improve future manuals.

Notation Conventions

General Syntax Notation

The following list summarizes the notation conventions for syntax presentation in this manual.

UPPERCASE LETTERS. Uppercase letters indicate keywords and reserved words; enter these items exactly as shown. Items not enclosed in brackets are required. For example:

```
MAXATTACH
```

lowercase italic letters. Lowercase italic letters indicate variable items that you supply. Items not enclosed in brackets are required. For example:

```
file-name
```

[ ] Brackets. Brackets enclose optional syntax items. For example:

```
TERM [\system-name.]$terminal-name
```

```
INT[ERRUPTS]
```

A group of items enclosed in brackets is a list from which you can choose one item or none. The items in the list may be arranged either vertically, with aligned brackets on each side of the list, or horizontally, enclosed in a pair of brackets and separated by vertical lines. For example:

```
LIGHTS [ ON              ]
       [ OFF             ]
       [ SMOOTH [ num ] ]
```

```
K [ X | D ] address-1
```

{ } Braces. A group of items enclosed in braces is a list from which you are required to choose one item. The items in the list may be arranged either vertically, with aligned braces on each side of the list, or horizontally, enclosed in a pair of braces and separated by vertical lines. For example:

```
LISTOPENS PROCESS { $appl-mgr-name }
                  { $process-name  }
```

```
ALLOWSU { ON | OFF }
```

| Vertical Line. A vertical line separates alternatives in a horizontal list that is enclosed in brackets or braces. For example:

```
INSPECT { OFF | ON | SAVEABEND }
```

… Ellipsis. An ellipsis immediately following a pair of brackets or braces indicates that you can repeat the enclosed sequence of syntax items any number of times. For example:

```
M address-1 [ , new-value ]...
```

```
[ - ] {0|1|2|3|4|5|6|7|8|9}...
```

An ellipsis immediately following a single syntax item indicates that you can repeat that syntax item any number of times. For example:

```
"s-char..."
```

Punctuation. Parentheses, commas, semicolons, and other symbols not previously described must be entered as shown. For example:

```
error := NEXTFILENAME ( file-name ) ;

LISTOPENS SU $process-name.#su-name
```

Quotation marks around a symbol such as a bracket or brace indicate the symbol is a required character that you must enter as shown. For example:

```
"[" repetition-constant-list "]"
```

Item Spacing. Spaces shown between items are required unless one of the items is a punctuation symbol such as a parenthesis or a comma. For example:

```
CALL STEPMOM ( process-id ) ;
```

If there is no space between two items, spaces are not permitted. In the following example, there are no spaces permitted between the period and any other items:

```
$process-name.#su-name
```

Line Spacing. If the syntax of a command is too long to fit on a single line, each continuation line is indented three spaces and is separated from the preceding line by a blank line. This spacing distinguishes items in a continuation line from items in a vertical list of selections. For example:

```
ALTER [ / OUT file-spec / ] CONTROLLER

   [ , attribute-spec ]...
```

Notation for Messages

The following list summarizes the notation conventions for the presentation of displayed messages in this manual.

Nonitalic text. Nonitalic letters, numbers, and punctuation indicate text that is displayed or returned exactly as shown. For example:

```
Backup Up.
```

lowercase italic letters. Lowercase italic letters indicate variable items whose values are displayed or returned. For example:

```
p-register

process-name
```

1 Introduction to SCREEN COBOL

SCREEN COBOL is a component of the Compaq *NonStop*™ Pathway/iTS transaction processing software. Together, Pathway/iTS and the underlying NonStop™ Transaction Services/MP (TS/MP) product supply the programs and operating environment required for developing online transaction processing applications.

SCREEN COBOL is a high-level programming language used for coding programs called requesters, which control display terminals, communicate with external processes (that is, processes outside of Pathway/iTS) that control intelligent devices, or communicate with other devices or processes in a Pathway environment. Operators at Pathway terminals or devices are usually entering online transactions.

To help migrate terminal requesters to the Internet, SCREEN COBOL programs can be converted into web clients consisting of a combination of Java code and HTML pages. This conversion feature allows you to maintain the same SCREEN COBOL source code for both terminals and web clients. For further information about converting SCREEN COBOL requesters to web clients, refer to the *Compaq NonStop™ Pathway/iTS Web Client Programming Manual*.

A warehouse inventory control system represents a typical online transaction processing application. For example, operators at terminals retrieve inventory information from the database to determine the quantities on hand of specific items. As new items are received and existing items are shipped, the operators perform update operations on the database to reflect current inventory.

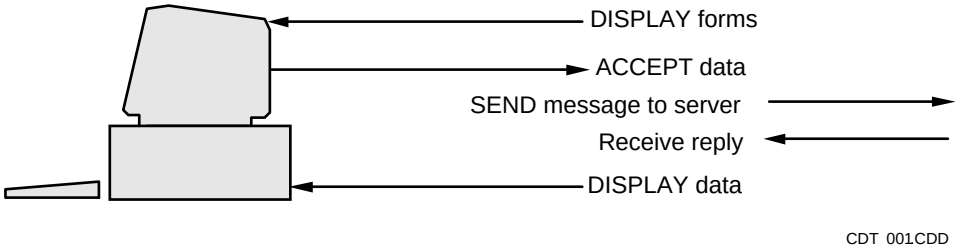
SCREEN COBOL programs handle such processing operations by performing the following types of functions:

- Displaying forms on terminal screens
- Accepting data that has been entered into those forms by operators
- Sending messages to, and receiving replies from, external processes that control intelligent devices
- Sending messages to, and receiving replies from, Pathway server processes that update the database

SCREEN COBOL programs themselves do not access databases. When an operator takes an action requiring database access, the SCREEN COBOL requester program controlling the terminal sends a message to another program, called a server, which performs the necessary database operations. After the server has completed the requested tasks, it sends a reply message back to the requester indicating whether it did so successfully.

Figure 1-1 illustrates the operations performed by SCREEN COBOL programs.

Figure 1-1. Operations Performed by SCREEN COBOL Programs



Pathway Environment Overview

To give you an overview of the Pathway transaction processing environment, the various components of this environment are described below, including how each component affects SCREEN COBOL programs.

Pathway System Components

The primary components of the Pathway environment are:

- The Pathway monitor (PATHMON) process—the central controlling process for operations in the Pathway environment
- PATHCOM—the command interface to the PATHMON process
- SCREEN COBOL—the high-level language used to code requester programs that act as intermediaries between display terminals (or external processes that control intelligent devices) and Pathway servers
- Terminal control process (TCP)—the process that interprets SCREEN COBOL object code and controls the terminals running transaction processing applications
- Requesters—programs that usually provide presentation services for terminal devices and communicate with server processes. Requesters may be tasks executing SCREEN COBOL code within a (multithreaded) terminal control process, or they may be Pathsend processes written in TAL, C, COBOL85, or Pascal.
- Servers—processes that perform database operations in response to messages received from requesters and reply to those messages
- Compaq NonStop™ Transaction Management Facility (TMF)—the data management product that is available for use with Pathway/iTS to maintain the consistency of the database and provide the tools for database recovery
- Compaq *Inspect*—the interactive, symbolic program debugging tool that you can use to examine and modify SCREEN COBOL programs

Pathway Monitor (PATHMON) Process

The PATHMON process is the central control process for a Pathway environment. The PATHMON process maintains information about the objects (terminals, TCPs, servers) in a PATHMON environment and controls the objects.

The PATHMON process enforces the limits for the system defined in the system configuration and monitors the operation of system objects by:

- Keeping a record of the object definitions in a control file
- Reporting status information in a log file for the TCPs, servers, terminals, and SCREEN COBOL programs
- Granting communication links between TCP and servers
- Reporting system errors within the Pathway environment
- Shutting down the system by stopping PATHMON-controlled objects

PATHCOM

PATHCOM is the command interface that is used to define and manage other objects controlled by the PATHMON process. PATHCOM supports several sets of object-related commands. These commands describe which terminals are controlled by each TCP, describe the capacity of the PATHMON environment by indicating the maximum number of objects that can exist, start and stop objects, and display status and statistical information.

SCREEN COBOL

SCREEN COBOL is the programming language used to define terminal displays and process data entered at terminals. The language is similar to COBOL. SCREEN COBOL and COBOL have the same program organization, coding conventions, and language elements. Both languages have the same major divisions:

- Identification Division
- Environment Division
- Data Division
- Procedure Division

Data descriptions in the SCREEN COBOL Working-Storage and Linkage Sections are the same format as those in COBOL.

Unlike COBOL, SCREEN COBOL provides features for screen handling and for exchanging messages with intelligent devices. Note that the term *intelligent device* as used within the context of Pathway/iTS covers a broad spectrum of entities ranging from personal computers, automated teller machines, and point-of-sale devices to *Guardian* operating environment processes and communication lines. In SCREEN COBOL, a Screen Section or a Message Section replaces the COBOL File Section. The Screen Section defines the data fields and other characteristics of a terminal screen; the

Message Section defines the messages sent to, and replies received from, an intelligent device (or the process that controls it). SCREEN COBOL verbs allow you to display and accept terminal screen data and to communicate with intelligent devices.

SCREEN COBOL source code is not compiled into machine-language instructions. The source code is compiled into object code called pseudocode. The pseudocode for SCREEN COBOL programs is stored in a library and interpreted by the terminal control process.

Terminal Control Process (TCP)

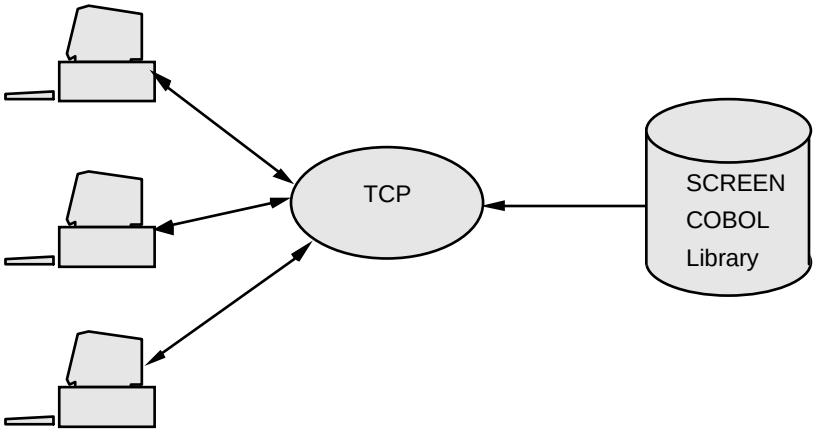
The TCP interprets and executes SCREEN COBOL programs. The terminal control provided by the TCP simplifies your programming task. The TCP is a multithreaded process that can execute several different SCREEN COBOL programs simultaneously. The TCP performs the following operations for each SCREEN COBOL program executed:

- Controls simultaneous execution of the SCREEN COBOL program at several terminals
- Maintains separate data areas and control information for each terminal under its control
- Performs the physical I/O operations to transmit data to and from terminals
- Provides a device-independent method of communicating with intelligent devices
- Performs field validation based on edit symbols in the SCREEN COBOL program
- Converts data between external and internal representations
- Sends messages to server processes
- Returns messages from server processes to appropriate terminals

Instructions to the TCP for checking and converting data are in an object file supplied as part of the Pathway/iTS product. You can replace these system routines with your own routines called user-conversion routines.

Figure 1-2 illustrates the TCP executing SCREEN COBOL code on behalf of multiple terminals. The TCP handles multiple terminals with a single copy of a SCREEN COBOL program and performs all I/O for the screen displays.

Figure 1-2. Multiple Terminal Control Through the TCP



In a Pathway environment, the TCP executing SCREEN COBOL code at a terminal constitutes a requester process. However, in this manual, discussions of requester-server functions refer to a SCREEN COBOL program unit as a requester program.

Requesters

Requesters usually provide presentation services for terminal devices and communicate with server processes. Requesters may be tasks executing SCREEN COBOL code within a (multithreaded) terminal control process, or they may be Pathsend processes written in Transaction Application Language (TAL), C, COBOL85, or Pascal.

Pathway/iTS requesters are written as SCREEN COBOL programs. SCREEN COBOL programs control screen displays, manage terminals, manage messages for intelligent devices, and perform calls to other SCREEN COBOL programs. SCREEN COBOL programs also send data to and receive replies from Pathway server processes. A request to a server, generated by the execution of a SCREEN COBOL SEND statement, is managed by a terminal control process (TCP).

Requester programs may also be written as Guardian processes in the Transaction Application Language (TAL), C, COBOL85, or Pascal. Such requester programs include calls to the Pathsend procedures provided as part of the NonStop™ TS/MP product. For information about writing Pathsend requesters, refer to the *NonStop™ TS/MP Pathsend and Server Programming Manual*.

Servers

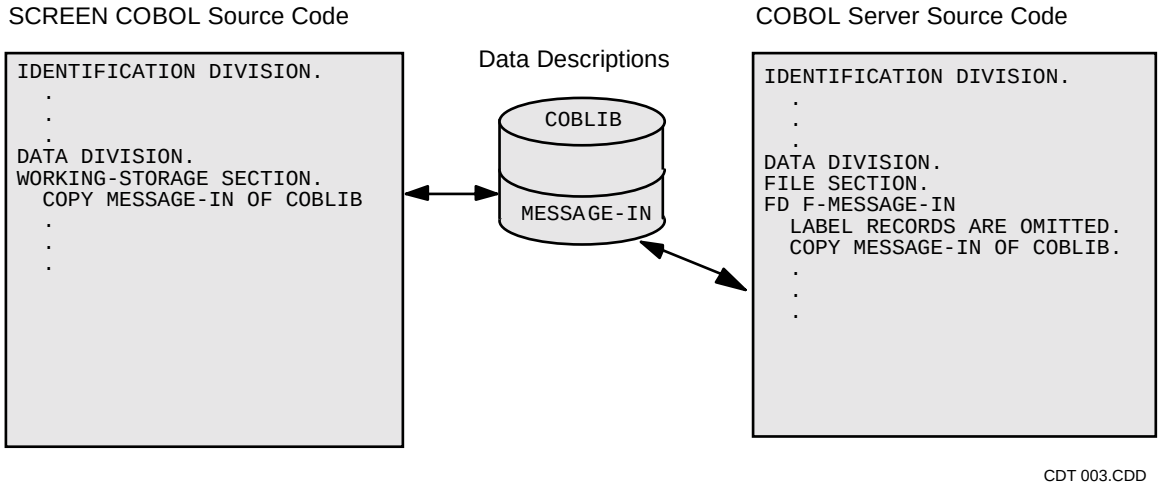
Servers are programs written in C, C++, COBOL85, pTAL, TAL, FORTRAN, or Pascal in the Guardian environment to respond to requests to perform database operations. The requests are made in the form of interprocess messages. When a SCREEN COBOL requester program is used, these messages are generated according to the statements in the SCREEN COBOL program and sent by a TCP. Servers receive the requests, perform database I/O functions, and return appropriate replies to the TCP.

A server is configured to be a member of a particular server class. The server class itself has specific characteristics that are defined within the PATHMON configuration. Individual servers within a server class use copies of the same server program code and separate data areas; the PATHMON process creates new servers from a single server program according to the configuration instructions.

A SCREEN COBOL program and its associated servers require corresponding message descriptions. To ensure message description correspondence, both programs can use common code for the description obtained from a source code library.

Figure 1-3 illustrates the message description correspondence between a SCREEN COBOL requester and a COBOL server.

Figure 1-3. Message Description Correspondence



Transaction Management Facility (TMF)

TMF maintains the consistency of a database and provides the tools for database recovery. TMF requires that monitored data files be flagged for auditing. TMF audits a file by maintaining before and after images of changes to the files. These images provide the basis for transaction backout, which cancels the effects of a partially completed transaction, and database rollforward, which restores a database to a consistent state after a catastrophic failure.

SCREEN COBOL programs communicate with TMF by executing particular statements. The BEGIN-TRANSACTION statement marks the beginning of a transaction, when the terminal enters transaction mode. The terminal remains in transaction mode until execution of the SCREEN COBOL END-TRANSACTION (or ABORT-TRANSACTION) statement. These statements begin and end a sequence of operations that are treated as a single transaction by TMF.

An executing SCREEN COBOL program is called a terminal-program unit in Pathway/iTS. To communicate with TMF, the terminal-program unit must be configured for TMF by issuing commands to PATHCOM.

Inspect

Inspect is an interactive symbolic program-debugging tool that you can use to examine and modify SCREEN COBOL programs. Inspect runs as a separate process that communicates through the TCP with the SCREEN COBOL program running on a Pathway/iTS terminal. By issuing commands to Inspect, you can control and modify an executing program.

Before you can use Inspect, the PATHMON environment must be configured for communication with Inspect. In addition, a symbol-table file generated for the program by the SCREEN COBOL compiler must be available to the TCP.

Communication Between Processes

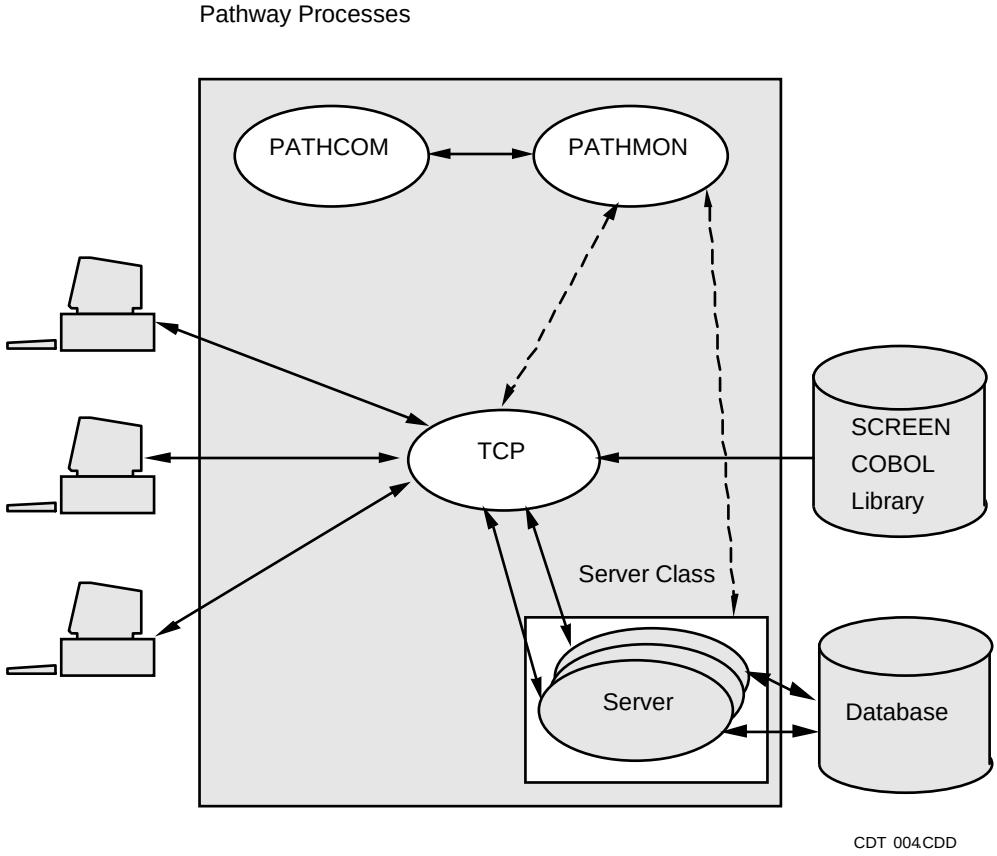
SCREEN COBOL programs running at terminals communicate with servers by exchanging interprocess messages through the TCP. The TCP executes a SCREEN COBOL SEND statement, which builds a message and specifies a server class. The TCP obtains a link to a server class from the PATHMON process and actually sends the message. A server replies with an interprocess message that reaches the appropriate terminal, again through the TCP.

When a SCREEN COBOL program provides instructions to communicate with a server in a different PATHMON environment, the program becomes location sensitive. In this situation, the SCREEN COBOL SEND statement indicates an external server by specifying the Guardian node name and PATHMON name for the PATHMON process controlling the external server. An external server is one that runs under a PATHMON environment different from that of the requesting SCREEN COBOL program.

Specifying the node name and PATHMON name makes communication possible among multiple PATHMON environments on the same Guardian node or on different Guardian nodes.

Figure 1-4 illustrates the communication between PATHMON environment processes active in requester-server message exchange. The SCREEN COBOL library contains the object code executed by the TCP.

Figure 1-4. Communication Between Processes in a PATHMON Environment



Developing Programs With System Tools

Compaq provides tools to develop source code and to generate and manage the object files. Object code is generated with the SCREEN COBOL compiler. The resulting object files are managed with the SCREEN COBOL Utility Program (SCUP).

Generating Object Files With the Compiler

To produce the SCREEN COBOL object files, you execute the compiler run command SCOBOLX. The compiler generates the object code according to the compiler commands specified in the run command and in the source code.

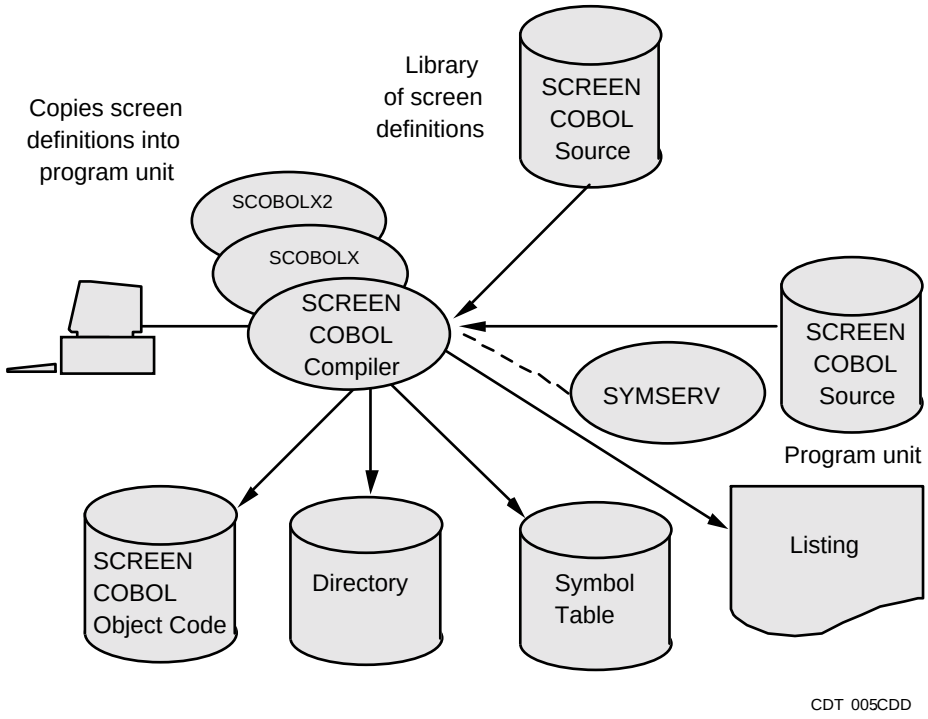
The SCREEN COBOL compiler has three associated processes (SCOBOLX, SCOBOLX2, and SYMSERV). These processes generate the object code, which is written to two or three files depending on selected compile options. The SCOBOLX and SCOBOLX2 processes produce two object files—a directory file and a code file. The optional third process (SYMSERV) produces a symbol table for the program.

Note. The SYMSERV process is not one of the Pathway/iTS object files. Therefore, SYMSERV must reside on the same volume as SCOBOLX and SCOBOLX2 when used to compile a SCREEN COBOL program.

To execute a SCREEN COBOL program, the directory and code files must be available to the TCP. The symbol-table file is required when INSPECT is used for program debugging. In addition to generating object code, the compiler builds a SCREEN COBOL library. Each time the compiler successfully compiles a source program, the compiler adds the new version of the object file to the previously compiled versions. The compiler, however, has no features for managing the object files.

Figure 1-5 illustrates the generating of SCREEN COBOL object files. In this figure, input to the compiler includes the program source code and code copied from a screen library file. The compiler output includes three object files (code, directory, and symbol table) and a listing that can include a compilation listing, a cross-reference listing, and a map.

Figure 1-5. Generating SCREEN COBOL Object Files



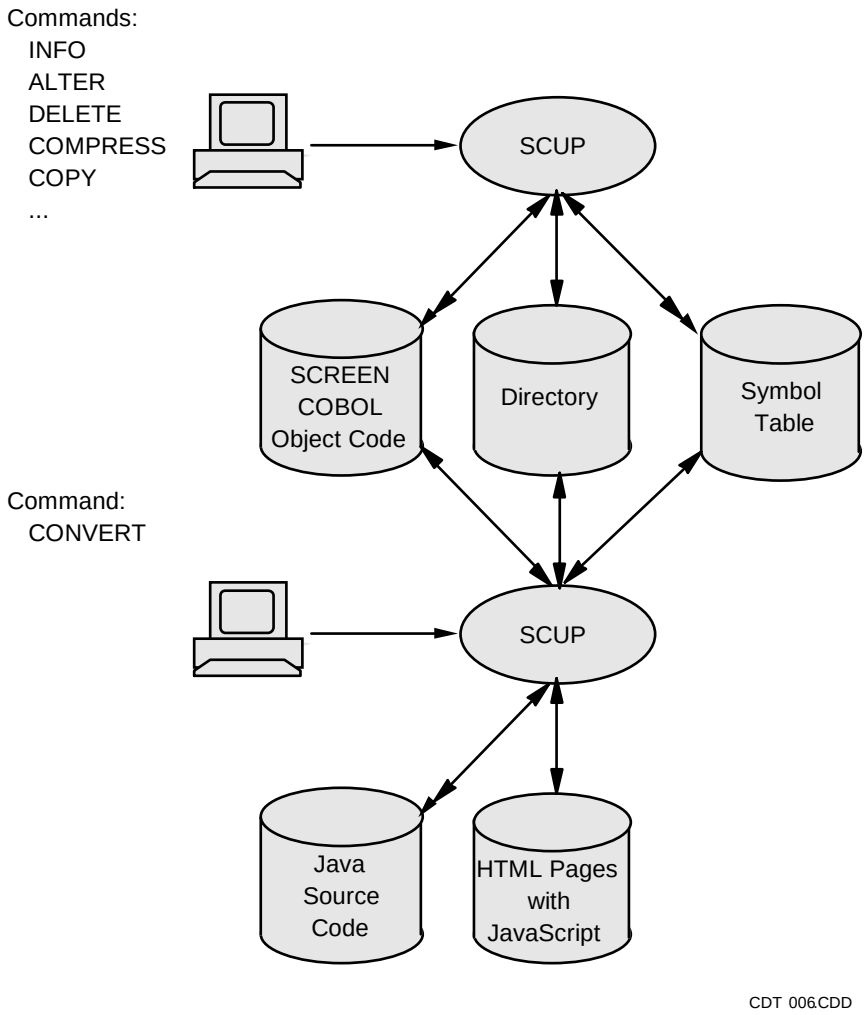
Note. In earlier releases, the version of the TCP under which a SCREEN COBOL program was executed had to be the same or later than that of the SCREEN COBOL compiler under which the program was compiled. For D40 and later releases, the version of the TCP under which a SCREEN COBOL program is executed can also be earlier than that of the SCREEN COBOL compiler unless it uses newer incompatible features, in which case the program will be assigned the version corresponding to the latest feature used. For example, you can now use a C31 TCP with a SCREEN COBOL program compiled under a D40 compiler provided the SCREEN COBOL program does not use any new incompatible features. New data structures, new statements, and new versions of statements are often generated by a given release of the compiler. These enhancements can be processed only by a TCP of the same or a later release.

Managing Object Files With SCUP

The SCUP utility provides functions for managing SCREEN COBOL object files, as shown in Figure 1-6. The functions are:

- Displaying information about the programs in the object files
- Deleting previously compiled program versions from the object files
- Generating a new SCREEN COBOL object file by copying programs from one SCREEN COBOL object file to another
- Reclaiming file space by compressing the object files
- Setting access flags that control access to programs
- Converting a group of programs in a SCREEN COBOL library into a web client consisting of Java classes and HTML pages

Figure 1-6. Managing SCREEN COBOL Object Files With SCUP



For further information about SCUP and how to use it, refer to the *Compaq NonStop™ Pathway/iTS SCUP Reference Manual*.

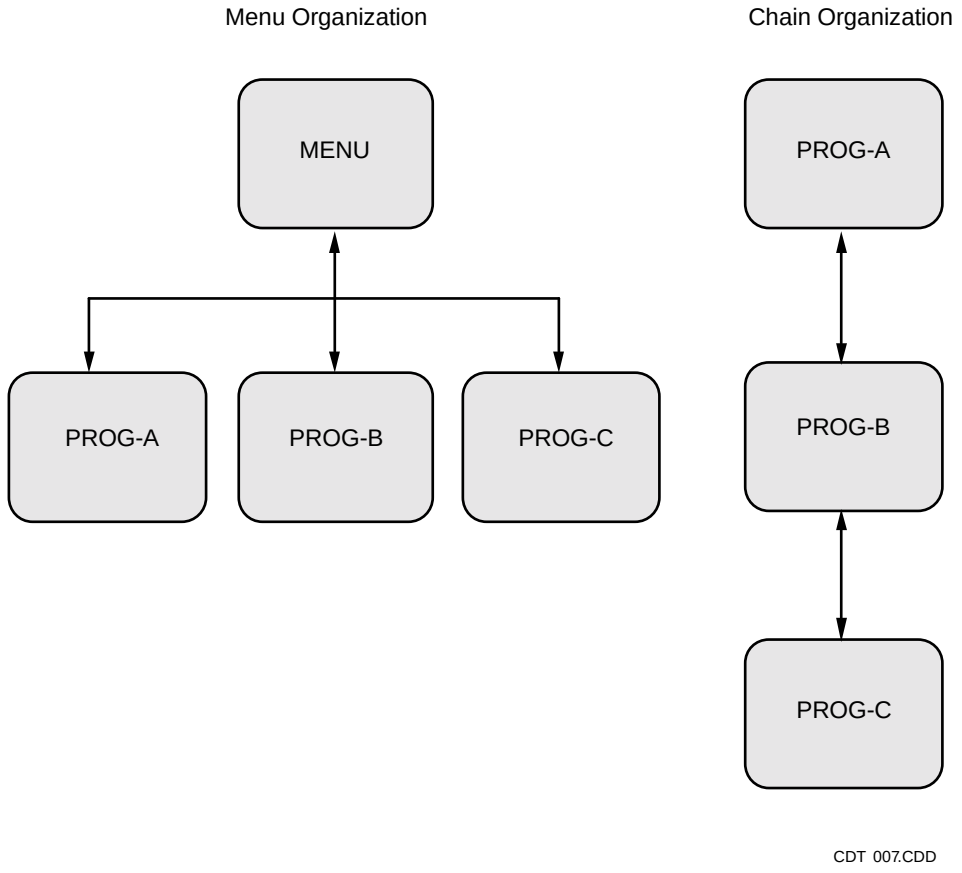
Designing Program Logic

Designing a transaction processing application involves designing SCREEN COBOL requester programs and servers and partitioning the workload between them. Typical design practices are discussed in the following paragraphs.

Organizing SCREEN COBOL Program Groups

SCREEN COBOL programs are usually organized into groups of simple, related programs. Each program performs a discrete function and calls other programs depending on how the Pathway application is designed to process the transactions. Figure 1-7 illustrates two typical organizations: the menu and chain.

Figure 1-7. Program Organizations



Both organizations are hierarchical with entry to the group through a single program. In the menu organization, the main menu screen displays a selection of operations, with any selection resulting in a call to another simple SCREEN COBOL program that handles one operation.

The chain organization consists of a series of programs. In the chain, the terminal operator's selections can call PROG-B from PROG-A and PROG-C from PROG-B.

General Rules for Program Design

The list that follows presents some general guidelines that are helpful when designing standard (screen-oriented) SCREEN COBOL requester programs. For concepts and guidelines pertaining to intelligent device (message-oriented) requester programs, refer to the *Compaq NonStop™ Pathway/iTS TCP and Terminal Programming Guide.*

- Design simple screens.
- Keep the operator informed of task completion, errors, the next step, and what the system is doing at all times.
- Design screens to display initial values and thus reduce keying of data. If no initial value is declared for a screen field, a default value can be established by moving a value into the data name associated with the field; this default value is changed only if the operator enters data into the field or the program moves another value into the field.
- Protect crucial screen fields; for example, protect primary key fields against update.
- Reduce errors on crucial screen fields by using check digits. Check digit processing can be performed by the SCREEN COBOL program or by user conversion procedures as described in the *Compaq NonStop™ Pathway/iTS TCP and Terminal Programming Guide.*
- Keep context information in the requester program and never in the server. Context is any information that is required by a process to resume operating in a previously existing environment.
- Use a modular program design for ease of maintenance.

2

SCREEN COBOL Source Program

SCREEN COBOL programs handle processing operations which control display terminals, communicate with external processes that control intelligent devices, and communicate with other devices or processes in a Pathway environment.

You can use the SCREEN COBOL procedural language to:

- Define the characteristics of terminal display screens
- Control how data is to be displayed on and accepted from a terminal screen
- Send messages to and receive replies from intelligent devices
- Indicate how data is to be converted and how editing checks are to be applied to data
- Specify and send messages to server processes to access or modify data in a database

SCREEN COBOL is available for use with:

- The 6510, 6520, 6530, 6540, and IBM 3270 terminals
- Those devices operating as conversational mode terminals as recognized by the Guardian file system
- Intelligent devices such as personal computers, point-of-sale devices, automated teller machines, communication lines, or Guardian processes, and the 6540 terminal operating as a personal computer

A SCREEN COBOL program is always designed as if to control a single terminal or device. The Terminal Control Process (TCP) that interprets the object code generated by the SCREEN COBOL compiler can, however, perform simultaneous executions of the same code for many terminals or devices under its control.

Program Operating Modes

Generally, a SCREEN COBOL program displays formatted information, receives data entered from a terminal, and performs some action based on the data. SCREEN COBOL enables you to write programs that perform these operations in:

- Block mode (full-screen accept and display operations)
- Conversational mode (line-by-line accept operations)
- Intelligent mode (device-independent operations)

Some of the SCREEN COBOL statements and clauses act differently depending on the mode.

Block Mode Program

To execute in block mode, a SCREEN COBOL program must be communicating with a block mode terminal. The screen definitions for any SCREEN COBOL program are restricted by the characteristics of the specific type of terminal on which your program runs.

A SCREEN COBOL program running in block mode performs as follows:

- Recognizes a specific terminal type
- Displays a full screen of information on the terminal
- Accepts data entered from the terminal one screen at a time
- Recognizes function keys and associates each with a particular function (for example, pressing the F1 function key might be associated with exiting from a screen)

Conversational Mode Program

A SCREEN COBOL program written for conversational mode operation can run on either a block mode terminal or a conversational mode terminal. Once a program is specified as conversational, that program performs according to the restrictions for a conversational terminal regardless of the type of terminal on which the program runs.

A SCREEN COBOL program running in conversational mode performs as follows:

- Displays information on the terminal during a DISPLAY statement, one line at a time
- Accepts data entered from the terminal, one line at a time
- Responds to a set of input-control characters when the terminal is enabled to accept data
- Recognizes only keyboard characters, carriage return, and line feed (not function keys)
- Restricts the display field attributes to `BELL` and `HIDDEN`

Intelligent Mode Program

A SCREEN COBOL program running in intelligent mode does not control how data appears to the intelligent device nor does it perform any other device-dependent functions. It is the responsibility of the intelligent device to read and process messages sent by the SCREEN COBOL program and to reply in a format accessible to the program.

A program that communicates with an intelligent device uses a Message Section instead of a Screen Section. It describes data to be sent to or received from the intelligent device in the Message Section. The program can also send data directly from Working-Storage, bypassing the Message Section.

In intelligent mode, a SCREEN COBOL program can:

- Send data directly from Working-Storage to the intelligent device
- Receive a reply from the intelligent device and move it directly to Working-Storage
- Map data from Working-Storage into a message entry in the Message Section
- Request the intelligent device to return a reply as a message entry in the Message Section

Program Organization

A SCREEN COBOL program is organized into four divisions that must appear in the following order:

1. The Identification Division identifies the program. Comments such as the name of the programmer, the date the program was written, and a description of the program can be declared in this division.
2. The Environment Division specifies the program execution environment. Display error attributes, processing options, computer equipment, and terminal equipment can be described in this division.
3. The Data Division defines the program data structures in terms of their formats and usage. In the Data Division are the following sections:
   - The Working-Storage Section in this division describes data local to the program.
   - The Linkage Section describes data passed from another program.
   - The Screen Section describes data displayed on and accepted from a terminal.
   - The Message Section describes messages sent to and replies returned from an intelligent device.
4. The Procedure Division specifies the processing steps of the program.

Language Elements

The SCREEN COBOL language elements fall into one of two categories:

- Character strings—strings of contiguous characters
- Separators—characters that separate one character string from another

The language elements that comprise the SCREEN COBOL source program are described in the following paragraphs.

SCREEN COBOL Character Set

The SCREEN COBOL character set is a subset of the ASCII character set and consists of 52 characters. Table 2-1 lists these characters.

Table 2-1. SCREEN COBOL Character Set

0-9	Digits	,	Comma
A-Z	Letters	;	Semicolon
	Space (blank)	.	Period (decimal point)
+	Plus sign	"	Quotation mark
–	Minus sign (hyphen)	(	Left parenthesis
*	Asterisk	)	Right parenthesis
/	Stroke (slash)	>	Greater than
=	Equal sign	<	Less than
$	Currency sign	@	Commercial at
	Double-byte character		

In addition, you can use characters from a double-byte character set if you specify the CHARACTER-SET IS KANJI-KATAKANA clause in the OBJECT-COMPUTER paragraph of the Environment Division. See the *Compaq NonStop™ Pathway/iTS TCP and Terminal Programming Guide* for more information about double-byte character sets.

Note. Existing program units that do not use double-byte characters need not specify the CHARACTER-SET IS KANJI-KATAKANA clause (even if 1-byte Katakana characters are used). If a program unit specifies the CHARACTER-SET IS KANJI-KATAKANA clause, 1-byte Katakana or 2-byte Katakana character sets (or both) are supported, depending on the terminal. Your application can use whichever Katakana character set is supported by the terminal; however, Kanji characters are only supported for program units that specify the CHARACTER-SET IS KANJI-KATAKANA clause.

The following definitions apply to the SCREEN COBOL character set:

- Alphabetic characters include letters A through Z and space.
- Numeric characters include digits 0 through 9.
- Special characters include all characters except letters A through Z, space, and digits 0 through 9.
- The full ASCII character set can be used in comments and literals.
- Any valid double-byte character from the Shift-JIS character set (X 0208) is allowed.

- Alphanumeric characters include any character in this set.

Note. Double-byte character sets are supported only on certain devices. See your service provider for a list of devices that support double-byte character sets.

Editing Characters

Editing characters are symbols that can be used in PICTURE clauses to format screen data. Table 2-2 lists the editing characters.

Table 2-2. Editing Characters

A	Alphabetic or space	–	Minus
B	Space insertion	CR	Credit
N	Double-byte character	+	Plus
P	Decimal position (scaled)	DB	Debit
V	Decimal position (fixed)	*	Check protect
X	ASCII character	$	Currency symbol
Z	Zero suppress	,	Comma (decimal point)
0	Zero	.	Period (decimal point)
9	Numeric digit	/	Stroke (right slash)

Punctuation Characters

Punctuation characters are used to separate words, sentences, or special clauses, and to group arithmetic relationships. Table 2-3 lists the punctuation characters.

Table 2-3. Punctuation Characters

,	Comma	(	Left parenthesis
;	Semicolon	)	Right parenthesis
.	Period		Space (blank)
“	Quotation mark	=	Equal sign

Separators

Separators are strings of one or more punctuation characters; they can have leading or trailing blanks. Table 2-4 lists and defines the separators.

Table 2-4. Separators

, ; .	A comma, semicolon, or period immediately followed by a space is a separator. A period can appear as a separator only when it terminates headers, entries, and sentences as defined by the syntax. A comma or semicolon that is a separator is treated as a space.
" "	Quotation marks are used to enclose nonnumeric literals. The characters appear in balanced pairs except when the literal is continued across a line. The first quotation mark must be preceded by a space, and the second one must be followed by a separator other than another quotation mark.
( )	Right and left parentheses enclose certain parts of character strings. Although they must appear in balanced pairs, each is considered a separator.
space	A space separates language elements.

Some character strings include punctuation characters, in which case those characters do not act as separators. Any character in the ASCII character set can appear in a nonnumeric literal, provided the character does not have special meaning to a hardware device.

SCREEN COBOL Words

A SCREEN COBOL word is a character string that forms a reserved word, user-defined word, or system name. A word can have a maximum of 30 characters.

Reserved Words

A reserved word has special meaning for the compiler. A reserved word cannot be used as a data item name or a system name. Reserved words are any of the following:

- Keywords
- Special registers
- Figurative constants

Reserved words must be spelled correctly and can be used only as specified in syntax.

User-Defined Words

A user-defined word can consist of any of the following characters:

- The letters A through Z
- The digits 0 through 9
- Hyphen (-)

A user-defined word must begin with a letter of the alphabet, must not begin or end with a hyphen, and must not contain embedded spaces. User-defined words are used for the following types of items:

- Procedure name
- Data name
- Mnemonic name
- Condition-name
- Program name
- Library name
- Text name

Note. A user-defined word cannot contain double-byte characters.

System Names

A system name is a SCREEN COBOL word that identifies part of the Guardian operating environment. System names are defined for equipment and operating system access. Use of each system name is restricted to a specific category, such as terminal function key or display attribute.

Literals

A literal is a character string whose value is specified either by a set of characters or by a reserved word that represents a figurative constant. A literal is numeric or nonnumeric.

Numeric Literals

A numeric literal is one or more digits (0 through 9), a plus or minus sign, and an optional decimal point. The value of the literal is the value of the digits. The following rules apply to numeric literals:

- A numeric literal can have a maximum of 18 digits.
- One sign character is allowed and must be the first character. The absence of a sign character indicates the literal is a nonnegative number.

- A numeric literal can have one decimal point, which can appear anywhere within the literal except as the last character. The absence of a decimal point indicates that the literal is an integer.

The following examples illustrate numeric literals:

- Integer numeric literals:

```
   +601
-234116
      0
     15
```

- Noninteger numeric literals:

```
+601.1
  89.6
   0.0051
   -.1
```

Nonnumeric Literals

A nonnumeric literal is any ASCII character string enclosed in quotation marks. The value of the literal is the string of characters between the quotation marks. The following rules apply to nonnumeric literals:

- Nonnumeric literals can have a maximum length of 120 characters, not including the surrounding quotation marks.
- If a quotation mark is part of the literal, it must be represented in the string as two contiguous quotation marks. Only one of these two quotation marks is included in the character count.

The following example illustrates a nonnumeric literal:

```
"THIS IS A NONNUMERIC LITERAL"

"12345 THIS IS A NONNUMERIC LITERAL ALSO"
```

The following example illustrates a nonnumeric literal with an embedded quotation mark:

```
"A "" IS PART OF THIS NONNUMERIC LITERAL"
```

Figurative Constants

A figurative constant is a constant that has been prenamed and predefined for the SCREEN COBOL compiler so that it can be written in the source program without having to be defined in the Data Division. Figurative constants do not require quotation marks. Table 2-5 lists and defines the figurative constants. The singular and plural forms of the various figurative constants are equivalent in meaning.

Table 2-5. Figurative Constants

ZERO ZEROS ZEROES	Depending on the context, represents the numeric value 0 or a string of one or more occurrences of the character 0
SPACE SPACES	Represents one or more ASCII space characters (blanks)
HIGH-VALUE HIGH-VALUES	Represents one or more binary 255 values. This value is the highest value that a byte of Working-Storage can contain. This constant can be used to initialize alphabetic and alphanumeric data items only.
LOW-VALUE LOW-VALUES	Represents one or more binary 0 values. This value is the lowest unsigned value that a byte of Working-Storage can contain. This constant can be used to initialize alpha and alphanumeric data items only.
QUOTE QUOTES	Represents one or more quotation mark characters. Neither of these words can be used in place of quotation mark characters around a nonnumeric literal string.
ALL *literal*	Repeats the value of *literal*, which must be a nonnumeric literal or figurative constant other than ALL *literal*. When a figurative constant is used, the word ALL is redundant and is used only for readability.

The following rules apply to figurative constants:

- When a figurative constant represents multiple characters, the length of the string is determined by the compiler.
- A figurative constant can be used wherever a literal appears in a format; when the literal must be numeric, only ZERO, ZEROS, or ZEROES is permitted.
- When a figurative constant is moved or compared to another data item, the figurative constant is repeated on the right until its size is equal to the size of the data item. This repetition of the figurative constant occurs regardless of whether there is a JUSTIFIED clause for the data item.
- Moving any figurative constant except SPACE or SPACES to a PIC N field is flagged as a SCOBOLX compiler error.

Mixed Data Items

Both 1-byte and 2-byte (double-byte) characters can coexist in data items declared as PIC X(n), where n is an integer from 1 through 32,000. Such data items are called mixed data items. You must be aware when manipulating a mixed data item that double-byte characters require two bytes of storage.

Note. On some terminals SO (shift out) and SI (shift in) characters are required when displaying mixed data items, and each such character takes up one byte. SO and SI characters are not required for data items using double-byte characters only.

Mixed data items are allowed in the Working-Storage Section, the Linkage Section, and the Screen Section. A PIC X(10) field can contain, for example, any of the following (without truncation):

- Five 2-byte characters and no other alphanumeric characters
- Four 2-byte characters and one or two 1-byte Katakana or other alphanumeric characters
- Three 2-byte characters and one to four 1-byte Katakana or other alphanumeric characters
- Two 2-byte characters and one to six 1-byte Katakana or other alphanumeric characters
- One 2-byte character and one to eight 1-byte Katakana or other alphanumeric characters
- No 2-byte characters and one to ten 1-byte Katakana or other alphanumeric characters

If the data item contains less than the maximum number of characters allowed, the appropriate number of padding space characters is added to the right.

Reference Format

A SCREEN COBOL source program can be written in Tandem standard reference format (specific to Compaq *NonStop™ Himalaya* systems) or ANSI standard reference format. The Tandem standard reference format has no sequence number field (columns 1-6), has no identification field (columns 73-80), and is restricted to lines of up to 132 characters.

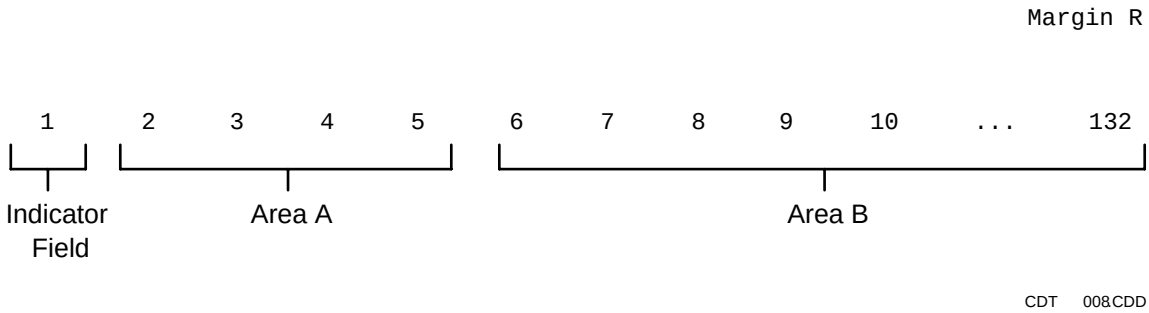
Although the SCREEN COBOL compiler assumes Tandem standard reference format, a SCREEN COBOL program can be written in either format or in a combination of both. Refer to source text options in Running the SCREEN COBOL Compiler on page 7-1 for information regarding format specification.

Tandem Standard Reference Format

Lines in Tandem standard reference format are not fixed length; they can have up to 132 characters. Lines longer than 132 characters are truncated; trailing blanks are ignored. For each line, Margin R is set to follow the last nonblank character in the line, regardless of the Margin R location in any previous line.

Trailing blanks from a previous line and initial blanks on a continuation line are ignored. Figure 2-1 shows the Tandem standard reference format.

Figure 2-1. Tandem Standard Reference Format



- Division, section, and paragraph headers must begin in Area A. The first sentence of a paragraph can begin on the same line as the paragraph header, provided at least one space follows the period terminator of the paragraph name.
- In general, SCREEN COBOL ignores the distinction between Area A and Area B. The lexical elements of a source program can occur anywhere between Margin A and Margin B.
- Level numbers 01 and 77 must begin in Area A.
- Level numbers 02-49, 66, and 88 can begin in either Area A or Area B.
- Area A of a continuation line should always be left blank.
- An * or / in the indicator field indicates a comment; a - indicates continuation; a ? indicates a compiler command line. If any other character appears in the indicator field, the last character in the preceding line is assumed to be followed by a space .

ANSI Standard Reference Format

Each line in ANSI standard reference format has 80 characters. The SCREEN COBOL compiler assures this by truncating lines over 80 characters, or adding blanks to fill out short lines.

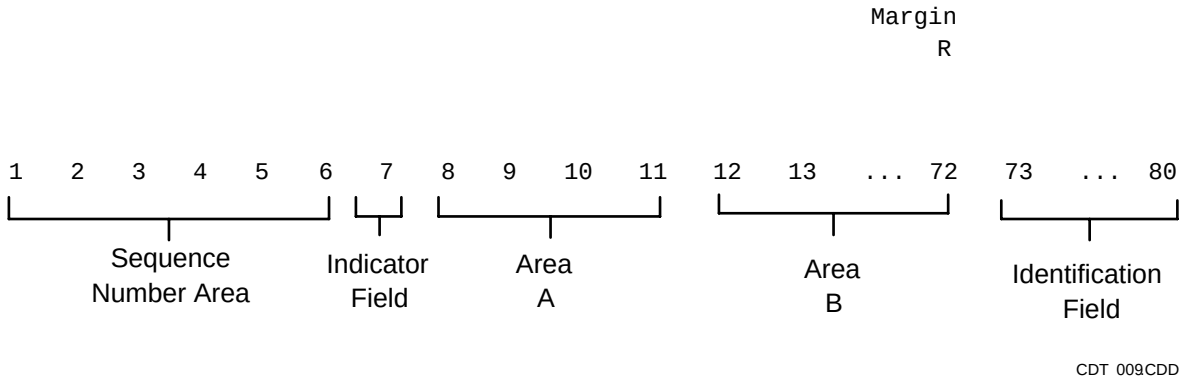
A literal string that exceeds one line must fill the line on which it begins; otherwise, any trailing blanks before the continued characters are included as part of the literal.

The sequence number area (1 through 6) assigns a number to each line of code or labels a line with any combination of ASCII characters.

The positions following Margin R (73 through 80) represent the identification field. Their contents, which can include any ASCII character, are treated as a comment, and have no effect on the meaning of the program.

Figure 2-2 shows the ANSI standard reference format.

Figure 2-2. ANSI Standard Reference Format



- Division, section, and paragraph headers must begin in Area A. The first sentence of a paragraph can begin on the same line as the paragraph header, provided at least one space follows the period terminator of the paragraph name.
- In general, SCREEN COBOL ignores the distinction between Area A and Area B. The lexical elements of a source program can occur anywhere between Margin A and Margin B.
- Level numbers 01 and 77 must begin in Area A.
- Level numbers 02-49, 66, and 88 can begin in either Area A or Area B.
- An * or / in the indicator field indicates a comment; a - indicates continuation; a ? indicates a compiler command line. If any other character appears in the indicator field, the last character in the preceding line is assumed to be followed by a space.

Comment Lines

Comment lines can appear anywhere in a SCREEN COBOL program. Comment lines are indicated by an asterisk (*) or forward slash (/) character in the indicator field, which is column 1 in the Tandem standard reference format and column 7 in the ANSI standard reference format. These special characters indicate that the entire line is a comment.

When a listing of the program is printed, with comment lines indicated by a forward slash (/), a page eject is performed before printing the comment line.

Continuation Lines

Any word or literal in a SCREEN COBOL program can be continued. Continuation lines are indicated by the hyphen character in the indicator field.

If the previous line has a nonnumeric literal without a closing quotation mark, the first nonblank character in Area B of the continuation line must be a quotation mark. The continuation begins with the character immediately following that quotation mark.

Compiler Command Lines

Compiler command lines are indicated by the question mark character in the indicator field. The line is an instruction for the SCREEN COBOL compiler.

Normally, a compiler line in ANSI standard reference format is identified by a question mark in column 7; however, the SCREEN COBOL compiler interprets any line with a question mark in column 1 as a compiler command, even when the ANSI standard reference format is being used. In this special case, the line is treated as beginning with the indicator field; no sequence number area exists. Refer to Section 7, Compilation, for detailed information regarding compiler commands.

Arithmetic Operations

Arithmetic operations are specified in the Procedure Division with the ADD, COMPUTE, DIVIDE, MULTIPLY, and SUBTRACT statements. These operations have the following common features:

- The data descriptions of the operands do not have to be the same. Any necessary conversion and decimal point alignment is supplied throughout the calculation.
- The maximum size of each operand is 18 decimal digits.
- Each arithmetic operation is evaluated using an intermediate data item. If the size of the result being developed is larger than this intermediate data item, the SCREEN COBOL program will be suspended by the TCP with an arithmetic overflow error. The contents of the intermediate data item are moved to the receiving data item according to the rules of a MOVE statement.

When a sending and receiving item in an arithmetic statement share part of their storage areas, the result is undefined.

Arithmetic Expressions

An arithmetic expression is one of the following:

- A numeric elementary item
- A numeric literal
- A numeric elementary item and a numeric literal separated by arithmetic operators
- An arithmetic expression enclosed in parentheses

Data items and literals appearing in an arithmetic expression must be either numeric elementary items or numeric literals on which arithmetic operations can be performed. Any arithmetic expression can be preceded by a plus or minus sign.

Arithmetic Operators

Five binary arithmetic operators and two unary arithmetic operators are used in arithmetic expressions. These operators are represented by specific characters and must be preceded and followed by a space. Table 2-6 lists binary arithmetic operators. Table 2-7 lists unary arithmetic operators.

Table 2-6. Binary Arithmetic Operators

+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation

Table 2-7. Unary Arithmetic Operators

+	The effect of multiplying by +1
-	The effect of multiplying by -1

When a plus or minus sign immediately precedes a numeric literal (with no intervening spaces) the sign becomes a part of that literal, making it a signed numeric literal. The sign is neither a binary or unary operator. For example,

```
X +2
```

is equivalent to:

```
X, +2
```

which is two separate expressions.

A plus sign in any other situation is treated as a binary operator if it is preceded by an operand, and treated as a unary operator if it is not preceded by an operand. For example, the following are equivalent expressions:

```
X + 2
X + + 2
```

Evaluation of Expressions

Parentheses can be used to specify the order in which the operations of an arithmetic expression are performed. Expressions within parentheses are evaluated first.

Evaluation of expressions within nested parentheses proceeds from the innermost set to the outermost set. When parentheses are not used, or expressions in parentheses are at the same level, the order of execution is as follows:

1. Unary plus and minus
2. Multiplication and division
3. Addition and subtraction

Parentheses are used to eliminate ambiguities in logic or to modify the normal sequence of execution in expressions where it is necessary to have some deviation. When the sequence of execution is not specified by parentheses, the order for consecutive operations at the same level is from left to right. The following example illustrates the normal evaluation order in the absence of parentheses:

```
a + b / c + d * f - g
```

is interpreted as:

```
(a + (b / c)) + (d * f) - g
```

with the sequence of operations proceeding from the innermost parentheses to the outermost. Expressions ordinarily considered ambiguous, such as:

```
a / b * c, a / b / c
```

are permitted in SCREEN COBOL. They are interpreted as if they were written:

```
(a / b) * c, (a / b) / c
```

Data items and literals appearing in an arithmetic expression must represent either numeric elementary data items or numeric literals.

Multiple Results

The ADD, COMPUTE, MULTIPLY, and SUBTRACT statements can have multiple results. Such statements behave as though they had been written in the following way:

1. One statement performs all necessary arithmetic to arrive at a result, and stores that result in a temporary storage location.
2. A sequence of statements transfers or combines the value of this temporary location with each result. These statements are considered to be written in the same left-to-right sequence in which the multiple results are listed.

For example, the result of the following statement:

```
ADD  a, b, c TO c, d(c), e
```

is equivalent to:

```
ADD a, b, c GIVING temp
ADD temp TO c
ADD temp TO d(c)
ADD temp TO e
```

where *temp* is the temporary storage location.

Intermediate Results

Intermediate results are maintained by SCREEN COBOL during the evaluation of arithmetic expressions. The maximum number of digits held for an intermediate result is 18. If this limit is exceeded, arithmetic overflow occurs. Table 2-8 uses the following abbreviations to explain intermediate operations:

- IR—the number of integer places carried for an intermediate result
- DR—the number of decimal places carried for an intermediate result
- OP1—the first operand in an arithmetic expression, which has the form 9(I1)V9(D1), where:
  - I1 is the number of integer places carried.
  - D1 is the number of decimal places carried for the first operand.
- OP2—the second operand in an arithmetic expression, which has the form 9(I2)V9(D2), where:
  - I2 is the number of integer places carried.
  - D2 is the number of decimal places carried for the second operand.
- OPR—the desired result, which has the form 9(IR)V9(DR), where IR is the number of places carried for the integer result and DR is the number of places carried for the decimal result

Table 2-8. Digits Held for Intermediate Results

Operation	Decimal Places
OP1 + OP2	DR is the greater of D1 or D2.IR is the lesser of (the greater of I1 or I2) or 18-DR.
OP1 - OP2	DR is the greater of D1 or D2.IR is the lesser of (the greater of I1 or I2) or 18-DR.
OP1 * OP2	DR is the sum (D1 + D2).IR is the lesser of (I1+I2) or 18-DR.
OP1 / OP2	DR is the greater of D1 or 1.IR is the lesser of (I1+D2) or 18-DR
OP1 ** OP2	OP2 is restricted to integer values.OP2's value is any literal or a data item.(For example: DR is D1; IR is 18-DR.)

Note. If (I1+D2+DR) is greater than 18, the low-order digits of the quotient are lost; in other words,any part of the quotient less than the following number is lost: 10 ** (I1+D2+DR-18)

The three examples that follow illustrate the division operation.

Example 1

A normal divide computation proceeds as follows:

```
03 A1 PIC S9(9)V9(9) VALUE 2.
03 A2 PIC S9(9)V9(8) VALUE 3.
03 AR PIC SV9(9).
       :
DIVIDE A1 BY A2 GIVING AR.
```

where:

```
                _____________
3.00000000 | 2.000000000
```

is computed as:

```
                      000000000.666666666
                    _____________________
000000003.00000000 | 2.000000000000000000
```

then moved to AR as:

```
.666666666
```

Example 2

Here is a second example:

```
03 A1 PIC S9(2)V9(9) VALUE 2.
03 A2 PIC S9(2)V9(8) VALUE 3.
03 AR PIC SV9(9).
       :
DIVIDE A1 BY A2 GIVING AR.
```

where:

```
                _____________
3.00000000 | 2.000000000
```

is computed as:

```
                 000000000.666666666
               ____________________
03.00000000 | 02.000000000000000000
```

then moved to AR as:

```
.666666666
```

When a division operation in an arithmetic expression involves a COMPUTE statement or a relational expression, the intermediate results are evaluated in two steps:

1. The actual division
2. The adjustment of that result for use in further computations

Example 3

In the following example, with either of the following:

```
COMPUTE AX = A1/A2 + A3 * A4.

IF A1/A2 + A3 * A4 LESS THAN AX GO TO ...
```

the division is performed before further evaluation of either of the above statements. The intermediate result is then adjusted to fit the conceptual PICTURE derived by examining the other operands in the expression.

Incompatible Data

An incompatible data condition occurs when a data item is referenced by a statement in the Procedure Division and that item contains characters not permitted by the statement. For example, if a position in a display numeric item contains an alphabetic character, A, and that item is used as an operand in an ADD statement, an incompatible data condition occurs. The result of this reference is undefined.

The class condition test is an exception to this rule because its purpose is to determine whether or not items contain legal data.

Conditional Expressions

Conditional expressions identify conditions that are tested by the program to select between alternate paths of control. Conditional expressions are specified in the IF and PERFORM statements.

The two categories of conditions for conditional expressions are: simple conditions and complex conditions. Either kind of condition can be enclosed within any number of paired parentheses without changing the category of the condition.

Simple Conditions

Simple conditions are of four kinds:

- Class conditions
- Condition-name conditions
- Relation conditions
- Sign conditions

Simple conditions have a truth value of true or false. Parentheses can enclose a simple condition without changing the truth value of the condition.

Class Condition

The class condition determines whether a DISPLAY item value is numeric or alphabetic.

Class condition syntax is:

```
data-name [ IS ] [ NOT ] { NUMERIC    }
                         { ALPHABETIC }
```

When NOT is included, the test condition is reversed. NOT NUMERIC tests for a field being nonnumeric; NOT ALPHABETIC tests for a field being nonalphabetic.

The NUMERIC test cannot be used with an item described as alphabetic. The NUMERIC test cannot be used with a group item composed of elementary items with data descriptions that include operational signs. If the data item being tested is signed, the item is numeric only if the contents are numeric and a valid sign is present. If the item is not signed, the item passes the test only if the contents are numeric and no sign is present. Valid signs for items with SIGN IS SEPARATE clause are + and -.

The ALPHABETIC test cannot be used with an item described as numeric.

Condition-Name Condition

A condition-name condition determines whether or not the value of a conditional variable is equal to one of the values predefined for the condition-name.

Condition-name condition syntax is:

```
condition-name
```

The *condition-name* must be a level 88 item defined in the Data Division and given a value or a range of values.

The condition is true if the value of the conditional variable is equal to one of the *condition-name* values or falls within one of the ranges of values (including both ends of the range) given with *condition-name*.

Relation Condition

A relation condition causes a comparison of two values. Each value can be a data item, a literal, or a value resulting from an arithmetic computation; both values cannot be literals. A relation condition has a truth value of true if the relation exists between the values.

Relation condition syntax is:

```
value-1 IS { [ NOT ] { LESS [ THAN ] }     } value-2
           {         { <             }     }
           {                               }
           { [ NOT ] { EQUAL [ TO ] }      }
           {         { =             }     }
           {                               }
           { [ NOT ] { GREATER [ THAN ] } }
           {         { >                 } }
```

The relational operators <, =, > (less than, equal to, greater than) determine the type of comparison made. A space must precede and follow each word of the relational operator. When NOT is included, the word NOT and the next keyword or relation character are one operator. NOT EQUAL is a truth test for an unequal comparison; NOT GREATER is a truth test for an equal or less comparison.

Two numeric values can be compared regardless of their usage (as defined by a USAGE clause). For all other comparisons, however, the values must have the same usage. If either of the values is a group item, nonnumeric comparison rules apply.

Comparison of Numeric Operands

Comparison of numeric operands is made with respect to the algebraic value of the operands. The length of the literal or arithmetic expression operands, in terms of the number of digits represented, is not significant. Zero is considered a unique value regardless of the sign.

Comparison of these operands is permitted regardless of the manner in which their usage is described. Unsigned numeric operands are considered positive.

Comparison of Nonnumeric Operands

Comparison of nonnumeric operands, or one numeric and one nonnumeric operand, is made with respect to the ASCII collating sequence of characters. The size of an operand is its total number of characters.

A noninteger numeric operand cannot be compared to a nonnumeric operand.

Numeric and nonnumeric operands can be compared only when their usage is the same. The following conventions apply:

- The numeric operand must be an integer data item or an integer literal.
- If the nonnumeric operand is an elementary data item or a nonnumeric literal, the numeric operand is treated as though it were moved to an elementary alphanumeric

data item of the same size as the numeric data item; the content of this alphanumeric data item is then compared to the nonnumeric operand.

- If the nonnumeric operand is a group item, the numeric operand is treated as though it were moved to a group item of the same size as the numeric data item; the content of this group item is then compared to the nonnumeric operand.

Comparison of Equal-Sized Operands

If the values of operands are equal in size, characters in corresponding positions are compared starting from the high order end. The comparison continues until either a pair of unequal characters is found or the low order end is reached. The values are equal when all pairs of characters are the same.

The first pair of unequal characters is compared to determine their relative position in the collating sequence. The value having the character that is higher in the collating sequence is the greater value.

Comparison of Unequal-Sized Operands

If the values of operands are unequal in size, comparison proceeds as though the shorter operand were extended on the right by sufficient spaces to make the operands equal in size.

Sign Condition

The sign condition determines whether or not the algebraic value of an arithmetic expression is greater than, less than, or equal to zero.

Sign condition syntax is:

```
arithmetic-expression [ IS ] [ NOT ] { POSITIVE }
                                     { NEGATIVE }
                                       { ZERO     }
```

arithmetic-expression must have at least one variable.

When NOT is included, the word NOT and the next keyword specify one sign condition that defines the algebraic test to be executed for truth value. NOT ZERO is a truth test for a nonzero, positive, or negative value. An item is positive if its value is greater than zero, negative if its value is less than zero, and zero if its value is equal to zero.

Complex Conditions

Complex conditions are formed by using simple conditions, combined conditions and/or complex conditions with logical connectives AND or OR, or by negating these conditions with the keyword NOT. The truth value of a complex condition, whether or not the value is enclosed in parentheses, is that truth value which results from the interaction of all the logical operators on the individual truth values of simple conditions, or on the intermediate truth values of conditions connected or negated.

Table 2-9 lists and defines the logical operators.

Table 2-9. Logical Operators

AND	Logical conjunction—The truth value is true if both conditions are true, and false if one or both are false.
OR	Logical inclusive OR—The truth value is true if one or both of the conditions is true, and false if both conditions are false.
NOT	Logical negation or reversal of truth value—The value is true if the condition is false and false if the condition is true.

The logical operators must be preceded by a space and followed by a space.

Negated Simple Condition

A simple condition is negated through the use of the logical operator NOT. The negated simple condition effects the opposite truth value for a simple condition. Parentheses enclosing negated simple condition do not change the truth value.

Negated simple condition syntax is:

```
NOT simple-condition
```

Combined and Negated Combined Conditions

A combined condition results from connecting conditions with AND or OR. Each condition can be a simple condition, a negated condition, a combined condition or negated combined condition, or a combination of these.

Combined and negated combined condition syntax is:

```
condition {  { AND } condition }
          {  { OR  }           }
```

Abbreviated Combined Relation Conditions

In a relation where one item is compared to several others, the relation can be abbreviated by leaving out the subject item name after the first reference to it. If the relational operator is the same as the previous operator, the operator can also be omitted.

Abbreviated combined relation condition syntax is:

```
condition { { AND } [ NOT ] [ operator ] object }...
          { { OR  }                              }
```

If NOT appears within the abbreviated condition and is not followed by an operator, the keyword negates that portion of the condition, but does not automatically carry forward

to the next relation. The following examples illustrate abbreviated combined relation conditions and their expanded equivalents.

Abbreviated Combined Relation Condition	Expanded Equivalent
`a > b AND NOT < c OR d`	`((a > b) AND (a NOT < c))` `OR (a NOT < d)`
`a NOT EQUAL b OR c`	`(a NOT EQUAL b) OR (a NOT EQUAL c)`
`NOT a = b OR c`	`(NOT (a = b)) OR (a = c)`
`NOT (a GREATER b OR < c)`	`NOT ((a GREATER b) OR (a < c))`

`NOT (a NOT > b AND c`
`AND NOT d)`
	`NOT ((((a NOT > b)`
	`AND (a NOT > c))`
	`AND (NOT (a NOT > d))))`

`(a + b - c) > d`
`AND NOT < e OR f`
	`(a + b - c) > d`
	`AND (a + b - c) NOT < e`
	`OR (a + b - c) NOT < f`

Condition Evaluation Rules

Parentheses are used to change the order in which individual conditions are evaluated when it is necessary to depart from the standard order. Conditions within parentheses are evaluated first. When conditions are within nested parentheses, evaluation goes from the innermost condition to the outermost condition.

When parentheses are not used or when conditions in parentheses are at the same level, the following order of evaluation is used until the final truth value is determined:

1. Values are established for arithmetic expressions.
2. Truth values for simple conditions are established in the following order:
   a. Relation conditions
   b. Class conditions
   c. Condition-name conditions
   d. Sign conditions
3. Truth values for negated simple conditions are established.
4. Truth values for combined conditions are established: AND logical operators, followed by OR logical operators.
5. Truth values for negated combined conditions are established.
6. When the sequence of evaluation is not completely specified by parentheses, the order of evaluation of consecutive operations of the same hierarchical level is from left to right.

Tables

Tables of data are common in data processing problems. For example, a data structure might have 20 total fields, described as twenty identical data items named total-one, total-two, ..., total-twenty. This would mean twenty different names, which could obscure the interrelated nature of the totals and make references awkward. A table structure simplifies this situation.

Tables are defined with an OCCURS clause in their data description. This clause specifies that an item is repeated as many times as stated. The item is considered to be a table element, and its name and description apply to each repetition. For example, the one-dimensional table mentioned in the preceding paragraph could be defined with this entry:

```
02  total  OCCURS 20 TIMES ...
```

In the Screen Section, a table must be an elementary item. In the Working-Storage Section and Linkage Section, the elements of a table can be groups of subordinate structures, some of which can also be tables. Thus, the previous example might appear in greater detail as:

```
02  total-g  OCCURS 20 TIMES.
    03 total-a ...
    03 total-b  OCCURS 3 TIMES ...
```

The expanded example describes total-a as a one-dimensional table, and describes total-b as a two-dimensional table because an OCCURS clause is applied to an item subordinate to the first OCCURS clause. If the description of a data item subordinate to total-b also had an OCCURS clause, the item would be a three-dimensional table. SCREEN COBOL allows a maximum of three dimensions in the Working-Storage Section and Linkage Section.

Frequently, tables are built in the Working-Storage Section with constant values that a program needs in addition to the data from external sources. An example of coding for a table containing the full calendar month names is:

```
WORKING-STORAGE SECTION.
01  month-name-table.
    05  FILLER        PIC X(9)  VALUE "JANUARY".
    05  FILLER        PIC X(9)  VALUE "FEBRUARY".
    05  FILLER        PIC X(9)  VALUE "MARCH".
    05  FILLER        PIC X(9)  VALUE "APRIL".
    05  FILLER        PIC X(9)  VALUE "MAY".
    05  FILLER        PIC X(9)  VALUE "JUNE".
    05  FILLER        PIC X(9)  VALUE "JULY".
    05  FILLER        PIC X(9)  VALUE "AUGUST".
    05  FILLER        PIC X(9)  VALUE "SEPTEMBER".
    05  FILLER        PIC X(9)  VALUE "OCTOBER".
    05  FILLER        PIC X(9)  VALUE "NOVEMBER".
    05  FILLER        PIC X(9)  VALUE "DECEMBER".
01  month-name-rtable REDEFINES month-name-table.
    05  month-name  OCCURS 12 TIMES   PIC X(9).
```

The term FILLER is a keyword that takes the place of a data name when it is unimportant to name an item. Because occurrences of a table element do not have

individual names, a reference to an occurrence must give its position number along with the data name of the table. The method of giving the position number, called subscripting, is described later in this section.

Data Reference

All items must be named so they can be referenced. Items given unique names can be referenced with no difficulty, but many programs contain items that do not have unique names. All elements of a table, for example, share a single name. Also, the same name can be used for more than one data item, and the same paragraph name can be used in different sections of the Procedure Division.

Names must be unique or made unique through qualification or subscripting.

Qualification

Every name must be unique, either because no other name has the same spelling and hyphenation, or because the name is subordinate to a unique name. In the latter case, including one or more of the higher-level names qualifies the subordinate item and makes it unique. Although enough qualification must be present to make a name unique, it is not necessary to include all levels.

- For data name references, group names can be used for qualification. Level 01 names are the most significant qualifiers, then level 02, and so forth.
- For condition-name references, the name of the condition variable can be used as qualification, even if the variable is an elementary item.
- For paragraph name references, the section name is the only qualifier available. References to paragraphs within the same section never require qualification.
- For copy text references in COPY statements, the copy text name must be qualified if the text library that defines it is not the default library for the compilation.
- Level 01 names and section names must be unique because they cannot be further qualified. Regardless of available qualification, a name cannot be both a data name and a procedure name.

An item is qualified by following a data name, a condition-name, a paragraph name, or a copy text name by one or more phrases composed of a qualifier preceded by connective IN or OF. IN and OF are equivalent.

Qualification syntax is:

```
{ data-name      } [ { OF } qualification-name ] ...
{ condition-name } [ { IN }                   ]

paragraph-name [ { OF } section-name ]
               [ { IN }              ]

copy-text [ { OF } library-name ]
          [ { IN }              ]
```

The qualification rules are as follows:

- Each qualifier must be at a higher level than the previous one and must stay within the same structure of the name it qualifies.
- The same name cannot appear at different levels in a structure; otherwise, the name could qualify itself.
- If a data name or a condition-name is assigned to more than one data item, the data name or condition-name must be qualified each time it is referenced (except in the REDEFINES clause where, by context, qualification is unnecessary).
- A paragraph name cannot be duplicated within a section. Within its own section a paragraph name does not require qualification. When a section name is used to qualify a paragraph name, the word SECTION is not part of the name.
- A data name used as a qualifier is not subscripted, even if the data name is described with an OCCURS clause.
- A name can be qualified even when the name is unique.
- If more than one combination of qualifiers is available to make a name unique, any combination can be used.

In the following example, all data names except *prefix* are unique. Qualification must be used to reference either *prefix* item.

```
01 transaction-data ...        01 master-data ...
   03 item-no ...                  03 code-no ...
      05 prefix ...                    05 prefix ...
      05 code ...                      05 suffix ...
   03 quantity ...                 03 description ...
```

Using the same example, any of the following sentences could be used to move the contents of one *prefix* to the other *prefix*:

```
MOVE prefix OF item-no TO prefix OF code-no.
MOVE prefix OF item-no TO prefix OF master-data.
MOVE prefix OF transaction-data TO prefix IN code-no.
MOVE prefix IN transaction-data TO prefix IN master-data.
```

Subscripting

Subscripts are used to reference elements in a table. They are needed because all table elements have the same name.

The subscript can be an integer numeric literal or a data item that represents a numeric integer. When the subscript is a data item, the data item name can be qualified but not subscripted itself. The subscript can be signed and, if signed, it must be positive.

The lowest possible subscript value is 1. This value selects the first element of a table. The other elements of the table are selected by subscripts whose values are 2, 3, 4, and so forth. If a subscript value greater than the size of the table is used, the result is undefined.

The subscript, or set of subscripts, is enclosed in parentheses and is appended to the element name of the table. When more than one subscript is required, they are written in the order of most significant value to least significant value.

Subscript syntax is:

```
{ data-name      } ( sub-1 [ , sub-2 [ , sub-3 ] ] )
{ condition-name }
```

Note that a multiple-subscripted data item must have a space character preceding all subscripts except the first.

The following examples illustrate subscripting:

```
MOVE total(8) TO report-total-8.
MOVE day of date(3) TO print-line-date.
MOVE month-name(month-number) TO report-month.
MOVE matrix(row, column) TO output-display-line.
```

Referencing a subscripted data item defined using a PIC X(n) clause (where n is an integer from 1 through 32,000) containing double-byte data redefined as PIC X(1) OCCURS n TIMES might reference only the left or right byte of a double-byte character. The left or right byte of a double-byte character by itself has no meaning and therefore the byte is undefined. For example:

```
WORKING-STORAGE SECTION.
   :
01 WS-KANJI-DATA                        PIC N(05).
01 WS-UNDEFINED-DATA                    PIC X.
01 WS-NAME-1                            PIC N(05).
01 WS-GROUP-REDEF REDEFINES WS-NAME-1.
   02 WS-BYTE-DATA                      PIC X OCCURS 10 TIMES.
      :
      :
PROCEDURE DIVISION.
      :
      :
  MOVE WS-KANJI-DATA TO WS-NAME-1.
  MOVE WS-BYTE-DATA(1) TO WS-UNDEFINED-DATA.
```

The receiving data in the example is undefined because the left or right byte of a double-byte character has no meaning.

Note. Arrays defined using a PIC N clause are referenced in units of two bytes.

Using Identifiers

An identifier is a data name made unique by qualifiers, subscripts, or qualifiers and subscripts. A data name being used as a subscript or qualifier cannot itself be subscripted.

Identifier syntax is:

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

  [ ( sub-1 [ , sub-2 [ , sub-3 ] ] ) ]
```

The following examples illustrate specification of identifiers:

```
unique-identifier

item-1 OF group-a

element OF name-table OF master-data (master-num)
```

Using Condition-Names

Items are tested frequently by a program. Assigning a condition-name to an item is a convenient way to refer to the item and determine its value.

Every condition-name must be unique or capable of being made unique through qualification and/or subscripting. If qualification is used to make a condition-name unique, the conditional variable can be used as the first qualifier. The containing data names of the conditional variable can also be used as qualifiers. If references to a conditional variable require subscripting, then any of its condition-names must have the same subscripting.

The following example illustrates a condition-name called restricted-use:

```
01 inventory.
   02 part-number OCCURS 100 TIMES ...
      03 prefix              PIC 99.
      03 use-code            PIC 9.
         88 restricted-use          VALUE 1.
      03 supplier-suffix     PIC 99.
```

The condition-name, `restricted-use`, might be referenced as:

```
IF restricted-use OF use-code IN part-number (30)
    NEXT SENTENCE,
ELSE...
```

Data Representation

In the Working-Storage Section and Linkage Section, data items are stored in a certain number of bytes; each byte is an 8-bit unit of storage. Bytes are grouped in pairs to form words.

Data items whose usage (as defined by a USAGE clause) is DISPLAY occupy one byte per character. Table 2-10 indicates the storage occupied by data items whose usage is COMPUTATIONAL.

Table 2-10. Storage Occupied by COMPUTATIONAL Data Items

PICTURE Size in Digits	Storage Occupied
1 through 4	2 bytes
5 through 9	4 bytes
10 through 18	8 bytes

In the Screen Section, items do not have individual storage assigned; storage of these items is of no consequence to SCREEN COBOL programming.

Standard Alignment

The standard rules for positioning data within an elementary item depend on the category of the receiving item. The rules are as follows:

- If the receiving data item is described as numeric, the sending data is aligned either by decimal point with zero fill on either end of the value or by truncation on the low end, as required. Truncation on the high end is not permitted, and if required, causes suspension of the program. When no decimal point is specified, the receiving data item is treated as if it had an assumed decimal point immediately following the rightmost character.
- If the receiving data item is described as alphanumeric or alphabetic, the sending data is aligned at the leftmost character position in the data item with space fill or truncation to the right as required.

Optional Alignment

Standard data representation and alignment rules are not always appropriate, so provisions exist to override them. The JUSTIFIED clause can be used in the data description to right justify data within a data item.

Sometimes a server requires that data items in messages be aligned on word boundaries. Data items aligned on word boundaries are said to be synchronized. Synchronization typically is achieved by organizing and describing data so that item boundaries coincide with word boundaries. This task can be eliminated by using the SYNCHRONIZED clause to force alignment of data items to their natural boundaries.

3 Identification Division

The Identification Division identifies the SCREEN COBOL program. The division has one required paragraph and five optional paragraphs. If other paragraphs are present, they are treated as comments.

The format of the Identification Division is:

```
IDENTIFICATION DIVISION.

PROGRAM-ID. program-unit-name.

[ AUTHOR. [ comment-entry ] ]

[ INSTALLATION. [ comment-entry ] ]

[ DATE-WRITTEN. [ comment-entry ] ]

[ DATE-COMPILED. [ comment-entry ] ]

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

The division header is:

```
IDENTIFICATION DIVISION.
```

The header must be terminated with a period separator.

Optional paragraphs AUTHOR, INSTALLATION, DATE-WRITTEN, and SECURITY are included for documentation purposes only. The *comment-entry* parameter can be any combination of characters from the SCREEN COBOL character set and represents text appropriate for the paragraph heading.

PROGRAM-ID Paragraph

The required PROGRAM-ID paragraph names the SCREEN COBOL program unit. The syntax of the PROGRAM-ID paragraph is:

```
PROGRAM-ID. program-unit-name.
```

program-unit-name

is the user-defined name of the SCREEN COBOL program unit. The name must follow the rules for user-defined names (see Section 2, SCREEN COBOL Source Program) and can differ from the file name of the source code or the object file. This name is used in a CALL statement when the program is referred to in another SCREEN COBOL program unit. This name is also used by the PATHCOM SET TERM INITIAL and SET PROGRAM TYPE commands.

DATE-COMPILED Paragraph

The optional DATE-COMPILED paragraph causes the compiler to generate the current date and time and insert it in the corresponding line of the source listing. The syntax of the DATE-COMPILED paragraph is:

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

comment-entry

is any combination of characters from the SCREEN COBOL character set.

When this paragraph is included, the compiler generates the current date and time, replacing the DATE-COMPILED line and any *comment-entry* with this line:

```
DATE COMPILED. yyyy/mm/dd - hh:mm:ss
```

yyyy

is the year and ranges from 0000 through 9999.

mm

is the month and ranges from 01 through 12.

dd

is the day and ranges from 01 through 31.

4 Environment Division

The Environment Division declares the operating environment of the program unit and provides optional error reporting for screen input operations. The division has two sections:

- A required Configuration Section
- An optional Input-Output Section

The syntax of the Environment Division is:

```
ENVIRONMENT DIVISION.

  CONFIGURATION SECTION.

    SOURCE-COMPUTER. comment-entry

    OBJECT-COMPUTER. object-computer-entry

    [ SPECIAL-NAMES. special-names-entry ]

  [ INPUT-OUTPUT SECTION. input-output-entry ]
```

The division header is:

```
ENVIRONMENT DIVISION.
```

The header must be terminated with a period separator.

Configuration Section

The required Configuration Section declares the operating environment of the program unit. These declarations can include terminal type characteristics and screen display attributes.

The section header is:

```
CONFIGURATION SECTION.
```

The header must begin in Area A and must be terminated with a period separator.

The Configuration Section contains two required paragraphs, the SOURCE-COMPUTER and OBJECT-COMPUTER paragraphs, and an optional SPECIAL-NAMES paragraph.

SOURCE-COMPUTER Paragraph

The required SOURCE-COMPUTER paragraph names the computer system by which the program unit is compiled. The SCREEN COBOL compiler assumes the system is a Compaq *NonStop™ Himalaya* system and treats any name given as a comment.

The SOURCE-COMPUTER paragraph syntax is:

```
SOURCE-COMPUTER. comment-entry.
```

comment-entry

is one or more words and cannot consist of blank or null characters.

OBJECT-COMPUTER Paragraph

The required OBJECT-COMPUTER paragraph names the computer system on which the object program runs. The SCREEN COBOL compiler assumes the system is a NonStop™ Himalaya system and treats the name given as a comment.

The OBJECT-COMPUTER paragraph syntax is:

```
OBJECT-COMPUTER. comment-word,

[ TERMINAL IS { T16-6510       } [ , ] ]
[             { T16-6520       }       ]
[             { T16-6530       }       ]
[             { T16-6540       }       ]
[             { IBM-3270       }       ]
[             { CONVERSATIONAL }       ]
[             { INTELLIGENT-0  }       ]
[             { INTELLIGENT-1  }       ]
[             { INTELLIGENT-2  }       ]
[             { INTELLIGENT    }       ]

[ CHARACTER-SET IS { USASCII        } ] .
[                  { FRANCAIS-AZ    } ]
[                  { FRANCAIS-QW    } ]
[                  { DEUTSCH        } ]
[                  { ESPANOL        } ]
[                  { UK             } ]
[                  { SVENSK-SUOMI   } ]
[                  { DANSK-NORSK    } ]
[                  { KANJI-KATAKANA } ]
```

comment-word

is a single word.

TERMINAL IS

specifies whether the program is a screen-oriented requester program that communicates with a terminal or a message-oriented requester program that

communicates with an intelligent device. If the TERMINAL IS clause is omitted, the program is assumed to be a screen-oriented requester program operating in block mode; the requester program can run on any of the block mode terminal types listed for a screen-oriented requester program. However, features unique to a particular terminal cannot be used if the TERMINAL IS clause is omitted.

For a screen-oriented requester program you can specify the type of terminal for which the requester program is intended and the type of communication, block mode or conversational.

```
T16-6510
T16-6520
T16-6530
T16-6540
```

specify the particular terminal operating in block mode.

- Program units compiled for a T16-6520 terminal can be run on a T16-6530 or T16-6540 terminal.
- Program units compiled for a T16-6530 can be run on a T16-6540.
- Program units compiled for one terminal type and run on another terminal type can use only those features available for the compiled terminal type.

```
IBM-3270
```

specifies an IBM 3270 type terminal operating in block mode.

```
CONVERSATIONAL
```

denotes any of the following terminals operating in conversational mode: 6510, 6520, 6530, or 6540 type terminals, IBM 3270 type terminals, or any other device that the operating system recognizes as operating in conversational mode.

Features unique to terminal types in block mode are not recognized for the same terminal types in conversational mode.

For Intelligent Device Support (IDS), the message-oriented requester program can communicate with AM3270, SNAX, TERMPROCESS, AM6520, X25AM, or a *Guardian* operating environment process. The TERMINAL IS clause specifies whether the terminal communicates in conversational mode or block mode.

```
INTELLIGENT-0
```

specifies conversational mode. WRITEREAD I/O protocol: write to a device and wait for a reply in conversational mode.

```
INTELLIGENT-1
```

specifies block mode. WRITE and READ I/O protocol: write to a device and then read from the device in block mode.

INTELLIGENT-2

specifies block mode. WRITEREAD I/O protocol: write to a device and wait for a reply in block mode.

INTELLIGENT

specifies that the PATHCOM SET TERM TYPE parameter determines conversational or block mode. If INTELLIGENT is specified but no PATHCOM SET TERM TYPE parameter is specified, the default is conversational mode.

The syntax for PATHCOM to enable these settings is given under the description of the SET TERM command in the *Compaq NonStop™ Pathway/iTS System Management Manual*.

Selecting INTELLIGENT-0 is appropriate if the device is to be placed in conversational mode and the device is configured to communicate with a Compaq access method for NonStop™ Himalaya systems, such as SNAX or X25AM. If INTELLIGENT-0 is selected and the TCP executes a SEND MESSAGE statement without a REPLY phrase, only a WRITE is issued to the device. If INTELLIGENT-0 is selected and the TCP executes a SEND MESSAGE statement without data, a WRITEREAD with a write count of 0 is issued.

Selecting INTELLIGENT-1 is appropriate if it is desired to have the device in block mode and the device is configured to communicate with a Compaq access method for NonStop™ Himalaya systems, such as SNAX or X25AM. If INTELLIGENT-1 is selected and the TCP executes a SEND MESSAGE statement without a REPLY phrase, a READ is not issued. If INTELLIGENT-1 is selected and the TCP executes a SEND MESSAGE statement without data, a WRITE is not issued.

Selecting INTELLIGENT-2 is appropriate if it is desired to have the device in block mode and the device is configured such that other processes are simultaneously requesting data from it. If this is the case, the WRITEREAD protocol guarantees that each reply is returned to its intended requester. If INTELLIGENT-2 is selected and the TCP executes a SEND MESSAGE statement without a REPLY phrase, a WRITE is issued. If INTELLIGENT-2 is selected and the TCP executes a SEND MESSAGE statement without data, a READ is issued.

When communicating with Guardian user processes, INTELLIGENT-0 or INTELLIGENT-2 should be used in order to invoke a WRITEREAD call to communicate with the process.

CHARACTER-SET IS

specifies a character set other than USASCII. This clause can be used only with terminal types IBM-3270, T16-6530, and T16-6540. For other terminal types, the USASCII character set is assumed. If specified, this clause must follow the TERMINAL IS clause. The only language that can be declared for IBM 3270 terminals is KANJI-KATAKANA.

If this clause is omitted for a program running on an IBM 3270 or a 6540 terminal, USASCII is used. If this clause is omitted for a program compiled for and running on a 6530 terminal, USASCII is used until the first DISPLAY BASE statement is

executed. After the first DISPLAY BASE, the character set specified in the terminal's configuration menu is used.

KANJI-KATAKANA

indicates that the program unit source file might contain double-byte characters in data fields or literals and instructs the compiler to allow the PIC N picture clause.

You must be aware that depending on the terminal, support for 1-byte Katakana characters, 2-byte Katakana characters, and lowercase and uppercase alphabetic characters varies.

- On 6530 terminals, both 1-byte and 2-byte Katakana character sets, as well as uppercase and lowercase alphabetic characters, are supported if CHARACTER-SET IS KANJI-KATAKANA is specified.
- On IBM 3270 terminals, either lowercase alphabetic characters or 1-byte Katakana characters are supported.
  - On IBM 3270 terminals configured to use both lowercase and uppercase alphabetic characters, a program unit cannot use 1-byte Katakana characters.
  - On IBM 3270 terminals configured to use 1-byte Katakana characters, the TCP upshifts lowercase alphabetic characters in the outbound data stream. Such terminals can run applications with both lowercase and uppercase alphabetic characters as well as 1-byte Katakana characters, but inbound data does not contain lowercase alphabetic characters that the terminal does not generate when thus configured.

Note. Applications that use 1-byte alphabetic characters only can run on any terminal for which the CHARACTER-SET IS KANJI-KATAKANA clause is valid. Applications that use 1-byte alphabetic characters as well as 1-byte Katakana characters can run only on a 6530 terminal or on an IBM 3270 terminal specifically configured to support EBCDIC/Katakana.

If the CHARACTER-SET IS clause is included and the character set type differs from the current setting in the terminal, or the terminal setting is unknown, the terminal is signaled the character set type at the execution of the first DISPLAY BASE statement in a program unit. After the program unit completes execution, the terminal is reset to its original character set.

The programmatic support of national-use characters affects the following areas:

- Field-characteristic clause UPSHIFT—Lowercase national-use characters are upshifted to their uppercase equivalents.
- Class condition—The condition ALPHABETIC checks for characters in the national-use characters.
- Symbol A in PICTURE clauses—A check is made for characters in the national-use character set.

Programmatic support of national-use characters does not affect the following areas:

- Field-characteristic clause MUST BE—Range tests are not supported for national use characters.
- Tests that involve collating sequence matters—Any comparison tests, such as less-than or greater-than relations, are not supported for national-use characters.

SPECIAL-NAMES Paragraph

The optional SPECIAL-NAMES paragraph allows you to select names and to have those names assigned to certain system names. The paragraph also matches features of a specific terminal with the words used in the program to refer to those features. With careful use of the correspondences established in the SPECIAL-NAMES paragraph, you can remove much of the dependence on terminal type from the body of the program unit.

The SPECIAL-NAMES paragraph syntax is:

```
SPECIAL-NAMES.

  [ { mnemonic-name IS { sys-name                } } ,... ]
  [ {                  { ( { sys-name } ,... ) } }      ]

  [ , CURRENCY [ SIGN ] IS literal-1 ]

  [ , DECIMAL-POINT IS COMMA ] .
```

mnemonic-name

is an identifier you select to be associated with a *sys-name* and can be used later in the Screen Section or the Procedure Division of the program to refer to a function key or display attribute indicated by *sys-name*.

A list of system names can be equated to a single *mnemonic-name* only if the system names refer to display attributes that can be combined. This causes the *mnemonic-name* to represent the combination of the display attributes. Except for terminals in the IBM 3270 family, only highlight display attributes can be combined.

sys-name

specifies a function key or display attribute available on the terminal. Table 4-1 lists the system names for function keys; Table 4-2 lists the system names for display attributes.

Table 4-1. System Names for Function Keys

System Name	6510	6520	6530/6540	3270
F1 through F16 (unshifted)	x(1)	x	x	
SF1 through SF16 (shifted)	x	x	x	
LINE-INS (unshifted)		x	x	
LINE-DEL (shifted)		x	x	
NEXT-PAGE (unshifted)		x	x	
S-NEXT-PAGE (shifted)		x	x	
PREV-PAGE (unshifted)		x	x	
S-PREV-PAGE (shifted)		x	x	
ROLL-DOWN (unshifted)		x	x	
S-ROLL-DOWN (shifted)		x	x	
ROLL-UP (unshifted)		x	x	
S-ROLL-UP (shifted)		x	x	
RETURN-KEY (2)			x	
CLEAR				x
ENTER				x
PA1 through PA3				x
PA4 through PA10				x
PF1 through PF24				x

1. x in the cells above indicates that the system name is valid for the terminal type.
2. You can define a return-key function in the SPECIAL-NAMES paragraph of a SCREEN COBOL program that runs on a 6530 or 6540 terminal. If a return-key function is defined, a function code is transmitted when the RETURN key is pressed; otherwise, pressing RETURN causes only a forward tab action. A return-key function is local to a program unit. The first DISPLAY-BASE statement causes the terminal to adjust the return-key operation to the setting indicated by the executing program unit.

Table 4-2. System Names for Display Attributes (page 1 of 2)

System Name	6510	6520	6530/6540	3270	CONVERSATIONAL (4)
BLUE				x(1)	
RED				x	
PINK				x	
GREEN				x [D](2)	
TURQUOISE				x	
NEUTRAL (3)				x	
YELLOW				x	
HIDDEN	x	x	x	x	x
NOTHIDDEN	x [D]	x [D]	x [D]	x [D]	x [D]
PROTECTED	x	x	x	x	
UNPROTECTED	x [D]	x [D]	x [D]	x [D]	
BLINK	x	x	x	x	
NOBLINK	x [D]	x [D]	x [D]	x [D]	
MDTOFF		x [D]	x [D]	x [D]	
MDTON				x	
NUMERIC-SHIFT				x	
BRIGHT				x	
NORMAL	x [D]	x [D]	x [D]	x [D]	
DIM		x	x	x	
REVERSE		x	x	x	
NOREVERSE		x [D]	x [D]	x [D]	
UNDERLINE (5)		x	x	x	
NOUNDERLINE		x [D]	x [D]	x [D]	
BELL				x	x
NOBELL				x [D]	x [D]
TOPLINE				x	
LEFTLINE				x	
NOTOPLINE				x [D]	

1. x in the cells above indicates that the system name is valid for the particular terminal type.
2. [D] refers to the default display attribute value for the particular terminal type.
3. The NEUTRAL system name actually causes the generation of white terminal characters.
4. Applies to any terminal specified as CONVERSATIONAL in the OBJECT-COMPUTER paragraph.
5. BOTTOMLINE and UNDERLINE are not the same.

Table 4-2. System Names for Display Attributes (page 2 of 2)

System Name	6510	6520	6530/6540	3270	CONVERSATIONAL (4)
NOLEFTLINE				x [D]	
RIGHTLINE				x	
NORIGHTLINE				x [D]	
BOTTOMLINE (5)				x	
NOBOTTOMLINE				x [D]	
BOXFIELD				x	

1. x in the cells above indicates that the system name is valid for the particular terminal type.
2. [D] refers to the default display attribute value for the particular terminal type.
3. The NEUTRAL system name actually causes the generation of white terminal characters.
4. Applies to any terminal specified as CONVERSATIONAL in the OBJECT-COMPUTER paragraph.
5. BOTTOMLINE and UNDERLINE are not the same.

`CURRENCY [ SIGN ] IS` *`literal-1`*

specifies a literal to be used instead of the dollar currency sign ($) and must be a single character and cannot be any of the following:

- Digits—0 through 9
- Characters—A B C D L P R S V X Z space
- Symbols—* + - , . ; ( ) " / =

`DECIMAL-POINT IS COMMA`

exchanges the function of comma and period in PICTURE character strings and numeric literals in the remainder of the program.

The following example illustrates the SPECIAL-NAMES paragraph:

```
SPECIAL-NAMES.
  ENTER-KEY   IS F1,
  EXIT-KEY    IS F16,
  INPUT-ATTR  IS UNDERLINE,
  SIGNAL-ATTR IS (REVERSE, NOUNDERLINE).
```

You must observe the following restrictions when combining display attributes in a mnemonic name declared in the SPECIAL-NAMES paragraph:

- You must not combine extended field attributes for 3270 terminals that Pathway does support with those that Pathway does not support. Pathway does not support DIM on 3270 terminals.
- You can combine any of the following display attributes with one another:

  ```
  BRIGHT
  HIDDEN
  MDTON
  ```

```
NUMERIC-SHIFT
PROTECTED
```

- You can combine one of the following highlight display attributes with one or more of the other highlight display attributes (that is, BRIGHT, HIDDEN, MDTON, NUMERIC-SHIFT, or PROTECTED).
- If a 3270 terminal supports the following outline display attributes, you can combine them with one another and with any highlight display attribute (that is, with BRIGHT, HIDDEN, MDTON, NUMERIC-SHIFT, PROTECTED, BLINK, REVERSE, or UNDERLINE.):

  ```
  TOPLINE or NOTOPLINE
  LEFTLINE or NOLEFTLINE
  RIGHTLINE or NORIGHTLINE
  BOTTOMLINE or NOBOTTOMLINE
  BOXFIELD
  ```

- You can combine outline display attributes with one or more highlight display attributes. For example, you can set up fields that combine a highlight display attribute with the attribute for a blinking field:

  ```
  (BLINK, TOPLINE, BOTTOMLINE)
  (REVERSE, LEFTLINE, RIGHTLINE)
  (UNDERLINE, BOTTOMLINE, TOPLINE)
  ```

Input-Output Section

The optional Input-Output Section provides error reporting for screen input operations. If this section is omitted, the error display attribute is dependent on the terminal type specified in the Configuration Section.

The section header is:

```
INPUT-OUTPUT SECTION.
```

The header must begin in Area A and must be terminated with a period separator.

The Input-Output Section syntax is:

```
SCREEN-CONTROL.
  ERROR-ENHANCEMENT [ IS ] mnemonic-name [ IN { FIRST } ]
                                           [    { ALL   } ]
  [ WITH [ NO ] AUDIBLE ALARM ] .
```

`ERROR-ENHANCEMENT [ IS ] mnemonic-name`

specifies the display attribute with which fields found to be in error are to be enhanced. *mnemonic-name* must be specified in the SPECIAL-NAMES Paragraph.

If this clause is omitted for terminals in block mode, the BLINK attribute is used for the 6510, 6520, 6530, and 6540 terminals; the BRIGHT attribute is used for the

IBM 3270 terminal. If this clause is omitted for terminals in conversational mode, no error enhancement occurs. For error enhancement, BELL must be specified.

`IN FIRST`

enhances the first field that is found to be in error. For terminals operating in conversational mode, IN FIRST is the only recognized enhancement option.

`IN ALL`

enhances all fields that are found to be in error. If IN ALL is not specified, only the first field containing an error is enhanced.

`WITH [ NO ] AUDIBLE ALARM`

enables or disables the audible indicator when an error is detected.

The ERROR-ENHANCEMENT option allows you to control some aspects of the error processing. For example, when an ACCEPT statement executes, the TCP checks the contents of input fields against the requirements of a PICTURE clause and any constraints, such as those imposed by a MUST BE field-characteristic clause. ACCEPT processing attempts to indicate which field is in error.

5 Data Division

The Data Division describes the data the program creates, accepts as input, manipulates, or produces as output. The Data Division has four sections:

- A Working-Storage Section
- A Linkage Section
- A Screen Section
- A Message Section

Each section is optional and is included only when the type of data the section defines is used in the program. Data described in the Data Division falls into three categories:

- Data developed internally by the program and placed in temporary areas described in the Working-Storage Section or Linkage Section
- Data specifically formatted for display on a terminal or received as input from a terminal. This data passes through the Screen Section.
- Data that is unformatted. This unformatted data allows for device independence because it is not formatted to match any specific device. This data passes through the Message Section.

The division begins with a division header. The format of the header is:

```
DATA DIVISION.
```

The header must be terminated with a period separator.

The format of the Data Division is:

```
DATA DIVISION.

[ WORKING-STORAGE SECTION.
    data-description-entries ]

[ LINKAGE SECTION.
    data-description-entries ]

[ SCREEN SECTION.
  [ input-control-entries ]
  screen-description-entries ]

[ MESSAGE SECTION.
  message-description-entries ]
```

Data Division Sections

The four sections of the Data Division each describe a different type of data. The sections are defined as follows:

- The Working-Storage Section describes the structure of local data developed within the program. Data entries in this section are initialized each time the program unit is called; therefore, values are not retained between calls.
- The Linkage Section describes the structure of parameter data passed to a subprogram by a CALL statement. Items described in the calling program are referred to in the USING clause of the Procedure Division header of a called program.
- The Screen Section describes the types and locations of fields in screens that can be displayed on the terminal. Screens described in the Screen Section are referred to in the Procedure Division of the program.
- The Message Section describes the data type, size, and relative position (sequence) of each field in a message. This section also defines the editing and conversion that must be performed on each field. Messages described in the Message Section are referred to in the Procedure Division of the program.

When multiple sections are included in a program, they must appear in the order shown. Items within each section can appear in any order.

Although a program can contain both a Screen Section and a Message Section, it cannot use both sections at once. A program either displays and accepts data on a terminal screen or it communicates with an intelligent device; it cannot do both. The TERMINAL IS clause of the OBJECT-COMPUTER paragraph dictates whether you can reference the Screen Section or the Message Section in the Procedure Division of your program.

Working-Storage Section

The Working-Storage Section defines records and miscellaneous data items used for internal purposes. Data entries in this section can be set to initial values. When local data items or intermediate storage are not needed, this section can be omitted.

The section begins with a section header. The format of the header is:

```
WORKING-STORAGE SECTION.
```

Data description entries for individual items follow the header. All item names must be unique. Subordinate data names can be duplicated as long as they can be qualified.

The maximum size of elementary items, group items and 01 level data items in the Working-Storage Section is 32,000 bytes. The maximum size of a data item that contains only double-byte characters is 16,000 double-byte characters (32,000 bytes) for a Working-Storage Section entry.

An 01 level data item can have a maximum size of 32,000 bytes and can comprise elementary and/or group items. For example:

```
WORKING-STORAGE SECTION.
    01 BIG-01-ITEM                PIC X(32000) VALUE SPACES.
    01 BIG-GROUP-ITEM-1.
       03 BIG-GROUP.               PIC X(16000) VALUE SPACES.
    01 BIG-GROUP-ITEM-2.
       03 BIG-GROUP.
          05 BIG-ITEM-2-A.         PIC X(16000) VALUE SPACES.
          05 BIG-ITEM-2-B.         PIC X(16000) VALUE SPACES.
    01 BIG-ELEMENTARY-ITEM-01.
       03 BIG-ELEMENTARY-GROUP.
          05 BIG-ELEMENTARY-ITEM PIC X(32000) VALUE SPACES.
```

Note. When very large data items are displayed using the Compaq *Inspect* debugging tool, truncation might occur.

Linkage Section

The Linkage Section describes data passed from the calling program to the program containing the Linkage Section (the called program). The Linkage Section associates the data items defined in the section with data items defined in the Working-Storage Section of the calling program. A Linkage Section is required in a called program even when no data is passed.

The section begins with a section header. The format of the header is:

```
LINKAGE SECTION.
```

Definitions in the Linkage Section should be the same size as the corresponding items in the Working-Storage Section of the calling program. If the definitions are larger, then an error is returned; if the definitions are smaller, then data might be truncated but no error is returned.

The calling program must contain a USING clause in the CALL statement to refer to the data structures to be passed to the called program. The called program must contain a USING clause in the Procedure Division header to refer to the data structures being passed to it.

The Linkage Section does not cause the system to allocate additional memory in the called program. The called and calling programs share the calling program's memory for the common data structures.

The structure of the Linkage Section is the same as that of the Working-Storage Section except the VALUE clause is prohibited for items other than level 88 items.

The maximum size of elementary items, group items, and 01 level data items in the Linkage Section is 32,000 bytes. The maximum size of a data item that contains only double-byte characters is 16,000 double-byte characters (32,000 bytes) for a Linkage Section entry. A more detailed explanation of this size limit is provided in the subsection SCREEN COBOL Limits on page 7-19.

An 01 level data item can have a maximum size of 32,000 bytes and can comprise elementary and/or group items. For example:

```
LINKAGE SECTION.
    01 BIG-01-ITEM                PIC X(32000).
    01 BIG-GROUP-ITEM-1.
       03 BIG-GROUP.               PIC X(16000).
    01 BIG-GROUP-ITEM-2.
       03 BIG-GROUP.
          05 BIG-ITEM-2-A.        PIC X(16000).
          05 BIG-ITEM-2-B.        PIC X(16000).
    01 BIG-ELEMENTARY-ITEM-01.
       03 BIG-ELEMENTARY-GROUP.
          05 BIG-ELEMENTARY-ITEM PIC X(32000).
```

Note. When very large data items are displayed using INSPECT, truncation might occur.

Screen Section

The Screen Section describes the screens that are referred to in the Procedure Division. The structure of the Screen Section is similar to that of the Working-Storage Section. The section makes provision for two types of screens: base and overlay.

The section begins with a section header. The format of the header is:

```
SCREEN SECTION.
```

Message Section

The Message Section describes the messages that are referred to in the Procedure Division. The structure of the Message Section is similar to that of the Working-Storage Section.

The section begins with a section header. The format of the header is:

```
MESSAGE SECTION.
```

Data Structure

Data is described through a set of entries that name the components of a structure, describe the attributes of those components, and describe the structure into which the components are organized. Each entry has a level number followed by a data name and possibly a series of independent clauses. The level numbers depict the structure, dividing the data into its smallest parts.

The lowest subdivisions of a structure, that is, those not further subdivided, are called elementary items. A structure can be a single elementary item or a series of elementary items.

Sets of elementary items can be referred to by combining them into groups. Groups, in turn, can be combined into groups; an elementary item, therefore, can belong to more than one group.

Level Numbers 01-49

Level numbers 01 through 49 describe the hierarchy of data items. The structure itself is assigned level number 01.

The system of level numbers shows the relationship of elementary items to group items. Data items within a group are assigned level numbers higher than that of the group item. Level numbers within the group need not be consecutive, but they must be ordered so that the higher the level number the lower the entry in the hierarchy.

A group includes all group and elementary items following it until a level number less than or equal to the level number of that group is encountered. All items or groups immediately subordinate to a given group item must be described using identical level numbers greater than the level number of that group item.

An example of level numbering is the following:

```
01  address-data.
    05  office-number.
        10  district       PIC 99.
        10  region         PIC 999.
    05  office-address.
        10  street         PIC X(25).
        10  city           PIC X(15).
        10  state          PIC X(5).
        10  zip-code       PIC 9(5).
01  personnel-data.
    05  office-manager     PIC X(35).
    05  no-of-employees    PIC 9(4).
    05  tax-groups.
        10  hourly         PIC 9(3).
            15  part-time  PIC 99.
            15  full-time  PIC 99.
        10  exempt         PIC 9(4).
```

Level Numbers 66, 77, and 88

Three additional types of data entries can exist in the Working-Storage Section and Linkage Section: level 66, level 77, and level 88. Entries that begin with these level numbers do not define the hierarchy of the item described.

Level number entries define items as follows:

- Level 66 specifies elementary items or groups introduced by a RENAMES clause. These entries are used to regroup contiguous elementary data items.
- Level 77 specifies an independent data item that is not a subdivision of another data item. The data item is not itself subdivided.

- Level 88 defines a condition name, including a value or range of values that define the condition to be tested.

Data Description Entry

A data description entry defines the characteristics of a data item. The entry can be used in the Working-Storage Section or Linkage Section of the SCREEN COBOL program.

When the program unit is compiled, numeric elementary data items in the Working-Storage Section that do not include an OCCURS clause are initialized to zero.

Several forms are available to describe items for various purposes. Some entries cause the creation of items (memory space is allocated), while others supply alternative descriptions or reference points for already existing data. Others supply specification of value ranges for later testing.

The syntax of the data description entry is:

```
{ WORKING-STORAGE SECTION. }
{ LINKAGE SECTION.         }
```

Format 1 is:

```
level-number { data-name-1 }
             { FILLER      }
  [ JUSTIFIED clause ]
  [ OCCURS clause ]
  [ PICTURE clause ]
  [ REDEFINES clause ]
  [ SIGN clause ]
  [ SYNCHRONIZED clause ]
  [ USAGE clause ]
  [ VALUE clause ]
```

Note. The VALUE clause in Format 1 applies to the Working-Storage Section only.

Format 2 is:

```
[ 66 new-name [ RENAMES clause ] ]
```

Format 3 is:

```
[ 88 condition-name , [ VALUE clause ] ]
```

- Format 1 describes data of levels 01 through 49 and level 77. The *data-name-1* entry is the name of the storage area defined by the subordinate items. In the following example, store-address refers to everything from street through zip-code.

```
01  sample-record.
    05  store-id.
        10  store-number       PIC 999.
        10  store-region       PIC X.
    05  store-manager          PIC X(35).
    05  store-address.
        10  street             PIC X(25).
        10  city               PIC X(15).
        10  state              PIC X(2).
        10  zip-code           PIC 9(5).
    05  FILLER                 PIC X(14).
```

  The FILLER keyword takes the place of a data name when it is unimportant to name an item. FILLER is commonly used when building Working-Storage records, such as error messages, where most of the text is groups of constants. The text groups can be separated by the filler. In the following example, FILLER defines an area in storage that cannot be referred to in the program except as part of the enclosing item, first-record:

```
01  first-record.
    05  record-code     PIC 99.
    05  record-type     PIC XX.
    05  FILLER          PIC X(30).
    05  division-code   PIC 999.
```

  A level 77 entry cannot itself be subdivided. Level 77 entries, like level entries 01 through 49, must be immediately followed by a data name or keyword FILLER. For example:

```
01  first-record.
    05 record-code      PIC 99.
    05 record-type      PIC XX.
77  temp-1              PIC X(4).
77  temp-2              PIC X(3).
```

  Various examples of level 77 items appear in Section 6.

- Format 2 describes a level 66 entry, which renames one or more contiguous elementary items. In the following example, the group card-codes is renamed code:

```
05  card-codes.
    10 store-code       PIC 9.
    10 state-code       PIC 9(4).
66  code RENAMES card-codes.
```

- Format 3 describes a level 88 entry, which assigns condition name values. In the following example, item tax-code is defined with a range of values:

```
05  tax-code            PIC 99.
    88  tax-range        VALUES ARE 01 THRU 20.
```

JUSTIFIED Clause

The JUSTIFIED clause causes nonstandard positioning of data within a receiving item. The clause can only appear in the data description of an elementary item; the clause cannot be used for a data item that is described as numeric.

```
{ JUST      } RIGHT
{ JUSTIFIED }
```

When the JUSTIFIED clause is omitted, standard alignment rules dictate that alignment is left justified and truncation or padding, when necessary, occurs on the right.

When a receiving data item is described with the JUSTIFIED clause, the standard alignment rules do not apply. If a sending item is too big for the receiving item, the sending item is truncated on the left. If the sending item is smaller than the receiving item, the rightmost character of the sending item is aligned with the rightmost character of the receiving field and the value is extended to the left with space characters.

Right justification does not strip trailing blanks from the sending field. Suppose, for example, that the Working-Storage Section includes:

```
01 WS-FIELD-1        PIC X(12)

01 WS-FIELD-2        PIC X(14)       JUSTIFIED RIGHT.
```

In addition, in the Screen Section a field is defined as:

```
05 SCREEN-FIELD-1   PIC X(10)
                     USING WS-FIELD-1.
```

Suppose the user enters “ABCDEFGHIJ” into SCREEN-FIELD-1, presses a valid function key, and an ACCEPT statement causes data to be moved into WS-FIELD-1. The data is then moved from WS-FIELD-1 to WS-FIELD-2 using the MOVE statement. As a result of this sequence of operations, the three fields then appear as follows:

```
SCREEN-FIELD-1 = "ABCDEFGHIJ"   (ten bytes)
WS-FIELD-1     = "ABCDEFGHIJ  " (twelve bytes)
WS-FIELD-2      = "  ABCDEFGHIJ  " (fourteen bytes)
```

Although right justification and leading-blank padding have occurred, trailing blanks from the input field were not stripped. Trailing blanks can be stripped from screen input fields during the execution of an ACCEPT statement by means of an alphanumeric input user conversion routine. Alternatively, logic for stripping trailing blanks can be included in a server program.

Note. The JUSTIFIED clause is ignored when an item with the literal given in a VALUE clause is initialized.

OCCURS Clause

The OCCURS clause defines tables and other sets of repeating items, thus eliminating the need for separate item entries. These tables can be a fixed number of elements or can vary within given limits. An OCCURS clause cannot be used in an 01 level entry.

Format 1 (for Fixed Length Tables) is:

```
OCCURS max [ TIMES ]
```

max

is an integer that represents the number of elements in the table.

Format 2 (for Variable Length Tables) is:

```
OCCURS min TO max [ TIMES ] DEPENDING [ ON ] depend
```

min

is an integer that represents the smallest number of elements in the table at any time. The integer must be zero or greater and less than or equal to *max*.

max

is an integer that represents the greatest number of elements the table can have at any time.

depend

is an integer data item that controls the size of the table. As the value of the *depend* item increases or decreases, the number of elements in the table increases or decreases. When the table size decreases, those elements beyond the new *depend* limit are lost even if the next statement increases the table to include them. When the table size increases, you must assign values to the new elements before using them.

The following example illustrates the OCCURS clause:

```
01 table-group.
   02  activity-count          PIC 99.
   02  activity-table          OCCURS 10 TO 20 TIMES
                                 DEPENDING ON activity-count.
       05  activity-entry      PIC 999.
```

When using the data name that represents a table item, you must use subscripts to access the item. You can use the data name without subscripts only when you want the entire table (for example, in a MOVE statement). If the data name is a group item, you must use subscripts for all items belonging to the group whenever they are used as operands. Subordinate data names used as objects of a REDEFINES clause are not considered operands and, therefore, cannot be subscripted.

A data description entry with an OCCURS DEPENDING ON clause can be followed, within its data description, only by descriptions of subordinate items. In other words, only one table with a variable number of occurrences can appear in a single data description, and the data items contained by the table must be the last data items in the data description.

Data items subordinate to an entry described with an OCCURS clause can themselves contain an OCCURS clause. Tables can consist of such multiple occurrences of subordinate tables for a maximum of three levels. A data description entry containing either format of the OCCURS clause can be followed by subordinate entries containing a fixed length table OCCURS clause; however, a data description entry with an OCCURS DEPENDING ON clause cannot be subordinate to a group entry described with either format of the OCCURS clause.

PICTURE Clause

The PICTURE clause defines the characteristics of an elementary item.

```
{ PIC     } [ IS ] character-string
{ PICTURE }
```

character-string

is one or more symbols that determines the category of an elementary item and places restrictions on the values assignable to the item.

A maximum of 30 characters is allowed in *character-string*. When the same PICTURE character repeats, you can write it once followed by an unsigned integer enclosed in parentheses. The integer indicates how many times that character is repeated. For example, the following PICTURE clauses are equivalent:

```
PIC 9(5)
```

```
PIC 99999.
```

Although a character string can be no longer than 30 characters, you can use the repetition technique to define items that otherwise would be longer than 30 characters.

Table 5-1 lists the character-string symbols that are used to describe a data item.

Table 5-1. Data Description Entry PICTURE Character-String Symbols

Symbol	Meaning
A	Represents a character position for a letter of the alphabet or a space character. The symbol is counted in the size of the data item.
N	Represents a double-byte character and is valid only in program units that specify the KANJI-KATAKANA keyword in the CHARACTER-SET IS clause of the OBJECT-COMPUTER paragraph in the Environment Division. The symbol is counted in the size of the data item.
P	Indicates scaling when the decimal point is not among or adjacent to the digits of the data item stored. The symbol is counted in determining the maximum number of digit positions in numeric items (the maximum is 18). One or more P symbols can appear only as a contiguous string to the left or right of all other digit positions in the PICTURE string. The P symbol is redundant when used with the V symbol because P implies an assumed decimal point. If an operation involves conversion of data from one form of internal representation to another and the data item being converted is described with the P symbol, each digit position described by a P is considered to have the value zero. The size of the data item includes those digit positions.
S	Represents a signed numeric value. The symbol is counted in the size of the item only if a SIGN IS SEPARATE clause is used.
V	Represents the decimal point location in noninteger numeric items. The symbol is not counted in the size of the item.
X	Represents a character position that can have any character from the ASCII character set and/or double-byte character sets. The symbol is counted in the size of the item.
9	Represents a character position for a digit. The symbol is counted in the size of the item.

Item Size

The size of a data item is determined by the symbols in its PICTURE string. Each A, X, and 9 count as one character position. An S counts as one character only if the item is subject to a SIGN IS SEPARATE clause.

If a data item is described as DISPLAY in a USAGE clause, the size of the item includes the PICTURE string symbols. If the item is described as COMPUTATIONAL, the size of the item is computed differently, as described under the USAGE clause.

Categories of Data

The PICTURE clause can describe categories of data: alphabetic, numeric, alphanumeric, and double-byte data items. The results of most statements in the Procedure Division depend on the categories of the data items. Some statements require certain categories for some or all of their operands. In some cases, a statement can take different actions depending on the category of the data items.

In the discussion that follows, 9 and A symbols within the PICTURE string are described as representing character positions that have only numbers or letters and spaces. For reasons of efficiency, the SCREEN COBOL compiler does not always require this restriction. Characters other than those permitted can be moved into these positions if they appear in the corresponding group positions of a sending data item. SCREEN COBOL considers every group item to be alphanumeric. Manipulations on group items ignore all PICTURE strings. For example, a move operation into a group item can cause any position of an item to contain any ASCII character.

Alphabetic Data

An alphabetic data item can have only A symbols in the PICTURE string. The contents of this type of item are represented externally as some combination of the 26 letters of the alphabet and the space character.

The following examples illustrate alphabetic data:

```
05  package-code    PIC AAA.
05  dept-id         PIC AA(6)AA.
05  dept-code       PIC AA(2)AA.
```

Numeric Data

A numeric data item can have 9, P, S, and V symbols in the PICTURE string. The number of digits described must be greater than zero but not more than 18. The contents of this type of item are represented externally as a combination of digits 0 through 9.

If the item is signed, a plus or minus is included when the data is moved to a screen item, or when a SIGN IS SEPARATE clause is specified. In all other instances, the sign is encoded within one of the digits.

The following examples illustrate numeric data:

```
05  division-total      PIC S9(10)V99.
05  fraction-amount     PIC PP99.
```

Alphanumeric Data

An alphanumeric data item can have combinations of A, X, and 9 symbols in the PICTURE string, but the item is treated as though the string contained all X symbols. The length of the item must be greater than zero but not more than 32,000 bytes. The contents of the item can be any combination of ASCII characters. A PICTURE string of all A symbols or all 9 symbols is not an alphanumeric item.

The following examples illustrate alphanumeric data:

```
10  stock-item-name      PIC X(25).
10  zone-id              PIC A(4)99.
```

Double-Byte Data

A double-byte data item that allows only double-byte data contains only N symbols in the PICTURE string.

If a VALUE clause is declared for a PIC N Working-Storage Section field, the value can consist only of characters from the Shift-JIS character set (X 0208) enclosed in quotation marks ("").

You can use the THRU/THROUGH clause with level 88 data items associated with double-byte character set literals. Byte-by-byte comparisons of all items in the THRU/THROUGH clause are performed. Double-byte character set data used in the THRU/THROUGH clause is treated as a byte string.

The following examples illustrate PICTURE strings for double-byte data:

```
10  data-id             PIC N.
10  data-name-1         PIC NNNN.
10  data-name-2         PIC N(10).
```

REDEFINES Clause

The REDEFINES clause allows a computer storage area to be described in more than one way. This capability is valuable for such tasks as input data validation when tests require different descriptions of the data. This capability is convenient when some portions of a record are constant, while other portions vary.

```
REDEFINES data-name-2
```

data-name-2

is the data item being redefined.

The REDEFINES entry must immediately follow the entry for the data item being redefined or must immediately follow the last item subordinate to that data item. The level number of the REDEFINES entry must be the same as the item being redefined by the clause.

The following rules apply to the REDEFINES clause:

- Level 66 and level 88 data items cannot be redefined.
- The redefined data item cannot have an OCCURS clause or a REDEFINES clause.
- The data name of the redefined item cannot be subscripted or qualified.
- Neither the original definition nor the redefinition can include an item whose size is variable due to an OCCURS clause of a subordinate entry.
- A VALUE clause cannot be included.
- When the level number is not 01, the redefinition should be the same as the number of character positions (bytes) in the data item you are redefining..
- The redefined item can be subordinate to an item with an OCCURS clause or a REDEFINES clause.
- The REDEFINES entry can be followed by subordinate data entries. Redefinition continues until the appearance of a level number less than or equal to that of the data name being redefined or until the ending of the current section of the Data Division.

- You can use a REDEFINES clause to redefine a PIC N field, which allows only double-byte data, as an alphanumeric PIC X field. Doing so makes it possible to move double-byte data items to alphanumeric data items. For example:

```
WORKING-STORAGE SECTION.
    :
01 WS-KANJI-ONLY-FIELD      PIC N(10)
01 WS-KANJI-TO-PIC-X-REDEF  REDEFINES WS-KANJI-ONLY
                              FIELD PIC X(20)
```

The REDEFINES clause redefines a storage area, not the data items occupying the area. Multiple redefinition of the same area is permitted, but all definitions must begin with a REDEFINES clause containing the data name of the entry that originally defined the area.

The following example illustrates the REDEFINES clause:

```
WORKING-STORAGE SECTION.

01  record-in.
    05  record-code        PIC 9.
    05  record-detail      PIC X(30).
    05  record-subtotal    PIC 9(3)V99.
01  record-total REDEFINES record-in.
    05  total-1            PIC 9(5)V99.
    05  total-2            PIC 9(5)V99.
    05  total-3            PIC 9(5)V99.
    05  total-4            PIC 9(5)V99.
    05  total-5            PIC 9(6)V99.
    05  total-5-sub REDEFINES total-5 PIC X(8).
```

RENAMES Clause

The RENAMES clause assigns a new data name to one or more contiguous elementary items within a data description. RENAMES does not cause any allocation of storage. The clause can only be used with a level 66 entry.

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

new-name

is the new name for a group item or elementary item.

old-name

is a group item, an elementary item, or the first of several items to be given a new name.

end-name

is the last group item or elementary item to be included in the new name.

The RENAMES clause merely renames a group of existing data items and does not redescribe any of their characteristics; therefore, no other clauses can be used. One or more RENAMES entries can be written for a structure; these entries can occur in any order, but must immediately follow the last data description entry of the structure.

When the THROUGH option is not specified, *new-name* merely renames *old-name*. *new-name* is a group item only if *old-name* is a group item.

When the THROUGH option is specified, the following rules apply:

- *old-name* and *end-name* must be data areas within the same structure.
- *old-name* and *end-name* cannot have the same names, but the names can be qualified.
- *old-name* and *end-name* cannot be the names of data entries with level number 01, 77, 66, or 88.
- *old-name* and *end-name* cannot be described by an OCCURS clause in their definitions, and they cannot be subordinate to an item described by an OCCURS clause.
- *end-name* cannot name an item that occupies character positions preceding the beginning of the area described by *old-name*.
- *end-name* cannot name an item that is subordinate to *old-name.*
- Items within the renamed area cannot be described by an OCCURS clause.

When the THROUGH option is specified, *new-name* is a group item that includes all elementary items within the bounds established by *old-name* and *end-name*. The following defines the beginning and end of the group:

- If *old-name* is an elementary item, the new group item begins with *old-name.*
- If *old-name* is a group item, the new group item begins with the first elementary item of *old-name.*
- If *end-name* is an elementary item, the new group item ends with *end-name.*
- If *end-name* is a group item, the new group item ends with the last elementary item of *end-name.*

The following example illustrates the RENAMES clause:

```
05  card-codes.
    10  store-code          PIC 9.
    10  state-code          PIC 9(4).
05  account-number          PIC 9(6).
05  check-digit             PIC 9.
66  card-number RENAMES card-codes THRU check-digit.
```

SIGN Clause

The SIGN clause specifies the position and mode of an operational sign for a numeric data item. The clause can only be used for items that are described as DISPLAY in a USAGE clause and have an S symbol in the PICTURE string.

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

`LEADING`

indicates the sign is at the beginning of the item.

`TRAILING`

indicates the sign is at the end of the item.

`SEPARATE [ CHARACTER ]`

specifies the sign becomes a separate character and is counted in the size of the item. A + for positive and a – for negative is placed at the beginning or end of the item value.

If this phrase is omitted, the sign is not counted in the size of the item. Depending on whether you specify LEADING or TRAILING, the sign is at the beginning or end of the item.

The following example illustrates the SIGN clause:

```
05  WS-subtotal-value  PIC S9(02) SIGN IS TRAILING SEPARATE.
```

SYNCHRONIZED Clause

The SYNCHRONIZED clause forces alignment of an elementary item on the most natural computer storage boundary.

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

`RIGHT` and `LEFT`

have no effect in SCREEN COBOL.

A VALUE clause must not appear for any group item that has a subordinate item described with the SYNCHRONIZED clause.

In most cases, the alignment supplied automatically by the compiler is the most natural; however, the SYNCHRONIZED clause affects alignment in a few special cases. Alignment considerations are as follows:

- Alignment requirements can cause SCREEN COBOL to generate implicit FILLER data. The existence of this generated data must be accounted for in certain situations.

- DISPLAY items are composed of one or more character positions and are stored as an equal number of 8-bit bytes. The byte boundary is their natural storage boundary; therefore, the SYNCHRONIZED clause has no effect on DISPLAY item alignment.
- COMPUTATIONAL items are stored as an even multiple of bytes. Their most natural storage unit is some multiple of the 16-bit computer word; each of these words contains two bytes. The SCREEN COBOL compiler automatically aligns COMPUTATIONAL items to word boundaries. This is also the natural boundary for small COMPUTATIONAL items (those items with PICTURE strings containing up to four 9s).
- Larger COMPUTATIONAL items (those items with pictures containing five or more 9s) are naturally stored as one or two 32-bit doublewords. The SYNCHRONIZED clause affects these items; it forces alignment on a doubleword boundary.
- All items of levels 01 and 77 in the Working-Storage Section and Linkage Section are automatically allocated by the SCREEN COBOL compiler to begin on a word boundary. The compiler treats these items as simultaneously beginning on a byte, word, and doubleword boundary. Thus, each of these items is aligned to its most natural storage boundary.
- Words begin on two-byte boundaries; doublewords begin on four-byte boundaries. Alignment, either automatic or as requested by use of the SYNCHRONIZED clause, generates implicit FILLER data in some cases.
    - If an odd number of character positions precedes a word-aligned item within a record, the compiler inserts one character position (byte) of FILLER data before the item to complete allocation of the preceding word.
    - If the number of character positions preceding a doubleword aligned item within a record is not a multiple of four, the compiler inserts FILLER data (1, 2, or 3 bytes) to complete allocation of the preceding doubleword. These extra bytes are not part of the data item.
    - If a group item contains two items separated by implicit FILLER bytes, these bytes are a part of that group item. A group item always begins with the first character position of its first elementary item, however, ignoring any implicit FILLER bytes that were generated to align that item properly. Thus, the initial character positions of a group item are never implicit FILLER bytes.
- Special considerations apply when aligning an elementary data item that is described with an OCCURS clause, is subordinate to a group item described with an OCCURS clause, or both. In these cases, all occurrences of the data item must be aligned uniformly.
    - The first occurrence of the item is aligned to the required storage boundary (if the elementary item also begins a containing table's first occurrence, that table's first occurrence is defined to begin at the first character position of the item). When the aligned item is itself a table, the first occurrence ends on the appropriate storage boundary (byte, word, doubleword) and the remaining occurrences follow without additional FILLER bytes.

- When the aligned item (or table of aligned items) belongs to a higher-level table, further adjustment might be necessary. If the elementary item is word-aligned and the containing group occurrence consists of an odd number of character positions, the compiler inserts one byte of FILLER data after each group occurrence. If the item is doubleword aligned and the size of the containing group occurrence is not a multiple of four, the compiler inserts the appropriate amount of FILLER data (1, 2, or 3 bytes) after each group occurrence. In all cases, inserted bytes are not part of the containing occurrences themselves, but are included in group items that contain the complete table. The preceding sequence is repeated for each higher-level table.

The following example illustrates alignment as it applies to multiple OCCURS clauses:

```
01  master.
    02  table-1 OCCURS 5 TIMES.
        03  table-2 OCCURS 5 TIMES.
            04  table-3  OCCURS 5 TIMES.
                05  item-a       PIC 999 COMPUTATIONAL.
                05  item-b    PIC X.
            04  item-3        PIC X.
        03  item-2            PIC X.
```

Although master appears to occupy this many bytes:

```
(((2+1) * 5+1 ) * 5+1) * 5 = 405 bytes
```

it actually occupies:

```
((2+1+1) * 5+1+1) * 5+1+1) * 5 = 560 bytes
```

due to the alignment requirement for the COMPUTATIONAL item.

Implicit FILLER bytes must be accounted for in several situations. These bytes are counted when determining the size of group items that contain them. Thus, when a data item contains implicit FILLER bytes, the character positions of the bytes are included in the allocation requirements of the item. Also, implicit FILLER bytes must be included among the character positions redefined if a containing group item appears as the object of a REDEFINES clause.

Automatic alignment or requested alignment of data items described by redefinition of character positions (through use of the REDEFINES clause) follows the rules described in the preceding paragraphs. However, when the first data item allocated by a redefinition requires word alignment or doubleword alignment, the data item being redefined must begin on the appropriate boundary. In other words, SCREEN COBOL does not permit redefinitions that require insertion of implicit FILLER bytes before the first data item of the redefinition. Any bytes inserted at other places within the redefinition are counted when determining the redefinition size.

USAGE Clause

The USAGE clause defines how a data item is stored within the Compaq *NonStop™ Himalaya* system and, normally, affects the number of character positions used. The USAGE clause does not restrict how the item is used; however, some statements in the Procedure Division require certain usages for their operands.

```
[ USAGE [ IS ] ] { COMP          }
                 { COMPUTATIONAL }
                 { DISPLAY       }
```

`COMP or COMPUTATIONAL`

indicates a numeric data item that is suitable for computations.

`DISPLAY`

indicates a data item value that is stored in the standard data format as a sequence of ASCII characters. If this clause is omitted, the default is DISPLAY.

A USAGE clause can be written at any level. A USAGE clause written at the group level applies to each elementary item in the group. The usage of an elementary item cannot contradict the USAGE clause of a group to which the item belongs. Note, however, that a group item is always considered to be alphanumeric by SCREEN COBOL; thus, the USAGE clause of a group item might not always apply to the manipulation of the item.

A COMPUTATIONAL item has a value suitable for computations and, therefore, must be numeric. The PICTURE string of the item can have only the symbols 9, S, V, and P. Two to eight bytes are selected for a COMPUTATIONAL item, depending on the number of 9 symbols in the PICTURE string, as Table 5-2 indicates.

Table 5-2. Storage Occupied by COMPUTATIONAL Data Items

Number of 9 Symbols	Size of Data Item
1 through 4	2 bytes
5 through 9	4 bytes
10 through 18	8 bytes

Declaration of a group item as COMPUTATIONAL implies that all subordinate items in the group are COMPUTATIONAL. The group item itself cannot be used in computations.

A DISPLAY item has a value that is stored in the standard data format as a sequence of ASCII characters. The characteristics of the item are given in the PICTURE string.

If the PICTURE string of a numeric item contains an S symbol, the item has an operational sign. If a SIGN IS SEPARATE clause is not specified, the operational sign is maintained as part of either the leading or trailing digit; the affected character position will contain a nondigit ASCII character.

VALUE Clause

A VALUE clause specifies the initial value of a Working-Storage item or the value of a level 88 condition name.

Format 1 (Data Initialization) is:

```
VALUE [ IS ] literal
```

literal

is the initial value to be assigned to a data item. The value can be a figurative constant.

Format 2 (Condition Name Entries) is:

```
88 condition-name , { VALUE [ IS ]   }
                    { VALUES [ ARE ] }

   { value-1 [ { THROUGH } value-2 ] } , ...
   {         [ { THRU    }         ] }
```

condition-name

is the name of the condition value.

value-1

is either a single literal value or the first of a range of literal values tested by the condition.

value-2

is the final literal value in a range of literal values tested by the condition. The value must be greater than *value-1*.

VALUE Clause for Data Initialization

Format 1 of the VALUE clause is used to assign an initial value to a Working-Storage item at the time the program is entered. The VALUE clause must not conflict with other clauses in the data description of an item or in the data descriptions of other items within the hierarchy. The following rules apply:

- If the VALUE clause is omitted, the compiler initializes to zero numeric elementary data items in the Working-Storage Section that do not use an OCCURS clause.
- If an item is numeric, all literals of the VALUE clause must be numeric and must be in the range of values set by the PICTURE string. Truncation of nonzero digits is not allowed. A signed numeric literal applies only to a signed numeric PICTURE string. Initialization follows standard alignment rules.
- If an item is nonnumeric, all literals of the VALUE clause must be nonnumeric and must not exceed the size of the PICTURE string. JUSTIFIED clauses are ignored.

- The VALUE clause is not permitted in a data description entry that meets the following criteria:
  - The entry contains an OCCURS or REDEFINES clause.
  - The entry is subordinate to an entry containing an OCCURS or REDEFINES clause.
  - The entry has a variable size due to an OCCURS clause in a subordinate entry.
- If the VALUE clause is used for initialization at the group level, the literal must be a figurative constant or a nonnumeric literal. The group area is initialized without consideration for the individual elementary or other group items within this group. Thus, the group should not have items with descriptions that include JUSTIFIED or USAGE IS COMPUTATIONAL clauses. A VALUE clause cannot appear at the subordinate levels within this group.

The following example illustrates the VALUE clause used for data initialization:

```
WORKING-STORAGE SECTION.
01  main-heading.
    05  FILLER         PIC XX     VALUE SPACES.
    05  FILLER         PIC X(8)   VALUE "DIVISION".
    05  FILLER         PIC XX     VALUE SPACES.
    05  FILLER         PIC X(6)   VALUE "REGION".
        :
01  counters.
    05  no-of-reads    PIC 9(5)   VALUE ZEROS.
    05  no-of-writes   PIC 9(5)   VALUE ZEROS.
```

VALUE Clause for Condition-Name Entries

Format 2 of the VALUE clause is used with condition-name entries. A data item assigned in the Data Division using a level 88 data item is a condition-name; the item under which the 88 appears is the condition variable. A value or a range of values can be defined within this variable for testing. Each entry under a condition variable includes a condition-name with a VALUE clause specifying a value or a range of values for that condition-name.

All condition-name entries for a particular condition variable must immediately follow the entry describing that variable. A condition-name can be associated with any data description entry, even if specified as FILLER, with the following exceptions:

- A condition-name cannot be associated with a level 66 or 77 item.
- A condition-name cannot be associated with a group item with a JUSTIFIED or USAGE IS COMPUTATIONAL clause.

A single value, several values, or a range of values can be given for a condition-name entry.

The following example illustrates single values for condition-names:

```
05  return-code     PIC 99.
    88  end-of-file          VALUE 01.  }
    88  error-on-read        VALUE 02.  }
    88  permanent-error      VALUE 03.  }
    88  error-on-write       VALUE 04.  }
```

where return-code is is the condition variable and end-of-file, error-on-read, permanent-error, and error-on-write are condition-names.

A statement using one of these condition-names might look like this:

```
IF end-of-file,
    PERFORM end-up-routine.
```

The following example illustrates a range of values for a condition-name:

```
05  tax-code          PIC 99.
    88  tax-range     VALUES ARE 00, 03, 07 THROUGH 11.
```

A statement testing whether tax-code has the value 00, 03, 07, 08, 09, 10, or 11 might look like this:

```
IF NOT tax-range
   PERFORM tax-error-routine.
```

Screen Description Entry

A screen description entry declares the characteristics of a screen format. The entry is used in the Screen Section of the SCREEN COBOL program.

A screen can be composed of any combination of literal fields, input fields, output fields, input-output fields, and overlay areas. Each of these items can be combined into logically related groups. A group declaration simplifies referring to related fields, but a group declaration is not required.

The two types of screens are: base and overlay.

- A base screen can be displayed independently. This type of screen can contain areas upon which overlay screens can be displayed.
- An overlay screen is displayed in an overlay area of a base screen. This allows a base screen (with, for example, a constant header section) to be used with various overlay screens.

The structure of the screen description entry is similar to a data description entry. The screen description entry is a series of declarative sentences, each beginning with a level number to indicate the hierarchy. A higher number indicates that the entry is subordinate to the previous entry. The 01 level is the highest statement in the paragraph. Subordinate entry levels can be any number from 02 through 49.

The syntax for screen description entries is:

```
SCREEN SECTION.

  01 base-screen-name [ BASE ] [ SIZE clause ]

     [ input-control-character clauses ]

     [ field-characteristic clauses ] ... .

     { screen group } ...
     { screen field }

     [ screen overlay area ]

[ 01 overlay-screen-name  OVERLAY  SIZE clause ]
[                                              ]
[    [ field-characteristic clauses ] ... .    ]
[                                              ]
[    { screen group } ...                      ]
[    { screen field }                          ]
```

Level 01 introduces a screen description entry. This level defines the name of the screen (a name by which the screen is known throughout the program), defines the size of the screen, and indicates whether the screen is a base or overlay screen. The intermediate levels define groups of items. The highest numbered levels define the characteristics of the screen fields.

The screen description can have the following parts: screen name, screen overlay area, screen group, and screen field. Each of these parts defines a specific attribute of the screen. The following example illustrates a screen description entry:

```
SCREEN SECTION.
01  ENTER-AMT  BASE  SIZE 12, 80.
    05 FILLER         AT 1,  12  VALUE "ORDER DETAIL ENTRY".
    05 FILLER         AT 2,  1   VALUE "CUSTOMER".
    05 FILLER         AT 4,  1   VALUE "ITEM".
    05 FILLER         AT 4,  10  VALUE "QUANTITY".
    05 LINE1-HEADER   AT 5,  1   VALUE "MENU LIST".
    05 OVER1 AREA     AT 6,  1   SIZE 10,80.
01  OVER1-SCREEN OVERLAY SIZE 10,80.
    05 LINE1-OVERLAY  AT 2,  10  VALUE
                                  "1 DISPLAY PREVIOUS ORDER".
```

The input-control character clauses are available for terminals in conversational mode to define the specific input-control characters to be used during execution of an ACCEPT statement. These clauses are described later in this section.

The field-characteristic clauses are available to define the characteristics of screen fields. These clauses are also described later in this section.

Base Screen

A base screen is a screen that is initially displayed on the terminal and is used to establish the current screen for each program unit. In contrast to an overlay screen that is displayed in the overlay area of a base screen, the base screen can be displayed independently.

```
01 screen-name [ BASE ] [ SIZE lines , cols ]
 [ field-characteristic-clause ] ...
```

screen-name

is the name given to the base screen.

SIZE *lines, cols*

indicates the size of the screen. The number of lines and columns can each range from 1 through 255. The size can be no larger than the physical limits of the terminal screen for base screens.

If this option is omitted, the default is 24 lines, 80 columns.

field-characteristic-clause

is one or more clauses that define default characteristics for all fields subordinate to the screen unless these characteristics are explicitly overridden for a particular group or field. The clauses that can appear here are:

```
FILL                    WHEN ABSENT
mnemonic-name           WHEN BLANK
UPSHIFT                 WHEN FULL
USER CONVERSION
```

Screen Overlay Area

A screen overlay area defines an area of a base screen within which an overlay screen can be displayed. When overlay screens are used in a program, a screen overlay area must be defined in the base screen description entry.

```
level-num area-name AREA AT line, col SIZE lines, cols
```

level-num

is a numeric literal that indicates the hierarchy. The value must be within the range of 2 through 49. Subordinate entries are not allowed.

area-name

is the name given to the screen overlay area.

AREA AT *line, col*

specifies the position of the upper left-hand corner of the area relative to the boundaries of the screen.

SIZE *lines, cols*

determines the number of lines and columns included in the area. The entire area must lie within the boundaries of the base screen, and no fields can overlap the area.

For T16-6510 terminals, the *cols* value must be the same as the number of columns declared for the base screen.

Overlay Screen

An overlay screen is a screen that is displayed in an overlay area of a base screen.

```
01 screen-name  OVERLAY  SIZE lines, cols

 [ field-characteristic-clause ] ...
```

screen-name

is the name given to the overlay screen.

SIZE *lines, cols*

indicates the size of the overlay screen. The size can be no larger than the size of the overlay area into which it is to be placed. For 6510 terminals, the width must be exactly the same as the base screen.

field-characteristic-clause

is a clause that defines default characteristics for all fields subordinate to the screen unless explicitly overridden for a particular group or field. The clauses that can appear here are:

```
FILL
mnemonic-name
UPSHIFT
USER CONVERSION
WHEN ABSENT
WHEN BLANK
WHEN FULL
```

Screen Group

A screen group is a combination of fields that are grouped together to provide collective references to the subordinate fields and to define the common characteristics of the fields. A screen group can contain subordinate groups.

```
level-num { group-name } [ [ AT ] line, column ]
          { FILLER     }

  [ field-characteristic clause ] ...

  { screen-field } ...
  { screen-group }
```

level-num

is a numeric literal that indicates the hierarchy. The value must be within the range of 2 through 48.

group-name

is the name given to the group.

FILLER

is a keyword that takes the place of *group-name*.

AT *line, column*

specifies the home position of the group relative to the boundaries of the screen. The line number and column number must be within the size specified for the screen. The positions of subordinate fields can be given relative to the home position; this allows you to move groups easily.

If this clause is omitted, group relative addressing is not allowed in the group.

field-characteristic clause

is one or more clauses that define default characteristics for all fields subordinate to the group unless these characteristics are explicitly overridden for a particular field. The clauses that can appear here are:

FILL
mnemonic-name
UPSHIFT
USER CONVERSION
WHEN ABSENT
WHEN BLANK
WHEN FULL

Screen Field

A screen field is a single elementary item.

```
level-num { field-name }
          { FILLER     }

  [ field-characteristic-clause ] ... .
```

level-num

is a numeric literal within the range of 2 through 49 that indicates the hierarchy.

field-name

is the name given to the field.

FILLER

is a keyword that takes the place of *field-name*. FILLER must be used for a literal field.

field-characteristic-clause

is one or more clauses that define a characteristic of the field. The clauses that can appear here depend on the field type.

Table 5-3 lists the four types of screen fields that are determined by the data association clauses TO, FROM, and USING. It also lists the clauses that can be used with the four types of screen fields.

Note. T16-6520, T16-6530, and T16-6540 considerations: If two successive literals have the same attributes, no separation is necessary; otherwise, at least one space must separate them.

Table 5-3. Screen Field Types and Allowable Field-Characteristic Clauses

Screen Field Type	Determined By	Required Clauses	Optional Clauses
Literal	No TO, FROM, or USING clause	AT clause VALUE clause	Mnemonic-name clause
Input	TO clause only	AT clause or REDEFINES clause and PICTURE clause	FILL clause LENGTH clause Mnemonic-name clause MUST BE clause OCCURS clause RECEIVE clause SHADOWED clause UPSHIFT clause USER CONVERSION clause VALUE clause WHEN ABSENT clause WHEN BLANK clause WHEN FULL clause
Output	FROM clause only	AT clause or REDEFINES clause and PICTURE clause	ADVISORY clause FILL clause Mnemonic-name clause OCCURS clause SHADOWED clause UPSHIFT clause USER CONVERSION clause VALUE clause
Input-Output	USING clause or TO and FROM clauses	AT clause or REDEFINES clause and PICTURE clause	ADVISORY clause FILL clause LENGTH clause Mnemonic-name clause MUST BE clause OCCURS clause RECEIVE clause SHADOWED clause UPSHIFT clause USER CONVERSION clause VALUE clause WHEN ABSENT clause WHEN BLANK clause WHEN FULL clause

Input-Control Character Clauses

Input-Control character clauses are for terminals operating in conversational mode. These clauses define the characters used during the execution of an ACCEPT statement to perform the following:

- Delimit a screen field or a group of screen fields described with an OCCURS clause
- Terminate or abort the processing of an ACCEPT statement
- Restart the processing of an ACCEPT statement

These clauses, which are used only by terminals that are running in conversational mode, have the following syntax:

```
01 screen-name { [ BASE ] [ SIZE clause ] }
               { OVERLAY  SIZE clause     }

   [ ABORT-INPUT [ IS ] { "nonnumeric-literal"     } ]
   [                    { numeric-literal          } ]
   [                    {   [, numeric-literal ]   } ]
   [                    { OFF                      } ]

   [ END-OF-INPUT [ IS ] { "nonnumeric-literal"    } ]
   [                     { numeric-literal         } ]
   [                     {   [, numeric-literal ]  } ]
   [                     { OFF                     } ]

   [ FIELD-SEPARATOR [ IS ] { "nonnumeric-literal" } ]
   [                        { numeric-literal      } ]
   [                        { OFF                  } ]

   [ GROUP-SEPARATOR [ IS ] { "nonnumeric-literal" } ]
   [                        { numeric-literal      } ]
   [                        { OFF                  } ]

   [ RESTART-INPUT [ IS ] { "nonnumeric-literal"   } ]
   [                      { numeric-literal        } ]
   [                      {   [, numeric-literal ] } ]
   [                      { OFF                    } ]
```

Note. Programs using double-byte data must use only single-byte (ASCII) characters to define input-control clauses.

The following example illustrates the input-control character clauses:

```
SCREEN SECTION.
01  CUSTOMER-REC-SCREEN  BASE  SIZE 24, 80
                               FIELD-SEPARATOR ","      (1)
                               GROUP-SEPARATOR OFF
                               ABORT-INPUT     "AI"     (2)
```

```
                                  END-OF-INPUT    64, 64    (3)
                                  RESTART-INPUT   "2" .      (4)
```

(1) Documents the default field-separator character.

(2) Defines the keyboard abort-input characters as AI.

(3) Defines the keyboard end-of-input characters as @@ (ASCII code 64 represents @).

(4) Defines the keyboard restart-input character as 2.

The input-control character clauses are described in alphabetic order in the following paragraphs.

ABORT-INPUT Clause

The ABORT-INPUT clause defines the characters used to terminate the processing of the current ACCEPT statement with an abort termination status. The ABORT-INPUT clause is recognized only by terminals operating in conversational mode.

```
ABORT-INPUT [ IS ] { "nonnumeric-literal"                    }
                   { numeric-literal [,numeric-literal ] }
                   { OFF                                     }
```

"nonnumeric-literal"

is one or two alphanumeric characters enclosed in quotation marks.

numeric-literal

is one or two integers. Each integer must be within the range of 0 through 255. *numeric-literal* is the decimal value of an 8-bit binary number.

If a process is responding in place of a terminal, SCREEN COBOL interprets the 8-bit pattern (two numeric literals convert to a 16-bit pattern) as a nonkeyboard character.

OFF

specifies that ABORT-INPUT is not available for the current screen.

If this clause is omitted, the abort-input characters are @@.

If used, the ABORT-INPUT clause must be specified at the 01 screen level. A character defined for ABORT-INPUT cannot be specified for another input-control character.

If the abort-input character is entered during an ACCEPT statement, no values in the Working-Storage Section are changed by that ACCEPT statement.

END-OF-INPUT Clause

The END-OF-INPUT clause defines the characters used to indicate the end of the last input field for the current ACCEPT statement. The END-OF-INPUT clause is recognized only by terminals operating in conversational mode.

```
END-OF-INPUT [ IS ] { "nonnumeric-literal"                    }
                    { numeric-literal [, numeric-literal ] }
                    { OFF                                     }
```

"*nonnumeric-literal*"

is one or two alphanumeric characters enclosed in quotation marks.

numeric-literal

is one or two integers. Each integer must be within the range of 0 through 255. *numeric-literal* is the decimal value of an 8-bit binary number.

If a process is responding in place of a terminal, SCREEN COBOL interprets the 8-bit pattern (two numeric literals convert to a 16-bit pattern) as a nonkeyboard character.

OFF

specifies END-OF-INPUT is not available for the current screen.

If this clause is omitted, the end-of-input characters are //.

If used, the END-OF-INPUT clause must be specified at the 01 screen level. A character defined for END-OF-INPUT cannot be specified for another input-control character.

FIELD-SEPARATOR Clause

The FIELD-SEPARATOR clause defines the character used to separate one screen field from another during an ACCEPT statement. If a screen field description includes an OCCURS clause, each occurrence is treated as one field. The FIELD-SEPARATOR clause is recognized only by terminals operating in conversational mode.

```
FIELD-SEPARATOR [ IS ] { "nonnumeric-literal" }
                       { numeric-literal      }
                       { OFF                  }
```

"*nonnumeric-literal*"

is one alphanumeric character enclosed in quotation marks.

numeric-literal

is one integer that must be within the range of 0 through 255. *numeric-literal* is the decimal value of an 8-bit binary number.

If a process is responding in place of a terminal, SCREEN COBOL interprets the 8-bit pattern as a nonkeyboard character.

`OFF`

specifies that FIELD-SEPARATOR is not available for the current screen.

If this clause is omitted, the field-separator character is a comma (,).

If used, the FIELD-SEPARATOR clause must be specified at the 01 screen level. The character defined for FIELD-SEPARATOR cannot be specified for another input-control character. In the following example, the FIELD-SEPARATOR clause defines S as the keyboard character to be used.

```
SCREEN SECTION.
01  EMP-RECORD-SCREEN   BASE  SIZE 24, 80
                                FIELD-SEPARATOR IS "S" .
```

GROUP-SEPARATOR Clause

The GROUP-SEPARATOR clause defines the character used during the processing of an ACCEPT statement to indicate one of the following:

- Last item in an OCCURS clause
- End of a field, if the field preceding the group separator has no multiple occurrences

The GROUP-SEPARATOR clause is recognized only by terminals operating in conversational mode.

```
GROUP-SEPARATOR [ IS ] { "nonnumeric-literal" }
                       { numeric-literal      }
                       { OFF                  }
```

"*nonnumeric-literal*"

is one alphanumeric character enclosed in quotation marks.

numeric-literal

is one integer that must be within the range of of 0 through 255. *numeric-literal* is the decimal value of an 8-bit binary number.

If a process is responding in place of a terminal, SCREEN COBOL interprets the 8-bit pattern as a nonkeyboard character.

`OFF`

specifies that GROUP-SEPARATOR is not available for the current screen.

If this clause is omitted, the group-separator character is a semicolon (;).

If used, the GROUP-SEPARATOR clause must be specified at the 01 screen level. The character defined for GROUP-SEPARATOR cannot be specified for another input-control character.

RESTART-INPUT Clause

The RESTART-INPUT clause defines the characters used to restart input processing during the current ACCEPT statement. The RESTART-INPUT clause is recognized only by terminals operating in conversational mode.

```
RESTART-INPUT [ IS ] { "nonnumeric-literal"                  }
                     { numeric-literal [,numeric-literal] }
                     { OFF                                   }
```

"*nonnumeric-literal*"

is one or two alphanumeric characters enclosed in quotation marks.

numeric-literal

is one or two integers. Each integer must be within the range of 0 through 255. *numeric-literal* is the decimal value of an 8-bit binary number.

If a process is responding in place of a terminal, SCREEN COBOL interprets the 8-bit pattern (two numeric literals convert to a 16-bit pattern) as a nonkeyboard character.

OFF

specifies that RESTART-INPUT is not available for the current screen.

If this clause is omitted, the restart-input characters are two exclamation points (!!).

If used, the RESTART-INPUT clause must be specified at the 01 screen level. A character defined for RESTART-INPUT cannot be specified for another input-control character. If the current ACCEPT statement is restarted, the data entered before the restart-input characters does not change the values of the associated data items in Working-Storage. If data is entered on the same line following the restart-input characters, the data is ignored.

Field-Characteristic Clauses

Field-characteristic clauses specify various characteristics of screen fields. These field-characteristic clauses have the following syntax and are described in alphabetic order in the following paragraphs.

```
level-num { field-name } { [ AT ] line-spec, column-spec }
          { FILLER     } { REDEFINES field-name-2        }

 [ ADVISORY ]

 [ CONTROLLED [ BY ] data-name-1 ]

 [ CONVERT BLANKS ]

 [ FILL nonnumeric-literal ]

 [ LENGTH [ MUST BE ]                              ]
 [                                                 ]
 [   { literal-1 [ { THROUGH } literal-2 ] } ... ]
 [   {           [ { THRU    }           ] }     ]

 [ mnemonic-name ] ...

 [ MUST [BE] { literal-1 [ { THROUGH } literal-2 ] } ... ]
 [           {           [ { THRU    }           ] }     ]

 [ OCCURS { lines-phrase [ columns-phrase ] } ]
 [        { columns-phrase [ lines-phrase ] } ]
 [                                           ]
 [   [ DEPENDING [ ON ] data-name-1 ]        ]

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

 [ PROMPT screen-field ]

 [ RECEIVE [ FROM ] { ALTERNATE             } ]
 [                  { ALTERNATE OR TERMINAL } ]
 [                  { TERMINAL              } ]
 [                  { TERMINAL OR ALTERNATE } ]

 [ SHADOWED [ BY ] data-name-1 ]

 [ { TO    } data-name-1 ]
 [ { FROM  }             ]
 [ { USING }             ]

 [ UPSHIFT [ INPUT        ] ]
 [         [ OUTPUT       ] ]
 [         [ I-O          ] ]
 [         [ INPUT-OUTPUT ] ]

 [ USER [ CONVERSION ] numeric-literal ]

 [ VALUE nonnumeric-literal ]
```

(continued)

```
[ WHEN { ABSENT } { CLEAR } ]
[      { BLANK  } { SKIP  } ]

[ [ WHEN ] FULL { TAB  } ]
[               { LOCK } ]
```

ADVISORY Clause

The ADVISORY clause identifies a single output or input-output field as the field to be used for informational and error messages generated by the TCP.

```
ADVISORY
```

Every base screen should have an advisory field. The field should be alphanumeric with a size of at least 35 characters. Error messages that appear in this field are described in Appendix A.

An overlay screen must not have an advisory field.

The ADVISORY clause cannot be associated with a field that allows only double-byte data.

For terminals in conversational mode, an advisory field must be defined for the screen, or the standard advisory messages are not displayed on the terminal.

AT Clause

The AT clause specifies the location of the field.

```
AT line-spec, column-spec
```

line-spec

specifies the line in which the field begins.

column-spec

specifies the column in which the field begins.

Both *line-spec* and *column-spec* can appear in the following forms:

numeric-literal

This form represents the line or column relative to the beginning of the screen.

```
* [ { + | - } numeric-literal ]
```

This form represents a location relative to the current position. The current position begins at line 1, column 1 and is advanced to the first available position following a field after that field is declared.

```
@ [ { + | - } numeric-literal ]
```

This form represents a location relative to the home position of the group containing the field declaration. The home position is the first data character of the field and is specified for the group with the AT clause.

Either the AT clause or the REDEFINES clause must be included in every screen field declaration. If both clauses appear in the screen field declaration, they must refer to exactly the same position.

CONTROLLED Clause

The CONTROLLED clause specifies a Working-Storage Section or Linkage Section data item representing a structure capable of supporting run-time control of a screen field's display attributes.

When you use an appropriate combination of the CONTROLLED clause, the DYNAMIC modifier, and the shadowed clause, you can combine the operations DISPLAY BASE and DISPLAY OVERLAY, or TURN and DISPLAY, into one operation.

```
CONTROLLED [ BY ]  data-name-1
```

data-name-1

is the name of a group data item structure that can support all of the currently defined Pathway screen field attributes.

Specifying Data Items in a Control Structure

A control structure for a particular attribute definition must comply with the layout and data types; otherwise, a compile-time error occurs.

An attribute control element (attrib-TOKEN) is a token consisting of an attribute identifier (attrib-ID) and the attribute's value (attrib-VALUE). The number of attrib-TOKEN items in the structure, up to the maximum defined by MAX-TOKEN-PAIRS (which has a maximum of 15), is controlled through the structure's TOKEN-COUNT data item. To support simultaneous attribute changes of a screen field (for example, in TURN operation), you must define the appropriate number of attrib-TOKEN items. The values are as specified in the following table.

A Data Item Controlling...	Must Have an attrib-ID of...	With These Values...
Field Attributes	1 through 14, as follows: 1 = Bright 2 = Hidden 3 = Mdton 4 = Numeric-shift 5 = Protected 6 = Blink 7 = Reverse 8 = Underline 9 = Bell 10 = Topline 11 = Leftline 12 = Rightline 13 = Bottomline 14 = Boxfield	-1 = Field control inactive, IGNORE 0 = Field control active, attribute OFF 1 = Field control active, attribute ON
Color Attributes (COLOR-ID)	15 = Color	0 through 7, as follows: 0 = Color default 1 = Blue 2 = Red 3 = Pink 4 = Green 5 = Turquoise 6 = Yellow 7 = Neutral
Field Attribute	16 = SOSI-DISABLED	0 = The user can input both one-byte and two-byte characters 1 = The user can input only two-byte characters

Note. SOSI creation is not allowed to PIC A or PIC 9 fields when the CHARACTER-SET is KANJI-KATAKANA.

Using a Control Structure

A screen field specifying a CONTROLLED clause associates itself with a Working-Storage or Linkage Section data item (*data-name-1*) that supports run-time control and definition of the screen field's display attributes. The associated control structure becomes operative only when the DISPLAY BASE DYNAMIC, DISPLAY OVERLAY DYNAMIC, and TURN DYNAMIC OPERATIONS clauses are invoked. Table 5-4 shows the effect of the CONTROLLED clause on the formation of screen field attributes during these operations.

Table 5-4. Effect of CONTROLLED Clause on Screen Field Display Attribute

	TURN, DISPLAY BASE, and DISPLAY OVERLAY Operations Change Screen Field Display Attributes at...	
Screen Field Control	**Without DYNAMIC**	**With DYNAMIC**
Field not controlled	Compile time	Compile time
Field controlled: inactive, value = -1	Compile time	Compile time
Field controlled: active, value = 0 (OFF)	Compile time	Run time
Field controlled: active, value = 1 (ON)	Compile time	Run time

Note. The error 3072 (RUN-TIME DYNAMIC ATTRIBUTE SETTING INVALID) is returned if an incorrect structure for *data-name-1* is encountered at run-time. The compiler only detects syntax errors, undeclared identifiers, and whether *data-name-1* is in Working-Storage. This error occurs only for IBM 3270 terminals when a field is defined as a combination of two or more of the BLINK, REVERSE, and UNDERLINE attributes.

Note. For 65XX terminals, a field which is defined as BRIGHT and BLINK will only blink, and the field which is defined as REVERSE and BRIGHT will only be reversed. Further, a field which is defined as HIDDEN and BRIGHT will be hidden, and the field defined as HIDDEN and BLINK will also be hidden.

As the control structure is a Working-Storage data item, there is no restriction on sharing the control structure of one screen field with that of a different screen field

Effect of OCCURS Clause

If the screen field defined with the CONTROLLED clause has an OCCURS clause, the *data-name-1* structure must have the same maximum number of occurences as that defined in the Working-Storage Section.

The following control structure provides the minimum information set required for a Working-StorageWorking-Storage group data item (*data-name-1*).

```
01 DATA-NAME-1.
   05 TOKEN-COUNT              PIC S99    COMP.
   05 attrib-TOKEN.
      10 attrib-ID             PIC S99    COMP.
      10 attrib-VALUE          PIC S99    COMP.

77 BRIGHT-ID                   PIC S99, VALUE  1.
77 HIDDEN-ID                   PIC S99, VALUE  2.
77 MDTON-ID                    PIC S99, VALUE  3.
77 NUMERIC-SHIFT-ID            PIC S99, VALUE  4.
77 PROTECTED-ID                PIC S99, VALUE  5.
```

```
77 BLINK-ID                    PIC S99, VALUE  6.
77 REVERSE-ID                  PIC S99, VALUE  7.
77 UNDERLINE-ID                PIC S99, VALUE  8.
77 BELL-ID                     PIC S99, VALUE  9.
77 TOPLINE-ID                  PIC S99, VALUE  10.
77 LEFTLINE-ID                 PIC S99, VALUE  11.
77 RIGHTLINE-ID                PIC S99, VALUE  12.
77 BOTTOMLINE-ID               PIC S99, VALUE  13.
77 BOXFIELD-ID                 PIC S99, VALUE  14.
77 COLOR-ID                    PIC S99, VALUE  15.

77 MAX-TOKEN-PAIRS             PIC S99, VALUE  15.

77 ATTRIB-IGNORE               PIC S99, VALUE  -1.
77 ATTRIB-OFF                  PIC S99, VALUE  0.
77 ATTRIB-ON                   PIC S99, VALUE  1.
77 COLOR-DEFAULT               PIC S99, VALUE  0.
77 COLOR-BLUE                  PIC S99, VALUE  1.
77 COLOR-RED                   PIC S99, VALUE  2.
77 COLOR-PINK                  PIC S99, VALUE  3.
77 COLOR-GREEN                 PIC S99, VALUE  4.
77 COLOR-TURQUOISE             PIC S99, VALUE  5.
77 COLOR-YELLOW                PIC S99, VALUE  6.
77 COLOR-NEUTRAL               PIC S99, VALUE  7.
```

The following example structure supports control for a screen field's BLINK and REVERSE attribute settings. This structure causes an associated screen field to blink in reverse video.

```
01 CONTROL-ITEM-1.
   05 TOKEN-COUNT               PIC S99    COMP, VALUE 2.
   05 ATTRIB-TOKEN-BLINK.
      10 ATTRIB-ID              PIC S99    COMP, VALUE 6.
      10 ATTRIB-VALUE           PIC S99    COMP, VALUE 1.
   05 ATTRIB-TOKEN-REVERSE.
      10 ATTRIB-ID              PIC S99    COMP, VALUE 7.
      10 ATTRIB-VALUE           PIC S99    COMP, VALUE 1.
```

The following example structure supports control for a screen field's COLOR and REVERSE attribute settings. This structure causes an associated screen field to change to red in reverse video.

```
01 CONTROL-ITEM-2.
   05 TOKEN-COUNT               PIC S99    COMP, VALUE 2.
   05 ATTRIB-TOKEN-COLOR.
      10 ATTRIB-ID              PIC S99    COMP, VALUE 15.
      10 ATTRIB-VALUE           PIC S99    COMP, VALUE 2.
   05 ATTRIB-TOKEN-REVERSE.
      10 ATTRIB-ID              PIC S99    COMP, VALUE 7.
      10 ATTRIB-VALUE           PIC S99    COMP, VALUE 1.
```

CONVERT BLANKS Clause

The CONVERT BLANKS clause allows the USER CONVERSION clause to be invoked when the terminal operator enters blanks or fill characters.

```
CONVERT BLANKS
```

Note. When a user presses the Tab key to bypass a field, the MDT (modified data tag) does not get set (in other words, it is off). In such situations, use the CONVERT BLANKS clause in conjunction with the WHEN ABSENT CLEAR clause to force blanks in a field. This allows the USER CONVERSION clause to be invoked.

In the following example, the conversion procedure for FIELD1 is invoked when the terminal operator enters data other than blanks or fill characters. The conversion procedure for FIELD2 is invoked when the operator enters data, blanks, or fill characters. The conversion procedure for FIELD3 is invoked when the terminal operator enters data, blanks, or fill characters, or when the field is skipped.

```
SCREEN SECTION

01 MENU1
   05  FIELD1           PIC X(20)
                        AT 4, 45
                        TO WS-1
                        USER CONVERSION 1.

   05  FIELD2           PIC X(20)
                        AT 5, 45
                        TO WS-2
                        CONVERT BLANKS
                        USER CONVERSION 1.

   05  FIELD3           PIC X(20)
                        AT 6, 45
                        TO WS-3
                        WHEN ABSENT CLEAR
                        CONVERT BLANKS
                        USER CONVERSION 1.
```

FILL Clause

The FILL clause declares a padding character for the field. When output to the field does not fill the full width specified, the padding character fills in to the right of the field.

```
FILL nonnumeric-literal
```

nonnumeric-literal

is one character long. If a FILL clause is used with a field that allows only double-byte data (PIC N), the fill character must be a double-byte character. If a FILL

clause is used with a PIC X or a PIC A field, the fill character must be a valid single-byte (ASCII) character.

If this clause is omitted, the default fill character is a space, except on 3270 terminals, where the default fill character is a null. Also on 3270 terminals, if the data is shorter than the length of the field, the remainder of the field is filled with nulls.

On input, the trailing FILL characters are removed from the input string before the input is analyzed for errors and converted. If a TO clause contains a numeric field, the leading and trailing FILL characters are removed before the input is processed. FILL characters embedded within a field are not removed.

If FILL and OCCURS clauses are both used with a field, on output the FILL clause applies to all occurrences of the field, regardless of the setting of *data-name-1* in a DEPENDING ON clause.

LENGTH Clause

The LENGTH clause specifies the acceptable number of characters that can be entered into a screen input field. The number of characters input is determined before conversion but after the fill characters are removed.

```
LENGTH [ MUST BE ] {literal-1[ { THROUGH }literal-2 ] }, ...
                   {          [ { THRU    }          ] }
```

literal-1 and *literal-2*

are numeric values from 0 through the field size. If *literal-2* is included, its value must be greater than *literal-1*.

The maximum value allowed by the compiler is 255.

If this clause is omitted, any number of characters are allowed within the constraints of the picture.

If the LENGTH clause is used with double-byte fields, the values given to the clause indicate the number of characters of the given type that the operator must enter. For example, the clause:

```
LENGTH MUST BE 6
```

means one of the following:

- Six displayable single-byte (ASCII) characters are required for a PIC X(10) field.
- Six single-byte (ASCII) characters are required for a PIC A(50) field.
- Six double-byte characters (having 12 bytes total) are required for a PIC N(30) field.

If you use a mixed field, the LENGTH MUST BE clause refers to the absolute number of bytes that the operator must enter. For example, PIC A(10)N(5)X(5) with a LENGTH MUST BE 6 clause means that an operator must enter six alphabetic characters. A LENGTH MUST BE 11 clause is not possible here because the operator

would have to enter ten alphabetic characters for the first ten bytes—and half of a double-byte character for the eleventh byte.

The following example specifies that FLD1 is optional (length can be 0), but must be five characters long if it is entered; FLD2 is required, but 1 through 5 characters can be entered.

```
04  FLD1  AT 1, 1  TO X  PIC A9999  LENGTH  0, 5.
04  FLD2  AT 2, 1  TO Y  PIC ZZZZ9  LENGTH  1 THRU 5.
```

When a field is optional and no characters are input, the value of the associated data item is changed by the ACCEPT statement according to the WHEN ABSENT/BLANK field-characteristic clause.

Mnemonic-Name Clause

The mnemonic-name clause allows you to specify display attributes for a screen field. The mnemonic-name is associated with a display attribute in the SPECIAL-NAMES paragraph of the Environment Division.

```
mnemonic-name
```

The display attributes combined with the default values for unspecified attributes determine the display attributes for the field when the field is displayed initially. Display attributes can be restored by a RESET statement, as described in Section 6, Procedure Division.

The default value for the protection attribute depends on the screen field type. If the field is an input or input-output field, the default is UNPROTECTED. If the field is an output field, the default is PROTECTED.

A *mnemonic-name* can be associated with the display attribute system names listed in Table 4-1 and Table 4-2.

MUST BE Clause

The MUST BE clause specifies the acceptable values for an input screen field.

```
MUST [ BE ] { literal-1 [ { THROUGH } literal-2 ] } , ...
            {           [ { THRU    }           ] }
```

literal-1 and *literal-2*

are numeric literals for numeric items and nonnumeric literals for alphanumeric items.

Any figurative constant except ALL can be specified.

The literals used in this clause must match for the screen field and the associated data item, or an error is generated. For example, if a screen field receives alphanumeric character data, that data must go into a data item that is defined with a nonnumeric PICTURE clause. Numeric items are compared numerically; alphanumeric items are compared left to right according to the ASCII character set. For example, an input string

9 is less than 10 if the screen PICTURE clause is numeric. An input string "9" is greater than "10" if the screen PICTURE clause is nonnumeric.

When the MUST BE clause is processed, a numeric literal is scaled to match the PICTURE clause defined for the associated data item. For example, if you have a PICTURE clause of PIC 999.99 in the Screen Section data item and a MUST BE clause of MUST BE 100, the MUST BE clause is scaled to MUST BE 100.00. You do not have to include the decimal places in the MUST BE clause unless you want to specify the value to two decimal places.

The MUST BE clause by itself does not make a screen field required. For a field to be required, three clauses are necessary:

- LENGTH clause—specifies that a length greater than zero is required (This clause must be specified for the TCP to verify the MUST BE clause.)
- MUST BE clause—specifies acceptable values if the field is present
- MDTON (Modified Data Tag On) attribute—specifies data validation even if the field is unchanged

You can use the THROUGH/THRU clause with level 88 data items associated with double-byte character-set literals. Byte-by-byte comparisons of all items in the THROUGH/THRU clause are performed. Double-byte character set data used in the THROUGH/THRU clause is treated as a byte string.

OCCURS Clause

The OCCURS clause specifies multiple occurrences of screen fields. This clause can define a column, a row, or a rectangular array of fields. Each occurrence of the field is identical except for location, and each is associated with a particular occurrence of a Working-Storage data item having an OCCURS clause.

```
OCCURS { lines-phrase [ columns-phrase ] }
       { columns-phrase [ lines-phrase ] }

  [ DEPENDING [ ON ] data-name-1 ]
```

columns-phrase is:

```
   IN literal-1 COLUMNS { OFFSET   } { literal-k } ,...
                        { SKIPPING }
```

lines-phrase is:

```
   ON literal-2 LINES [ SKIPPING literal-3 ]
```

IN...COLUMNS, ON...LINES

determines the number of field occurrences, the location of each field occurrence, and the ordering of the field occurrences.

literal-1

is a numeric literal that specifies the number of field occurrences on a line.

literal-k

is a numeric literal that specifies the horizontal spacing of the field columns.

When OFFSET is specified, *literal-k* is the number of spaces between the first column of a field occurrence (*literal-1*) and the first column of the next field occurrence (*literal-1* +1) on the same line.

When SKIPPING is specified, *literal-k* is the number of spaces between the last column of a field occurrence (column k) and the first column of the next field occurrence (column k+1) on the same line. There can be at most (*literal-1*) -1 separations. If there are fewer separations, the last *literal-k* is used repeatedly. No separation is required after the last literal.

literal-2

is a numeric literal that specifies how many lines contain occurrences.

literal-3

is a numeric literal that specifies how many lines are skipped between lines containing occurrences of the field.

DEPENDING

indicates that the number of occurrences is variable.

data-name-1

is the unsubscripted name of an elementary numeric item where the current number of occurrences is defined. This item must be defined in the Working-Storage Section or Linkage Section. On input (execution of an ACCEPT statement), this item is set. On output (execution of a DISPLAY statement), this item is used to define the number of values output. If FILL and OCCURS clauses are both used with a field, on output the FILL clause applies to all occurrences of the field, regardless of the setting of *data-name-1* in a DEPENDING ON clause.

The following conventions apply to the OCCURS clause:

- When the IN...COLUMNS phrase is omitted, a single occurrence on each line is indicated.
- The order of the phrases determines the order in which the occurrence numbers are assigned to the occurrences.
  - If the ON...LINES phrase is specified first, the occurrences are numbered sequentially from line to line down a column.
  - If the IN...COLUMNS phrase is specified first, the occurrences across a line are numbered sequentially.

- A screen field described with an OCCURS clause and associated with a data item by a TO, FROM, or USING clause, must define the same maximum number of occurrences in the OCCURS clause as is specified in the associated data item OCCURS clause. The following example is a Working-Storage data item associated with the screen field.

```
WORKING-STORAGE SECTION.
01  GAME-SCHE-REC.
    :
    05  TABLE-A       PIC X(8)  OCCURS 4 TIMES.
    :
SCREEN SECTION.
    :
    05  FIELD-A        AT 6, 10  PIC X(8)  USING TABLE-A
                                  OCCURS IN 4 COLUMNS
                                  SKIPPING 1.
```

- If the data item named in the TO, FROM, or USING clause has subordinate items and contains multiple OCCURS clauses, the maximum number of occurrences for each OCCURS clauses must match the maximum number of occurrences specified in the corresponding screen field descriptions.

- A single screen description can have any number of variable length tables. The restriction of one for each structure that applies to the Working-Storage Section and Linkage Section does not apply to screens.

- The OCCURS clause cannot define a screen group; however, the clause can define screen fields in a screen group. The following example shows how the OCCURS clause defines each screen field in a screen group:

```
SCREEN SECTION.
    :
    03  DATA-OUT.
        05  JOB-ENTRY.
            07  JOB-NR      AT 6,16  PIC Z(4)
                             FROM JOB-NR OF SCREEN-TEMP
                             OCCURS ON 10 LINES
                             SKIPPING 1.
            07  JOB-STATE   AT 6,22  PIC X(5)
                             FROM JOB-STATE OF SCREEN-TEMP
                             OCCURS ON 10 LINES
                             SKIPPING 1.
```

- A reference to a screen field that is described with an OCCURS clause should appear without a subscript when the field is used as one of the screen identifiers in an ACCEPT statement. In other statements where screen identifiers can be used, a reference to a screen field described with an OCCURS clause can appear with or without a subscript. A reference without a subscript refers to all occurrences of the table. A reference that includes a subscript refers only to the occurrence selected by the value of the subscript.

- When a screen field described with a DEPENDING phrase is referred to in an ACCEPT statement, part of the ACCEPT statement processing performed by the TCP is the determination of the size of the table—the value to be stored into *data-name-1*. All occurrences of the field are examined and the TCP sets

data-name-1 to the occurrence number of the last occurrence that was entered. If the field is also a required field, all preceding occurrences of the field must also be entered. Failure to do this causes a PREVIOUS FIELD MISSING error message to be displayed for the terminal operator.

- Several tables on the same screen might have the same *data-name-1* in their DEPENDING phrase. If the tables are referred to in the same ACCEPT statement, the value of *data-name-1* is set to the maximum of the values that would be computed when considering each table separately. If this causes the value of *data-name-1* to be set greater than the highest supplied occurrence of a table whose fields are required, the input is in error and a REQUIRED FIELD MISSING or EARLIER FIELD MISSING (depending on the order of the fields) message is displayed for the terminal operator.
- When a reference to a screen field described with a DEPENDING phrase appears without a subscript in any statement other than an ACCEPT statement, the reference is to all occurrences within the current size of the table, as specified by the value in *data-name-1*.
- When the value of *data-name-1* of the DEPENDING phrase decreases and a DISPLAY statement displays the table of screen field values, the fields previously displayed beyond the current limit remain on the screen. To avoid having old screen field values remain on the screen, you can handle the display in one of the following ways:
  - Specify a DISPLAY BASE or DISPLAY RECOVERY statement before the DISPLAY statement for the current table of values.
  - Do not use the DEPENDING phrase; define the table for the maximum length and always display the full table with field values set appropriately.
- The DEPENDING phrase specifies the maximum size of a screen table that your program can refer to during execution. The maximum size of a screen table is the lower of the following values:
  - The literal specified in the OCCURS clause of the associated Working-Storage table
  - The literal specified in the IN ... COLUMNS or ON ... LINES phrase in the OCCURS clause of the screen table definition

  Generally, you should make certain that these values are the same. If the TCP references a screen table item with a subscript value that exceeds the maximum size allowed for the screen table, the TCP suspends the terminal and displays an error message.
- By including the DEPENDING clause in the screen table definition you can allow the screen table to have a varying size based on the value in *data-name-1.* The current size of the screen table specified in *data-name-1* determines the maximum subscript value allowed for that screen table and the maximum number of screen table items that are operated on when no subscript value is specified.
  - If the TCP references a screen table item with a subscript value that exceeds the current size, the TCP suspends the terminal and displays an error message.

- If the current size exceeds the maximum size and the screen table is referenced with a subscript, the TCP suspends the terminal and displays an error message.
- If the current size exceeds the maximum size and the screen table is referenced without a subscript, the TCP executes the statement as if the current size is zero. The TCP does not, however, flag this as an error and the operation is performed on zero items in the screen table.

The following example illustrates the OCCURS clause:

```
05  FLD-A   AT 6, 10  PIC X(8) FROM TBL-A
            OCCURS IN 4 COLUMNS OFFSET 10.
```

An equivalent OCCURS clause would be:

```
OCCURS IN 4 COLUMNS SKIPPING 2.
```

PICTURE Clause

The PICTURE clause defines the format in which the data appears on the terminal screen.

```
{ PIC     } [ IS ] character-string
{ PICTURE }
```

character-string

can take the same form as that described in the data description entry with the following exceptions:

- The symbol S cannot appear in the picture.
- Numeric edited forms and alphanumeric edited forms are allowed.

The maximum size allowed by the compiler is 255 bytes.

Generally, screen field input is edited in a manner that is inverse to normal editing functions implied by the PICTURE clause. The input editing always correctly reconverts a value, using the same PICTURE clause for input and output.

The input editing process is different for the two classes of the input item:

- Alphanumeric input—Only the left-hand portion of the picture corresponding to the actual number of input characters must be matched. The remaining portion of the picture is ignored.
- Numeric input—Leading and trailing spaces and fill characters are first removed from the input data string. If there are only spaces in the input string (this can occur when a terminal operator simply tabs from one field to the next), the string is considered to indicate the value 0. Otherwise an attempt is made to match each character in the picture with a character in the input data, proceeding from right to left. If no match is made, the data is considered to be in error.

Some picture symbols are special in that the positions they represent might be omitted from the input data string. Symbols that can be included in this category are

Z, comma, multiple plus and minus signs, CR, DB, and multiple currency signs. If a mismatch occurs with an input character of this type, and if a space would be acceptable at that point in the input string, the data is not considered in error; the picture symbol is replaced by a space, and the TCP attempts to match the input character with the next picture symbol. Table 5-5 gives the description of these character-string symbols.

Table 5-5. Screen Description Entry PICTURE Character-String Symbols (page 1 of 2)

Symbol	Meaning
A	Represents a character position for a letter of the alphabet or a space character. If the character is not a letter or a space, it is flagged as an error.
B	Represents a character position where a space must occur in the input. The space is deleted during conversion into its associated data item. This character should not be used as the rightmost character of a numeric picture because trailing spaces are removed before conversion.
N	Represents a double-byte character and is valid only in program units that specify the KANJI-KATAKANA keyword in the CHARACTER-SET clause of the OBJECT-COMPUTER paragraph in the Environment Division.
P	Indicates an implicit decimal position (with value zero) to be used in aligning the decimal point in the numeric result. Refer to the description of the V symbol for cautions.
V	Indicates the decimal point location in a numeric item in which the terminal operator will not enter an explicit decimal point. The alignment takes place from the last character entered in the field. Use this symbol with care because the variable length nature of data entered could cause unintended alignments to occur. To avoid alignment problems, you should use the LENGTH clause to require full length entry whenever you use a picture with implicit decimal places and potentially absent positions (for example, positions defined with the Z symbol).
X	Represents a character position that can have any character from the ASCII character set.
Z	Represents a position that must be a digit or must be a space if no digits appear to the left of the symbol. The symbol is replaced by a space during editing only when it is one of a set of multiple Z symbols. A space is equivalent to a zero for purposes of conversion.
9	Represents a character position that must be a digit.
0	Represents a character position where a zero must appear. The zero is deleted during conversion into the associated data item.
/	Represents a character position where a right slant must appear. The / is deleted during conversion into the associated data item.
,	Represents a character position where a comma must appear if any digits appear to the left of it. If no digits appear to the left of the symbol, the character must be a space (or other floating insertion character). The comma is deleted during conversion into the associated data item.

Table 5-5. Screen Description Entry PICTURE Character-String Symbols (page 2 of 2)

Symbol	Meaning
.	Represents a character position where a period must appear and indicates decimal point alignment. The period is deleted during conversion into the associated data item.
+	Represents a position where either a plus or a minus sign must appear. Multiple plus signs represent positions that must contain some number of digits preceded by a single plus sign or a single minus sign, preceded by spaces. The symbol is replaced by a space during editing only when it is one of a set of multiple plus signs.
-	Represents a position where either a space or a minus sign must appear. Multiple minus signs represent positions that must contain some number of digits preceded by an optional minus sign, preceded by spaces. The symbol is replaced by a space during editing only when it is one of a set of multiple minus signs.
CR	Represents positions that must contain the characters CR or spaces. These symbols are replaced by spaces during editing if the value is not negative.
DB	Represents two positions that must contain the characters DB or spaces. These symbols are replaced by spaces during editing if the value is not negative.
*	Represents a position that must be a digit or an asterisk. If the position is a digit, the digit must be to the left of all asterisks.
$	Represents a position where a currency symbol must appear. Multiple currency symbols represent positions that must contain some number of digits preceded by a currency symbol, preceded by spaces. The symbol is replaced by a space during editing only when it is one of a set of multiple currency symbols.

Item Size

The size of a data item is determined by the symbols of its PICTURE string. The character-string symbols DB and CR each count as two character positions. Symbols V and P do not count as character positions. All others count as one character position.

PROMPT Clause

The PROMPT clause associates a named screen item for output with a screen field for input. During the processing of an ACCEPT statement, the contents of a named screen item can be displayed (to assist the terminal operator) before the screen input is read. The PROMPT clause is valid only for terminals operating in conversational mode.

```
PROMPT screen-field
```

screen-field

is the name of a previously defined screen field. The contents of *screen-field* can be described in the Screen Section with a VALUE clause or in a Working-Storage data item and output with a FROM clause. The contents of

screen-field are used as a prompt for the screen field described with the PROMPT clause.

For terminals operating in conversational mode, *screen-field* is used as a signal for input. In the Screen Section, a screen field description must precede the associated PROMPT clause in the same screen description.

During execution of the ACCEPT statement, the value specified in the prompt screen field is displayed before the terminal is able to receive input. The prompt value is always displayed in the first column of a screen line.

The following example illustrates a PROMPT clause with *screen-field* described in the Screen Section. When the associated ACCEPT statement executes, LAST NAME appears on the screen followed by a set of parentheses (delimiting the field size) and the cursor.

```
SCREEN SECTION .
01 ADDCUST-SCREEN  BASE  SIZE 24, 80 .
   05 NAME1-PROMPT     AT 3,2  VALUE "LAST NAME: " .
   05 LAST-NAME-FIELD  AT 3,13 PIC X(10) USING CUST-LAST-NAME
                                  LENGTH MUST BE 1 THRU 10
                                  PROMPT NAME1-PROMPT .
```

The next example illustrates a PROMPT clause with *screen-field* described in the Working-Storage Section and output with a FROM clause.

```
WORKING-STORAGE SECTION.
O1 NEWCUST-REC.
   05 NEW-LAST-NAME         PIC X(10) VALUE SPACES.
   :
01 WS-PROMPT-VALUE          PIC X(11) VALUE "LAST NAME: ".
   :
SCREEN SECTION.
01 NEWCUST-SCREEN.
   05 LAST-NAME-PROMPT AT 3,2  PIC X(11) FROM WS-PROMPT-VALUE.
   05 LAST-NAME-FIELD  AT 3,13 PIC X(10) USING NEW-LAST-NAME
                                  LENGTH MUST BE 1 THRU 10
                                  PROMPT LAST-NAME-PROMPT.
```

The PROMPT clause displays a screen field with or without parentheses depending on the screen-field definition.

- If the screen field is defined with a FROM or USING phrase, the PROMPT clause displays the value currently stored in the associated Working-Storage data item in parentheses following the prompt. For example, if LAST NAME (Brown) appears, Brown was the value entered during the last ACCEPT statement for this field.
- If the screen field is defined with a TO phrase, the PROMPT clause does not display parentheses.

RECEIVE Clause

The RECEIVE clause specifies whether screen-field data can be accepted from a terminal, another kind of device, or both. This option is supported only for applications running on 6530 terminals with version C00 (or later) microcode and 6AI (revision A00) firmware. If this clause is omitted, data can be accepted only from the terminal keyboard.

```
RECEIVE [ FROM ] { ALTERNATE             }
                 { ALTERNATE OR TERMINAL }
                 { TERMINAL              }
                 { TERMINAL OR ALTERNATE }
```

ALTERNATE

causes data to be accepted from a device other than the terminal. The other devices that Pathway supports are:

- Optical character recognition reader
- Optical bar code reader
- Magnetic string reader for badges or cards

ALTERNATE OR TERMINAL

causes data to be accepted from one of the alternate devices listed above and from the terminal keyboard.

TERMINAL

causes data to be accepted only from the terminal keyboard.

TERMINAL OR ALTERNATE

causes data to be accepted from one of the alternate devices listed above and from the terminal keyboard.

The RECEIVE clause restricts input from the terminal keyboard for screen fields defined with the ALTERNATE option. These fields can accept data only from an alternate device that is connected to a 6530 terminal.

You can use the SCREEN COBOL TURN statement to change this attribute to a previously defined option.

An example of the RECEIVE clause is:

```
SCREEN SECTION.
01  INVENTORY-REC-SCREEN   BASE SIZE 24, 80.
    :
    05  PROD-FIELD   AT 5, 28 PIC X(10) RECEIVE FROM ALTERNATE
                                         USING WS-PROD-ID.
    05  COUNT-FIELD  AT 7, 28 PIC X(10) RECEIVE FROM
                                         ALTERNATE OR TERMINAL
                                         TO WS-PROD-COUNT.
```

REDEFINES Clause

The REDEFINES clause specifies that the screen field being defined is an alternate interpretation of a previously defined field.

```
REDEFINES  field-name-2
```

field-name-2

is the previously defined field.

The two fields must be identical in size and display attributes.

The REDEFINES clause allows an ACCEPT statement to be issued for a given physical field for two cases using different rules. An example would be postal codes in the U.S. and in the U.K.

```
05  ZIP-US AT 10, 10          PIC 999999
                                  LENGTH 0, 5
                                  TO ZIP-US-WS.
05  ZIP-UK REDEFINES ZIP-US   PIC XXXXXX
                                  LENGTH 0, 6
                                  TO ZIP-UK-WS.
```

Either the REDEFINES or the AT clause must be included in every screen-field declaration. If both clauses appear in the screen-field declaration, they must refer to the same position.

SHADOWED Clause

The SHADOWED clause associates a nonliteral screen field with a Working-Storage data item (shadow item). The shadow item can be used to determine whether input was supplied for the screen field or to control selection of the screen field for output statements.

A shadow item is used in association with a SHADOWED modifier that can be included in the following statements: DISPLAY, RESET, SET NEW-CURSOR, and TURN. When one of these statements executes and includes a SHADOWED modifier, the shadow items that are referred to in the statement are automatically examined. Resulting action depends on the value of the shadow items.

```
SHADOWED [ BY ] data-name-1
```

data-name-1

is the data item to be associated with a nonliteral screen field. Define *data-name-1* in the Working-Storage Section with a size of one byte (PIC X, PIC 9, or PIC 9 COMP).

A shadowed screen field is associated with at least two Working-Storage items, the primary item and the secondary (shadow) item. For example, below a USING clause (which could be either TO or USING) associates a primary Working-Storage item with

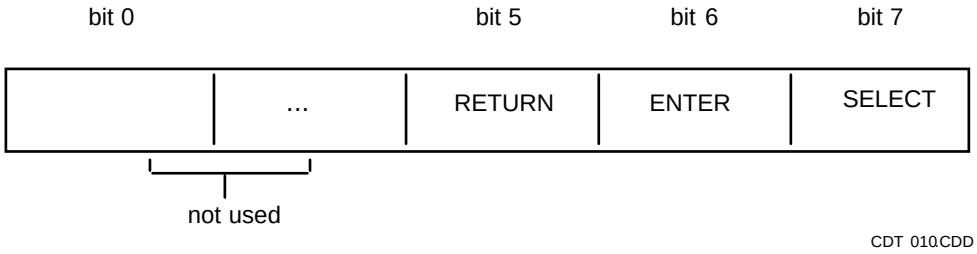
the screen field. The SHADOWED clause associates the shadow Working-Storage item with the screen field.

```
Screen Section                      Working-Storage Section

05 SCREEN-ITEMA ...                    :
     PIC X(10)                         :
     USING WS-ITEMA                 01 WS-ITEMA      PIC X(10).
     SHADOWED BY WS-A-SHADOW        01 WS-A-SHADOW   PIC 9 COMP.
```

If the screen field defined with a SHADOWED clause has an OCCURS clause, *data-name-1* given in the SHADOWED clause should be a data item having an OCCURS clause with the same maximum number of occurrences as the occurrences in the OCCURS clause of this corresponding field in the Screen Section.

The shadow item contains three subfields (bits). The diagram below shows the bits that constitute a shadow item.



The rightmost bit is the SELECT bit for the screen field. This bit is examined by the DISPLAY, TURN, RESET, and SET NEW-CURSOR statements that include the SHADOWED modifier. When the SHADOWED modifier is used in the statement, a field listed in the statement is not affected unless the SELECT bit in its shadow item is set to 1. This bit is set programmatically by the user by moving a value to the shadow item.

The bits to the left of the SELECT bit are the RETURN and ENTER bits. When a shadowed screen field is specified in an ACCEPT statement, a 1 or a 0 is stored into each of these bits. The values stored depend on the information received from the terminal. The values and the associated conditions are listed in Table 5-6. If the modified data tag (MDT) is set on (MDTON), the RETURN bit is always set on an ACCEPT statement.

Table 5-6. RETURN and ENTER Bit Values on Execution of an ACCEPT Statement

Information Received From Screen Field	RETURN *	ENTER
Tabbed across (field contains nothing)	0	0
Contains fill characters or spaces	1	0
Contains normal data	1	1
Attribute MDTON when ACCEPT executes:		
— Contains fill characters or spaces	1	0
— Normal data	1	1

* For a 6510 terminal, 1 is always stored in the RETURN bit.

Effect of ESCAPE Clause

If the ESCAPE clause is executed during the ACCEPT statement (for example, an abort input is specified for a terminal operating in conversational mode), the settings for the RETURN bit and the ENTER bit are undefined.

Using a SHADOWED Clause in DISPLAY Operations with DYNAMIC Modifier

When a shadowed screen field has an associated Working-Storage item item with its SELECT bit set to 1 and the DISPLAY DYNAMIC operation is used, the screen field acquires its initial screen contents from its FROM or USING Working-Storage data item. Otherwise, the initial screen contents are acquired from the compile-time literal value or the default initial value. Table 5-7 shows the effect of using DYNAMIC with the DISPLAY operation on screen field attributes.

Table 5-7. Effect of Shadowed Fields with DISPLAY Operation and DYNAMIC Modifier

	Screen Fields With These Characteristics Acquire Field Values…			
	From (Output) or USING (Input/Output)			To (Input)
DISPLAY BASE or DISPLAY OVERLAY Operation	SHADOWED: YES, SELECT=1	SHADOWED: YES, SELECT=0	SHADOWED: NO	
With DYNAMIC	FROM or USING Working-Storage data item	Literal*	Literal*	Literal*
Without DYNAMIC	Literal*	Literal*	Literal*	Literal*

* indicates the compile-time literal or default initial value.

Using a SHADOWED Clause

Using a shadow item usually involves the following operations:

- The program tests the shadow item to determine the value of ENTER and RETURN bits; these are set automatically by the ACCEPT statement depending on the data entered.
- The user programmatically sets the SELECT bit as desired by moving a value to the shadow item.

Table 5-8 shows the possible values for a shadow item and the corresponding bit patterns. The shadow item has different values depending on whether it is defined as numeric (PIC 9 or PIC 9 COMP) or alphanumeric (PIC X).

Table 5-8. Corresponding Shadow Item Values and Bit Values

		Bit Pattern*		
Shadow Item PIC 9	**Shadow Item PIC X**	**RETURN**	**ENTER**	**SELECT**
0	space	0	0	0
1	!	0	0	1
2	"	0	1	0
3	#	0	1	1
4	$	1	0	0
5	%	1	0	1
6	&	1	1	0
7	'	1	1	1

* 0—Bit is off. 1—Bit is set.

For example, if a numeric shadow item equals 4, the screen field associated with the shadow item held fill characters or blanks on the execution of the ACCEPT statement. In this situation, program logic could provide for displaying the screen field again by setting the SELECT bit. To set the SELECT bit, the program could move 5 to the shadow item and maintain the existing bit pattern for the RETURN and ENTER bits.

An example of the SHADOWED clause is:

```
SCREEN SECTION.
01  LOCATION-REC-SCREEN BASE SIZE 24, 80.
    05  STATE-FIELD  AT 5, 28  PIC X(2)
                      USING WS-STATE SHADOWED BY
                      WS-STATE-SHAD.        (1)
WORKING-STORAGE SECTION.
    05  WS-STATE-SHAD  PIC 9 COMP.          (2)
```

```
PROCEDURE DIVISION.
    MOVE 0 TO WS-STATE-SHAD.                (3)

    ACCEPT LOCATION-REC-SCREEN              (4)

    IF WS-STATE-SHAD EQUAL 4                (5)
       MOVE 5 TO WS-STATE-SHAD              (6)

    DISPLAY LOCATION-REC-SCREEN SHADOWED. (7)
```

(1) Specifies shadow

(2) Declares shadow item

(3) Stores 0 in shadow item

(4) ACCEPT statement stores 0 or 1 in RETURN and ENTER bits

(5) Tests shadow item

(6) Sets SELECT bit

(7) Displays only the screen fields referred to that have a SELECT bit set to 1

TO, FROM, USING Clauses

The TO, FROM, USING clauses are collectively referred to as data association clauses. These clauses specify a Working-Storage Section or Linkage Section data item that is associated with the screen field for moving data to and from the screen field. The clauses determine the general type of a field.

```
{ TO    } data-name-1
{ FROM  }
{ USING }
```

TO

specifies that data is to be moved from the screen field into the *data-name-1* area; this is an input association.

FROM

specifies that data is to be moved from the *data-name-1* area into the screen field; this is an output association.

USING

is equivalent to specifying both TO and FROM with the same data name.

data-name-1

is a Working-Storage data item associated with an elementary screen field; the field cannot be a subscripted item.

The following rules apply:

- A TO, FROM, or USING clause can be specified only with an elementary screen field.
- The TO and FROM clauses can both be specified for a screen field. If both clauses are specified, the data names can differ.
- If a data association clause is specified for any field, a PICTURE clause must also be specified for that field.
- The category of the screen field must be compatible with the associated data item in the Working-Storage Section or Linkage Section. A numeric edited field must be associated with a numeric data item, and an alphabetic edited field must be associated with an alphabetic or alphanumeric data item.

The data movement occurs in connection with the execution of a DISPLAY or ACCEPT statement. The statements explicitly or implicitly name the screen field containing the data association clause.

UPSHIFT Clause

The UPSHIFT clause specifies that lowercase alphabetic characters are to be translated to uppercase characters for input and output.

```
UPSHIFT [ INPUT        ]
        [ OUTPUT       ]
        [ INPUT-OUTPUT ]
        [ I-O          ]
```

If UPSHIFT appears by itself, INPUT-OUTPUT is assumed. If this clause is omitted, lowercase alphabetic characters for the field remain in lowercase. The UPSHIFT clause is a valid screen field attribute for PIC N fields but is really useful only on mixed fields (PIC N(10)A(10), for example).

Note. If an alphanumeric field is declared with the UPSHIFT and USER CONVERSION clauses, the TCP upshifts the field both before and after the user conversion procedure is called.

USER CONVERSION Clause

The USER CONVERSION clause gives a user-defined number to be passed with the field to a conversion procedure.

```
USER [ CONVERSION ] numeric-literal
```

The USER CONVERSION clause is used only if the application makes use of a user conversion procedure.

When an input screen field contains only blanks or fill characters, use the CONVERT BLANKS to allow the USER CONVERSION clause to be invoked.

When a user presses the Tab key twice to bypass a field, the MDT (modified data tag) does not get set. In such situations, use the WHEN ABSENT CLEAR clause to force blanks in a field. The WHEN ABSENT CLEAR clause, in conjunction with the CONVERT BLANKS clause, allows the USER CONVERSION clause to be invoked.

Refer to the *Compaq NonStop™ Pathway/iTS TCP and Terminal Programming Guide* for details regarding user conversion procedures.

VALUE Clause

The VALUE clause specifies the initial value of a screen field. The initial value is displayed during a DISPLAY BASE or OVERLAY statement, during a RESET DATA statement, and during screen recovery. The VALUE clause is required for literal screen fields.

```
VALUE nonnumeric-literal
```

nonnumeric-literal

is the character form of the specified value. The *nonnumeric-literal* must not be longer than the size specified for the field in the PICTURE clause; if it is shorter, the *nonnumeric-literal* is left-justified and padded with the fill character.

The value does not have to be valid according to conversion and checking restraints for input fields. However, if the value is not valid and the value is entered at the terminal during ACCEPT statement processing, the field is in error.

The VALUE clause cannot be used for a field using the OCCURS clause.

When the VALUE clause is used with a field that allows only double-byte data, the literal string provided must follow the same rules defined for a VALUE clause associated with a Working-Storage PIC N field.

The following example illustrates the VALUE clause:

```
SCREEN SECTION.
 01  ORD-DETAIL-SCRN  SIZE 12, 40.
     05  FILLER AT 1, 12  VALUE "ORDER DETAIL ENTRY".
     05  FILLER AT 2, 1   VALUE "CUSTOMER".
     05  ENTRY-GROUP  AT 5, 4.
         10  FILLER AT     @,     @  VALUE "ITEM".
         10  FILLER AT     @, @ + 9  VALUE "QUANT".
```

WHEN ABSENT or BLANK Clause

The WHEN ABSENT or BLANK clause controls the disposition of Working-Storage associated by TO or USING clauses with absent or blank fields.

If this clause is omitted, the default clauses are WHEN ABSENT SKIP (absent fields are skipped) and WHEN BLANK CLEAR (blank fields are cleared).

```
WHEN { ABSENT } { CLEAR }
     { BLANK  } { SKIP  }
```

ABSENT

indicates that the clause acts on a screen field that the terminal operator skips over without entering any data.

BLANK

indicates that the clause acts on a screen field in which a terminal operator enters either blank or fill characters, or skips over a BLANK field for which the screen field attribute, MDT (modified data tag), is set.

CLEAR

sets the Working-Storage to zero for numeric items and to spaces for alphabetic or alphanumeric items.

SKIP

leaves the Working-Storage unaltered.

When a user presses the Tab key twice to bypass a field, the MDT does not get set. In such situations, use the WHEN ABSENT CLEAR clause to force blanks in a field. The WHEN ABSENT CLEAR clause, in conjunction with the CONVERT BLANKS clause, allows the USER CONVERSION clause to be invoked.

WHEN FULL Clause

The WHEN FULL clause specifies the action to be taken when the last position of an input screen field is filled and additional characters are keyed into the terminal.

```
[ WHEN ] FULL { TAB  }
              { LOCK }
```

TAB

causes the cursor to advance to the next input field.

LOCK

causes the terminal to lock the keyboard.

If this clause is omitted, the default is LOCK.

The WHEN FULL clause is only effective for terminals that support more than one alternative action. Currently those terminals are the T16-6520, T16-6530, T16-6540, and the IBM3270.

Message Description Entry

A message description entry declares the characteristics of a message format. The message description entry is specified in the Message Section of a SCREEN COBOL program.

A number of message formats can be defined in the Message Section. Level 01 identifies the beginning of a message format. Subordinate group and elementary data items can be defined. The subordinate levels define groups or items that are consecutive fields within the message format.

The structure of the message description entry is similar to that of a data description entry. The message description entry is a series of declarative sentences, each beginning with a level number to indicate the hierarchy. A higher number indicates that the entry is subordinate to the previous entry. The 01 level is the highest statement in the paragraph. Subordinate entry levels can be any number from 02 through 49.

The syntax of a message description entry is:

```
MESSAGE SECTION.

01 message-name [ 01-clause... ].

[ level-number {group-name} [ group-clause... ].          ]
[                                                         ]
[  [ level-number {field-name} [ field-clause... ].  ]... ]
[                 {FILLER    }                            ]
```

message-name

is a user-defined name as described in Section 2, SCREEN COBOL Source Program.

01-clause

defines the characteristics of the message. Allowable clauses are:

```
FIELD-DELIMITER
MESSAGE-DELIMITER
MESSAGE FORMAT
PICTURE
RESULTING COUNT
TO/FROM/USING
USER CONVERSION
```

group-clause

defines the characteristics of the group item. Allowable clauses are:

```
OCCURS
OCCURS DEPENDING ON
PRESENT IF
USER CONVERSION
```

field-clause

defines the characteristics of the field item. Allowable clauses are:

```
FIELD STATUS
OCCURS
OCCURS DEPENDING ON
PICTURE
PRESENT IF
RESULTING COUNT
TO/FROM/USING
USER CONVERSION
```

FILLER Restrictions

The use of FILLER is permitted at any level. However, you must observe the following restrictions:

- When FILLER appears at the 01 level, that message cannot be the object of Procedure Division statements capable of Message Section access.
- When FILLER appears at any level, the FILLER item cannot have a FROM/TO/USING clause associated with it. A field defined as FILLER acts only as a place holder. This field cannot cause a reference to Working-Storage for reading or storing data.

FILLER Usage

To match the incoming data stream or create the desired output data stream, you can define the necessary FILLER fields in the Message Section.

1. On input, fields declared as FILLER are not processed. The TCP ignores a FILLER field and processes the next field.
2. On output, fields declared as FILLER are replaced with fill. Nonnumeric fields are filled with blanks.
3. Numeric fields are filled with zeros (binary if COMP, otherwise ASCII). PIC 1 fields are filled with zero bits.

The FILLER fields in the Message Section define the format of the data. FILLER fields in the paired Working-Storage Section data structure are optional.

A FILLER item in the Message Section acts only as a place holder. A FILLER item in the Working-Storage Section sets aside an area of real storage.

You might want to include FILLER items in Working-Storage to set aside storage for future use, or for documentation purposes.

FILLER items in Working-Storage cannot be referenced individually. However, the group item that includes the FILLER items can be referenced.

In the following example either of the paired Message Section and Working-Storage Section data structures is acceptable.

```
WORKING-STORAGE SECTION.
01 WS-MSG
   05 WS-A   PIC X
   05 WS-B   PIC X
   05 WS-C   PIC X
   05 WS-D   PIC X

MESSAGE SECTION.
01 MS-MSG
  05 MS-A   PIC X  FROM WS-A
  05 MS-B   PIC X  FROM WS-B
  05 FILLER PIC X
  05 MS-C   PIC X  FROM WS-C
  05 MS-D   PIC X  FROM WS-D


WORKING-STORAGE SECTION.
01 WS-MSG
   05 WS-A   PIC X
   05 WS-B   PIC X
   05 FILLER PIC X
   05 WS-C   PIC X
   05 WS-D   PIC X


MESSAGE SECTION.
01 MS-MSG
  05 MS-A   PIC X  FROM WS-A
  05 MS-B   PIC X  FROM WS-B
  05 FILLER PIC X
  05 MS-C   PIC X  FROM WS-C
  05 MS-D   PIC X  FROM WS-D
```

PICTURE and TO/FROM/USING Restrictions

A single-field message must have a PICTURE clause and a TO, FROM, or USING clause. For example:

```
WORKING-STORAGE SECTION.
01 WS-MSG   PIC X(10).

MESSAGE SECTION.
01 MSG      PIC X(10) USING WS-MSG.
```

When there are multiple fields within a message, the following rules apply:

- The 01 level item cannot have a PICTURE clause or a TO, FROM, or USING clause.
- The group level item cannot have a PICTURE clause or a TO, FROM, or USING clause.

- Each field level item, with the exception of the FILLER item, must have a PICTURE clause and a TO, FROM, or USING clause.

The following example shows the use of the PICTURE clause and the FROM clause in a multiple-field message.

```
WORKING-STORAGE SECTION.
01  WS-MSG.
  05  WS-GROUP.
    10  WS-FLD1    PIC X(10).
    10  WS-FLD2    PIC X(10).

MESSAGE SECTION.
01  MS-MSG.
  05  MS-GROUP.
    10  MS-FLD1    PIC X(10)   FROM WS-FLD1.
    10  MS-FLD2    PIC X(10)   FROM WS-FLD2.
```

USER CONVERSION and PRESENT IF Restrictions

The USER CONVERSION clause can be applied to 01 level items, group items, and field items. When the USER CONVERSION clause is used on the 01 level item or group item, it will be applied to the lower-level items as well, except under the following condition: if the lower-level item itself has a USER CONVERSION clause, then the upper level USER CONVERSION clause is overridden by the lower level USER CONVERSION clause.

Likewise, the PRESENT IF clause is propagated from one level (01 level or group item) down to any of its subordinate members. PRESENT IF clauses declared by subordinate members will override any previously propagated PRESENT IF attribute.

Message Description Entry Usage

The message format defined by the message description entry acts as a template through which the data is mapped as it is moved between the paired Working-Storage Section and Message Section PICTURE clauses. The purpose of this template is to order, format, and convert the data. The same message description can be used for both input and output.

The following is an example of a paired Working-Storage Section data description entry and Message Section message description entry:

```
DATA DIVISION.

WORKING-STORAGE SECTION.
01 WS-MSG
   05 WS-MSG-FLD1     PIC X(10)   VALUE IS 10.
   05 WS-MSG-FLD2     PIC X(10)   VALUE IS 20.
   05 WS-MSG-FLD3     PIC X(5)    VALUE IS 30.
   05 WS-MSG-FLD4     PIC 9(5)    COMP  VALUE IS 12345.
```

```
MESSAGE SECTION.
01 MSG                MESSAGE FORMAT IS FIXED.
   05 MSG-FLD1        PIC X(10)   FROM  WS-MSG-FLD2.
   05 MSG-FLD2        PIC X(5)    FROM  WS-MSG-FLD3.
   05 MSG-FLD3        PIC X(10)   FROM  WS-MSG-FLD1.
   05 MSG-FLD4        PIC 99,999  FROM  WS-MSG-FLD4.
```

For detailed information on how to define the message description entry to reorder, format, and convert the data, refer to the *Compaq NonStop™ Pathway/iTS TCP and Terminal Programming Guide*.

Clauses in Message Description Entry

FIELD-DELIMITER Clause

The FIELD-DELIMITER clause defines the character used to separate one field from another in a message. You can use this clause to define a field delimiter that corresponds to the field delimiter specified by the appropriate message format protocol. The default field delimiter is a comma (,).

The FIELD-DELIMITER clause can also disable the processing of the field delimiter. You must disable the field delimiter when dealing with the data stream as an entire message.

```
FIELD-DELIMITER [ IS ] { "character"   }
                       { decimal-value }
                       { ON            }
                       { OFF           }
```

"*character*"

is any single ASCII character enclosed in quotation marks. The default field delimiter is a comma (,).

decimal-value

is a number from 0 through 255. This number is the decimal representation of the desired ASCII or EBCDIC character. For example, 80 represents the letter P in ASCII or the ampersand (&) in EBCDIC.

ON

enables processing of delimiters at the field level.

OFF

disables processing of delimiters at the field level.

The following rules apply:

- To use field delimiters, you must specify FIXED-DELIMITED or DELIMITED in the MESSAGE FORMAT clause.

- If you omit the FIELD-DELIMITER clause and the message format is FIXED-DELIMITED or DELIMITED, field delimiter processing is enabled and the default field delimiter is a comma (,).
- When processing a message on a field-by-field basis, you can use the FIELD-DELIMITER clause with the RESULTING COUNT clause. Refer to the RESULTING COUNT Clause for details.

FIELD STATUS Clause

The FIELD STATUS clause identifies a Working-Storage Section data group or item that receives status information about an associated Message Section field. The FIELD STATUS Working-Storage Section item is updated when data is mapped through the associated Message Section field. The data is mapped through the field as the data is moved into or out of Working-Storage by statements such as: SEND MESSAGE, TRANSFORM, RECEIVE UNSOLICITED MESSAGE, or REPLY TO UNSOLICITED MESSAGE.

```
FIELD STATUS [IS] data-item-1
```

data-item-1

is a Working-Storage group or elementary data item. The size of *data-item-1* must be at least 4 bytes or compiler error 369 is generated. The following rules apply:

- The first 2 bytes of *data-item-1* contain the shadow portion of the field status information. The shadow portion is used with the PRESENT IF clause to determine if a conditionally present field is present.
- The second 2 bytes of *data-item-1* contain the field error portion of the field status information. The field error portion is used to obtain information on editing errors on fields where editing is specified.
- The following format is recommended for *data-item-1*; this format is recommended so you can have accessible variable names for the shadow portion and the field error portion:

```
02  FIELD-STATUS-AREA.
    03   SHADOW-INFO          PIC 9(4) COMP.
    03   FIELD-ERROR          PIC 9(4) COMP.
```

- You must define *data-item-1* as COMP data because the shadow values and error numbers that the TCP moves into these locations are binary values.
- *data-item-1* is always updated when data is mapped through the associated Message Section field. If you use FIELD STATUS to determine the presence of a conditionally present field, you can also use the field status information to detect editing errors on the field.

Shadow Data Item

When you use the FIELD STATUS clause with the PRESENT IF clause, you can test the shadow portion of the FIELD STATUS Working-Storage group item to determine if a conditionally present field is present.

You interpret the shadow data item based on the values it can assume. Those values depend on two factors: the state of the conditionally present field and whether the data is being input or output.

- Entered state—The field was received on input and was present in the data stream.
- Stored state—The field that was entered or generated was placed in the designated Working-Storage or Linkage Section data item.
- Selected state—The field should be selected for output from the designated Working-Storage or Linkage Section data item.

The entered and stored states reflect what happens during input processing. The TCP updates the value of the shadow item to reflect the entered or stored state.

The selected state only has effect during output processing. The selected state is under the control of the SCREEN COBOL program. Once the program sets the selected state, that state remains in effect until reset by the program.

During input processing, the TCP always updates the value of the shadow item to reflect the entered or stored state. This allows you to get the exact status of the last input operation for a field. This shadow information is important when you are processing conditionally present fields.

An input shadow data item can assume the value of 2, 3, 4, 5, 6 or 7.

- Shadow field = 2—A field has been entered (is present) but failed an edit operation; it is not stored.
- Shadow field = 4—A field has been stored but was not physically entered. The TCP generated the pad data to store in this field.
- Shadow field = 6—A field has been entered, has passed the edit phase, and has been stored.

In addition, the values that a shadow data item can assume after an input operation are offset by the setting of the selected state. Table 5-9 shows the possible states and the actual values of the shadow data item if the selected state is 0 or the selected state is 1.

Table 5-9. FIELD STATUS Clause Shadow Values

	Shadow Values With Selected State of 0		Shadow Values With Selected State of 1	
State	**Input**	**Output**	**Input**	**Output**
Entered	2	N.A.	3	N.A.
Generated and Stored	4	N.A.	5	N.A.
Entered and Stored	6	N.A.	7	N.A.
Selected	N.A.	0	N.A.	1

The TCP generates pad data if a field has not been entered, but needs to be physically stored (shadow field = 4 or 5). The TCP generates pad data so that the application does not reference previously stored data.

The pad data generated for the target Working-Storage data item is blank spaces for a nonnumeric item and zeros (binary if COMP, otherwise ASCII) for a numeric item.

The TCP considers a field to be not entered and generates pad data when:

- The message specifies that a field is conditionally present and the condition is not met so that the TCP finds the field to be absent
- The message specifies that field delimiters are enabled and optionally that message delimiters are enabled. The TCP finds:
  - Two consecutive delimiters—two field delimiters or a field delimiter followed by a message delimiter
  - A field delimiter as the last character of the message and the associated field is the last field of the message

The selected state only has effect during output processing on a field. The selected state provides you with the capability to override PRESENT IF processing on a field. Overriding PRESENT IF processing will be the exception rather than the common course of action. However, there might be instances where you need to output a given field even if the PRESENT IF processing shows the field to be absent.

The selected state can assume the value of 1 or 0.

- Selected state = 1—The TCP unconditionally outputs the field. The field is output regardless of the result of any conditionally present processing on the field caused by the PRESENT IF clause.
- Selected state = 0—The result of the PRESENT IF processing on the field is a factor in the decision to output that field.

Table 5-10 shows the logical relationship between the FIELD STATUS shadow data item and PRESENT IF processing.

Table 5-10. Relationship Between Selected State and PRESENT IF

PRESENT IF and FIELD STATUS	Values	Meaning	TCP Action
PRESENT IF *data-item*	1	Field Present	Output Field
FIELD STATUS *shadow-data-item*	1	Select Field for Output	
PRESENT IF *data-item*	0	Field Absent	Output Field
FIELD STATUS *shadow-data-item*	1	Select Field for Output	
PRESENT IF *data-item*	1	Field Present	Output Field
FIELD STATUS *shadow-data-item*	0	Selected State Has No Meaning	
PRESENT IF *data-item*	0	Field Absent	Does Not Output Field
FIELD STATUS *shadow-data-item*	0	Selected State Has No Meaning	

The field error data item allows the TCP to report specific edit errors that relate to the specified Message Section field. The following discussion on error processing tells you how to process the field error data item.

Use the ON ERROR clause to detect any error that occurs on input or output of the message from Working-Storage. Use the FIELD STATUS clause to detect specific edit errors.

If an error is detected on either an input or output operation, the ON ERROR path is taken. The processing of the ON ERROR clause for RECEIVE UNSOLICITED MESSAGE, REPLY TO UNSOLICITED MESSAGE, SEND MESSAGE, and TRANSFORM is the same as that for the CALL and SEND statements.

As part of the ON ERROR processing you need to check for a TERMINATION-STATUS of 5 or 15 to determine if any edit errors have occurred (5 indicates input phase and 15 indicates output phase). If TERMINATION-STATUS is 5 or 15, you can then process the FIELD STATUS field error data item to determine which field or fields had edit errors.

The process of deciding which message template the FIELD STATUS data item belongs to differs for the input message and the output message.

For input messages, you need to know which of the message templates of the YIELDS list you were processing when the error(s) occurred.

The relative position of the YIELDS list is returned in TERMINATION-SUBSTATUS.

The position is returned in TERMINATION-SUBSTATUS instead of TERMINATION-STATUS because this is the ON ERROR case. At this point TERMINATION-STATUS holds the error number. If this were the normal case, not ON ERROR, TERMINATION STATUS would be used to define the relative position in the YIELDS list.

The following SEND MESSAGE shows the YIELDS associated with the input messages when the ON ERROR clause is executed due to an edit error.

```
SEND MESSAGE MSG-3-OUT-M-1
    REPLY CODE FIELD IS WS-MSG-4-IN-FROM-MSG-4-IN-CODE
       CODE 1 YIELDS MSG-3-IN,
       CODE 2 YIELDS MSG-4-IN,
       CODE 3 YIELDS MSG-3-IN,
       CODE 4 YIELDS MSG-4-IN,
    ON ERROR PERFORM IDS-SERVER-SEND-ERROR.
```

The following PERFORM statement shows how to use TERMINATION- SUBSTATUS to decide which input message template to process.

```
IDS-SERVER-SEND-ERROR.
    PERFORM ONE OF
       PROC-MSG-3-IN-EDIT-STATUS,
       PROC-MSG-4-IN-EDIT-STATUS,
       PROC-MSG-3-IN-EDIT-STATUS,
       PROC-MSG-4-IN-EDIT-STATUS,
          DEPENDING ON TERMINATION-SUBSTATUS.
```

The following table shows which input message correlates with which TERMINATION-SUBSTATUS.

Message Template	TERMINATION-SUBSTATUS
YIELDS MSG-3-IN	1
YIELDS MSG-4-IN	2
YIELDS MSG-3-IN	3
YIELDS MSG-4-IN	4

For output messages, you know which set of FIELD STATUS data items to interrogate because only one message template can be specified on output.

Table 5-11 lists the error numbers that can be found in the field error data item.

Table 5-11. Relevant SEND MESSAGE Edit Advisory Error Numbers (page 1 of 2)

		Applies to	
Edit Advisory Error Number	**Meaning**	**Input**	**Output**
4	Length too short	Yes	No
6	Length longer than field	Yes	No
7	Alphanumeric editing expected an insertion character	Yes	No
8	Alphanumeric editing expected a digit	Yes	No
9	Alphanumeric editing expected a blank or alphabetic character	Yes	No

Table 5-11. Relevant SEND MESSAGE Edit Advisory Error Numbers (page 2 of 2)

Edit Advisory Error Number	Meaning	Applies to Input	Applies to Output
10	Numeric source does not match picture	Yes	Yes
11	Numeric value overflow during conversion	Yes	Yes
16	Field is absent	Yes	No

MESSAGE-DELIMITER Clause

The MESSAGE-DELIMITER clause defines the character(s) used to delimit the message's end. You can use this clause to define a message delimiter that matches the message delimiter that your program expects. The default message delimiter is two slashes (//).

The MESSAGE-DELIMITER clause can also disable processing of the message delimiter. You can disable processing of the message delimiter if message delimiters are not expected in the message format.

```
MESSAGE-DELIMITER [ IS ] { char-code [ , char-code ] }
                         { OFF                       }
```

char-code

is any single ASCII character in quotation marks or is a number from 0 through 255. This number is the decimal representation of the desired ASCII or EBCDIC character. For example, 80 represents the character P in ASCII or the character & in EBCDIC.

OFF

disables processing of delimiters at message level. The following rules apply:

- To use message delimiters, you must specify FIXED-DELIMITED or DELIMITED in the MESSAGE FORMAT clause.
- If you omit the MESSAGE-DELIMITER clause, message delimiter processing is enabled and the default message delimiter is two slashes (//).
- When dealing with the data stream on a message level, you can use the MESSAGE-DELIMITER clause with the RESULTING COUNT clause. Refer to the RESULTING COUNT clause for details.

MESSAGE FORMAT Clause

The MESSAGE FORMAT clause specifies the format of the data in the message. The MESSAGE FORMAT clause can appear only at the 01 level and determines the format of the entire record.

```
MESSAGE FORMAT [ IS ] { FIXED           }
                      { DELIMITED       }
                      { FIXED-DELIMITED }
                      { VARYING1        }
                      { VARYING2        }
```

`FIXED`

indicates that the message is fixed length. This is the default format.

`DELIMITED`

indicates that the message is made up of varying-length fields and that each field is terminated by a specified delimiter.

`FIXED-DELIMITED`

indicates that the message is made up of fixed-length fields and that each of these fields is terminated by a specified delimiter.

`VARYING1`

indicates that the message is variable length with a one-byte count to specify the actual number of characters (bytes) it contains. If the count is zero, the message is empty. The maximum size for a message of VARYING1 format is 255 bytes.

`VARYING2`

indicates that the message is variable length with a two-byte count that specifies the actual number of characters (bytes) it contains. If the count is zero, the message is empty.

The following rules apply:

- If you omit the MESSAGE FORMAT clause, the message format is FIXED.
- The format you choose depends on the intelligent device. For instance, some personal computers expect only fixed-length messages; other devices can handle variable-length messages.
- You must specify the maximum length of a variable-length message in the PICTURE clause for the message.
    - For a single-field message there is only an 01 level data item. The PICTURE clause in this 01 item specifies the maximum length of the message.
    - For a multiple-field message there is a PICTURE clause in each field. The maximum length of the message is the sum of the individual field lengths.

There cannot be a PICTURE clause in the 01 level data item in a multiple-field message.

- For FIXED and FIXED-DELIMITED formats, the message fields are fixed to the declared size; therefore, the message size is a fixed length.
  - FIXED format message lengths are the sum of their individual field lengths.
  - FIXED-DELIMITED messages are the sum of their individual field lengths plus the following:
  - If the FIELD-DELIMITER clause for the message is not OFF, add one byte for each possible field delimiter.
  - If the MESSAGE-DELIMITER clause for the message is not OFF, add one or two bytes for the message delimiter, depending on the size of the delimiter.
- For VARYING1 and VARYING2 formats, the TCP maintains the count of the actual number of data characters in the message.
  - On input, the TCP expects and removes a one-byte or two-byte message length count from the front of the message.
  - On output, the TCP prefixes a one-byte or two-byte message length count. Trailing blanks are truncated.
- For DELIMITED formats, the length of each field can vary in size from 0 to the declared length. If the FIELD-DELIMITER clause is not off, each field will be separated by a one-character field delimiter. If the MESSAGE-DELIMITER clause is not off, the message can be optionally terminated by either a one-character or two-character message delimiter.
  - On input, the TCP disassembles a message based on the declarations in the FIELD and MESSAGE-DELIMITER clauses.
  - With the exception of the last field in the message, field boundaries are determined by the field delimiter or by a field's declared length if FIELD-DELIMITER is off.
  - The last field in the message can be delimited by the message delimiter or by the physical end of the message if MESSAGE-DELIMITER is off.
  - On output, the TCP assembles a message out of the message's declared fields. You can control a field's actual length with the RESULTING COUNT clause. In the absence of RESULTING COUNT, the TCP uses the field's declared length. Trailing blanks are truncated.
  - If field and message delimiters have not been disabled, the TCP separates fields with a field delimiter and terminates messages with a message delimiter.
- In the DELIMITED and FIXED-DELIMITED message formats, the PIC 1 data type is allowed only if FIELD-DELIMITERS is OFF. SCREEN COBOL will not report this violation as an error.

Example 1

Message MSG1 has a fixed length of two characters.

```
01  MSG1 PIC X(2) MESSAGE FORMAT IS FIXED ...
```

This message could also have been described as:

```
01  MSG1 PIC X(2) ...
```

When the MESSAGE FORMAT clause is omitted, the message length is fixed by default.

Example 2

Message MSG2 can contain up to five characters of data. The actual number of data characters it contains is kept in a one-byte count; this count is not included in the size of the message specified in its PICTURE clause.

```
01  MSG2  PIC X(5) MESSAGE FORMAT IS VARYING1 ...
```

Example 3

Message MSG3 can contain up to eight characters of data. The actual number of characters is kept in a two-byte count that is not included in the message size specified in the PICTURE clause.

```
01  MSG3  PIC X(8) MESSAGE FORMAT IS VARYING2 ...
```

Example 4

The following example has a DELIMITED message format and uses the field delimiters to process the message.

```
WORKING-STORAGE SECTION.
01 WS-MSG4-RECORD-1.
   02  WS-LENGTH-1         PIC 9(4) COMP.
   02  WS-RECORD-1-GROUP.
       03  WS-RECORD-1     PIC X(1)
                           OCCURS 1 TO 16 TIMES
                           DEPENDING ON WS-LENGTH-1.
01 WS-MSG4-RECORD-2.
   02  WS-LENGTH-2         PIC 9(4) COMP.
   02  WS-RECORD-2-GROUP.
       03  WS-RECORD-2     PIC X(1)
                           OCCURS 1 TO 16 TIMES
                           DEPENDING ON WS-LENGTH-2.

MESSAGE SECTION.
01 MSG4  MESSAGE FORMAT IS DELIMITED
         MESSAGE-DELIMITER IS OFF
         FIELD-DELIMITER IS ":".
   05  MS-RECORD-1         PIC X(16) USING WS-RECORD-1
                           RESULTING COUNT IS WS-LENGTH-1.
   05  MS-RECORD-2         PIC X(16) USING WS-RECORD-2
                           RESULTING COUNT IS WS-LENGTH-2.
```

Message MSG4 consists of two fields and can contain up to 32 characters of data. The actual length of each field, in bytes, is placed into a separate Working-Storage location, WS-LENGTH-1 and WS-LENGTH-2 respectively. The field delimiter is not stored in the target Working-Storage location, WS-RECORD-1 or WS-RECORD-2. The field delimiter is not reflected in the RESULTING COUNT value.

Example 5

The following example has a DELIMITED message format but inputs the entire message without regard to delimiter processing.

```
WORKING-STORAGE SECTION.

01 WS-MSG5.
   05 WS-RECORD-LENGTH   PIC 9(4) COMP.
   05 WS-DATA-GROUP.
      10  WS-DATA        PIC X(1)
                         OCCURS 1 TO 1000 TIMES
                         DEPENDING ON WS-RECORD-LENGTH.

MESSAGE SECTION.

01 MSG5   MESSAGE FORMAT IS DELIMITED
          MESSAGE-DELIMITER OFF
          FIELD-DELIMITER OFF.
   05 MS-RECORD       PIC X(1000) USING WS-DATA
                      RESULTING COUNT IS WS-RECORD-LENGTH.
```

Message MSG5 can contain up to 1000 characters of data. The actual length of the field, in bytes, is placed into a separate Working-Storage location (WS-RECORD-LENGTH).

The Procedure Division usage would be to reference WS-DATA-OCCURS, using the value of WS-RECORD-LENGTH as the value for the actual limit of occurrences.

OCCURS/OCCURS DEPENDING ON Clauses

The OCCURS and OCCURS DEPENDING ON clauses specify a series of fixed or variable-length fields within a message. These clauses can be specified at the group or field level.

The OCCURS clause can be nested three levels deep. The OCCURS DEPENDING ON clause cannot be nested. However, the OCCURS clause can be nested within the OCCURS DEPENDING ON clause.

There is no limitation to how many times the OCCURS or OCCURS DEPENDING ON clauses can be repeated within a message.

```
OCCURS { num-lit-1 TIMES                }
       {                                }
       { num-lit-2 TO num-lit-3 TIMES   }
       {    DEPENDING [ON] num-name-1   }
```

num-lit-1

indicates the number of elements in a field. *num-lit-1* is a positive numeric literal that is greater than or equal to one.

num-lit-2

indicates the minimum number of elements in a field. *num-lit-2* is a positive numeric literal that is greater than or equal to zero.

num-lit-3

indicates the maximum number of elements in a field. *num-lit-3* is a positive numeric literal where *num-lit-3* is greater than *num-lit-2*.

num-name-1

provides a count of the number of occurrences of the data item that makes up the group or field. *num-name-1* is a Working-Storage or Linkage Section elementary numeric data item.

The following rules apply:

- The compiler does not do minimum bounds checking when processing message fields.
- When processing a message that contains a variable number of occurrences of a field, the number of occurrences of the field must be previously specified in the program. Therefore, the OCCURS DEPENDING ON clause can be used in the Message Section only under the following conditions:
  - You can only use the OCCURS DEPENDING ON clause in the Message Section to output a message. You cannot use OCCURS DEPENDING ON in the Message Section to input a message.
  - You should define the message with MESSAGE FORMAT IS DELIMITED in the Message Section.
  - You must declare the number of occurrences of the field you want to output in a Working-Storage Section elementary data item, *num-name-1*. For example:

```
WORKING-STORAGE SECTION.

01 WS-Message-Out
   05 WS-Message-Length  PIC 9(4) COMP.
   05 WS-Message-Data    PIC X OCCURS 1 TO 1024 TIMES
                         DEPENDING ON WS-Message-Length.

MESSAGE SECTION.

01 Message-Out           PIC X
                         FROM WS-Message-Data
```

```
                                OCCURS 1 TO 1024 TIMES
                                DEPENDING ON WS-Message-Length.
```

Note. Another way of sending variable length messages uses the RESULTING COUNT clause and results in faster processing. The following example accomplishes the same thing as the previous one, but runs three times faster.

```
WORKING-STORAGE SECTION.

01 WS-Message-Length    PIC 9(4) COMP.

01 WS-Message-Out
   05 WS-Message-Data   PIC X OCCURS 1 TO 1024 TIMES
                        DEPENDING ON WS-Message-Length.

MESSAGE SECTION.

01 Message-Out          MESSAGE FORMAT IS DELIMITED
                        MESSAGE DELIMITER IS OFF
                        FIELD DELIMITER IS OFF.
   05 Message-Data      PIC X(1024)
                        RESULTING COUNT IS
                        WS-Message-Length
                        FROM WS-Message-Out.
```

- The Working-Storage Section item associated with the FROM/TO/USING Message Section item defined by an OCCURS clause must also have an OCCURS clause or an OCCURS DEPENDING ON clause.
  - The number of occurrences of this Working-Storage Section item must be greater than or equal to the number of occurrences of the Message Section item.
  - The nested structure of this Working-Storage Section item must be identical with that of the Message Section item.
- When an OCCURS DEPENDING ON clause is used to define a group or field in the Working-Storage Section, that group or field must be the last item in the record.
- If a message description entry in the Message Section contains several OCCURS DEPENDING ON clauses, the message must be associated with several Working-Storage Section 01 records, where each record can contain only one OCCURS DEPENDING ON clause. For example:

```
WORKING-STORAGE SECTION.
01 WS-Msg4-Record-1.
   05  WS-Length-1         PIC 9(4) COMP.
   05  WS-Record-1         PIC X
                           OCCURS 1 TO 10 TIMES
                           DEPENDING ON WS-Length-1.
01 WS-Msg4-Record-2.
   05  WS-Length-2         PIC 9(4) COMP.
   05  WS-Record-2         PIC X
                           OCCURS 1 TO 10 TIMES
                           DEPENDING ON WS-Length-2.
```

```
MESSAGE SECTION.
01 Msg4-Out.
   05  MS-Record-1          PIC X
                            FROM WS-Record-1
                            OCCURS 1 TO 10 TIMES
                            DEPENDING ON WS-Length-1.
   05  MS-Record-2          PIC X
                            FROM WS-Record-2
                            OCCURS 1 TO 10 TIMES
                            DEPENDING ON WS-Length-2.
```

PICTURE Clause

The PICTURE clause specifies the length, data type, and editing format of a message field.

```
{ PIC     } [ IS ] character-string
{ PICTURE }
```

character-string

is a symbol that determines the category of a field, places restrictions on values assignable to the field, and defines editing operations.

The *character-string* can take the same form as that described in the data description entry with the following exceptions:

- Numeric edited and alphanumeric edited forms are allowed.
- Bit fields are allowed.

The maximum message size is 32,000 bytes.

Note. Each message field description must include a PICTURE clause.

The following rules apply:

- A maximum of 30 characters is allowed in *character-string*. When the same PICTURE character repeats, you can write it once followed by an unsigned integer enclosed in parentheses. The integer indicates how many times that character is repeated. For example, the following are equivalent:

  ```
  PIC 9(5)
  ```

  ```
  PIC 99999.
  ```

- Although only 30 characters can make up a *character-string*, you can use the repetition technique to define items longer than 30 characters.
- The *character-string* symbols that are defined in Table 5-12 are used to describe a message field.

Table 5-12. Message Description Entry PICTURE Character-String Symbols (page 1 of 2)

Symbol	Meaning
A	Represents a character position for a letter of the alphabet or a space character. If the character is not a letter or a space or a data bit from a PIC 1 field, it is flagged as an error.
B	Represents a character position where a space must occur in the input. The space is deleted during conversion into its associated data item. This character should not be used as the rightmost character of a numeric picture because trailing spaces are removed before conversion.
P	Indicates an implicit decimal position (with value zero) to be used in aligning the decimal point in the numeric result. Refer to the description of the V symbol for cautions.
V	Represents the assumed decimal point location in noninteger numeric items. Only one V can appear in a PICTURE character-string, and V cannot occur with an explicit decimal point in the same PICTURE character-string. A V is not counted in the item's size.
X	Represents any character in the ASCII character set whose numeric representation is between 0 and 255. Each X in the character string is counted as one character in determining the size of the message.
Z	Represents a position that must be a digit or must be a space if no digits appear to the left of the symbol. The symbol is replaced by a space during editing only when it is one of a set of multiple Z symbols. A space is equivalent to a zero for purposes of conversion.
1	Represents an individual bit. When used with a repeat factor of n, this field is made up of n bits.
9	Represents a character position that must be a digit.
0	Represents a character position where a zero must appear. The zero is deleted during conversion into the associated data item.
/	Represents a character position where a right slant must appear. The / is deleted during conversion into the associated data item.
,	Represents a character position where a comma must appear if any digits appear to the left of it. If no digits appear to the left of the symbol, the character must be a space (or other floating insertion character). The comma is deleted during conversion into the associated data item.
.	Represents a character position where a period must appear and indicates decimal point alignment. The period is deleted during conversion into the associated data item.
+	Represents a position where either a plus or a minus sign must appear. Multiple plus signs represent positions that must contain some number of digits preceded by a single plus sign or a single minus sign, preceded by spaces. The symbol is replaced by a space during editing only when it is one of a set of multiple plus signs.

Table 5-12. Message Description Entry PICTURE Character-String Symbols (page 2 of 2)

Symbol	Meaning
-	Represents a position where either a space or a minus sign must appear. Multiple minus signs represent positions that must contain some number of digits preceded by an optional minus sign, preceded by spaces. The symbol is replaced by a space during editing only when it is one of a set of multiple minus signs.
CR	Represents two positions that must contain the characters CR or spaces. These symbols are replaced by spaces during editing if the value is not negative.
DB	Represents two positions that must contain the characters DB or spaces. These symbols are replaced by spaces during editing if the value is not negative.
*	Represents a position that must be a digit or an asterisk. If the position is a digit, the digit must be to the left of all asterisks.
$	Represents a position where a currency symbol must appear. Multiple currency symbols represent positions that must contain some number of digits preceded by a currency symbol, preceded by spaces. The symbol is replaced by a space during editing only when it is one of a set of multiple currency symbols.

The PIC 1 format allows you to process fields of 1 bit in length or to process data in encoding schemes that require character lengths other than 1 byte. For example, packed decimal or binary coded decimal (BCD) require character lengths of 4 or 6 bits respectively. Here are some examples of the PIC 1 format:

```
PIC 1.      single binary digit (1-bit field)
PIC 1(6).   6 bits together (BCD character)
PIC 1(8).   8 bits together (1-byte field)
PIC 1(64).  64 bits (8 bytes) to form a single field
```

This format gives you the capability to decompose a message data stream on a bit-by-bit basis. This capability is useful when processing bit maps within a message.

The PIC 1 field specification raises the issue of data alignment to byte boundaries because it is now possible to define fields that are not exact multiples of full bytes.

When using the PIC 1 format, it is recommended that the Message Section maintain a byte-oriented structure. For example:

```
MESSAGE SECTION.

01  MS-ATM-Record
    05  MS-Bit-Map.
           10 MS-Bit-1-Savings              PIC 1(1)
                USING WS-Bit-1.
           10 MS-Bit-2-Checking             PIC 1(1)
                USING WS-Bit-2.
           10 MS-Bit-3-Credit-Card          PIC 1(1)
                USING WS-Bit-3.
           10 FILLER                        PIC 1(5).
    05 MS-Savings.
           10 Savings-Data                  PIC X(15)
                TO WS-Savings-Data
```

```
                PRESENT IF MS-Bit-1-Savings
                FIELD STATUS IS WS-Field-Status-Savings.
```

Also, it is possible to have a field other than a byte-sized one preceding a byte-sized field. For example:

```
MESSAGE SECTION.

01 MS-Message.
   05  Field-1          PIC 1.
   05  Field-2          PIC X.
```

The logical length of the above message is 9 bits rather than 16 because there is no padding between the single bit field and the following byte field. However, notice that data communication protocols round up the physical length of the message to a full byte boundary. The TCP receives two physical bytes, but only processes the first 9 bits. Therefore, you need to declare the message as follows:

```
MESSAGE SECTION.

01 MS-Message.
   05 Field-1          PIC 1.
   05 Field-2          PIC X.
   05 FILLER           PIC 1(7).
```

The 7 bits needed to round up the physical length to 16 bits can be declared as FILLER because these bits have to be declared but not processed.

Item Size

The size of a message field is determined by the symbols of its PICTURE string. The character-string symbols DB and CR each count as two character positions. Symbols V and P do not count as character positions. The character-string symbol 1 represents one bit. All other symbols count as one character position.

Input Editing Rules for Numeric and Alphanumeric Data

The input editing process is different for the two classes of the input item:

1. Alphanumeric input—Only the left-hand portion of the picture corresponding to the actual number of input characters must be matched. The remaining portion of the picture is ignored.
2. Numeric input—Leading and trailing spaces and fill characters are first removed from the input data string. Then an attempt is made to match each character in the picture with a character in the input data, proceeding from right to left. If no match is made, the data is considered to be in error.

Some picture symbols are special in that the positions they represent might be omitted from the input data string. Symbols that can be included in this category are Z, comma, multiple plus and minus signs, CR, DB, and multiple currency signs. If a mismatch occurs with an input character of one of these types, and if a space would be acceptable at that point in the input string, the data is not considered in error; the picture symbol is replaced by a space, and the editing process attempts to match the input character with the next picture symbol.

Format Conversion Rules for PIC 1 Format (Bit Fields)

The following rules apply when data involving bit fields is passed through the Message Section.

When a single field within the Message Section is defined as a PIC 1(n), the processing performed is dependent on the associated Working-Storage data item type.

Input Operations

- The source is a Message Section item. The destination is a Working-Storage Section item.
- For a Working-Storage PIC 9 COMP field, the data bits from the Message Section item are placed into the least significant bit positions of the Working-Storage item. There is high order zero fill or truncation if needed.
- For a Working-Storage PIC 9 field, the binary numeric value is translated into the ASCII characters that represent the decimal value of the field. If the Working-Storage item cannot support the number, the error "value overflow" is reported and the requester program is suspended, or an ON ERROR clause is activated.
- For a Working-Storage PIC X field, the data bits from the Message Section item are placed into the most significant bit positions of the Working-Storage item. There is low order zero fill or truncation if needed.
- For a Working-Storage PIC A field, the compiler detects and flags an error. A PIC A field is composed of ASCII characters that must represent: A through Z, a through z, or space.
- Truncation rules for input operations
    - Truncation occurs when the destination is shorter than the source.
    - If the destination Working-Storage field is PIC X, trailing bits are truncated. No indication of truncation is given to the application program.
    - If the destination Working-Storage field is PIC 9 COMP and that item cannot support the size of the Message Section item, the following happens: the TCP indicates that an overflow occurred by either suspending the requester program, or by activating ON ERROR processing.
- Filling rules for input operations
    - Filling occurs when destination is longer than source.
    - Destination Working-Storage PIC X fields are padded in the least significant (trailing) portion of the field with nulls (binary zeros).
    - Destination Working-Storage PIC 9 fields are filled in the most significant (leading) portion of the field with nulls (binary zeros).

Output Operations

- The source is a Working-Storage Section item. The destination is a Message Section item.
- Data is retrieved from the least significant or most significant portion of the source field, depending on whether the source field is numeric or nonnumeric.
- For a Working-Storage PIC 9 field, data bits from the least significant portion of the field are used to supply the data for the Message Section field item. There is right to left processing of both source and destination fields, least significant to least significant.
- For a Working-Storage PIC A or X field, data bits from the most significant portion of the field are used to supply the data for the Message Section field item. There is left to right processing of both source and destination fields, most significant to most significant.
- Truncation rules for output operations
  - Truncation occurs when the destination is shorter than the source.
  - For Working-Storage PIC X and A fields (nonnumeric data), trailing bits are truncated. The length of the destination Message Section item determines how many bits are allowed. No indication of truncation is given to the application program.
  - For Working-Storage PIC 9 and 9 COMP fields (numeric data), no loss of data is allowed. The TCP indicates that an overflow occurred by either suspending the requester program, or by activating ON ERROR processing.
- Filling rules for output operations:
  - Filling occurs when destination is longer than source.
  - For Working-Storage PIC X fields, the TCP pads the least significant (trailing) portion of the Message Section field with nulls (binary zeros).
  - For Working-Storage PIC 9 fields, the TCP fills the most significant (leading) portion of the Message Section field with nulls (binary zeros).

Example

On input, if a PIC 1(9) Message Section item is placed into a PIC X(2) or PIC 9(2) Working-Storage item, the resulting values are:

```
PIC X(2) bit positions:  0 1 2 3 4 5 6 7   0 1 2 3 4 5 6 7
resulting X(2) data:     x x x x x x x x   x 0 0 0 0 0 0 0

PIC 9(2) bit positions:  0 1 2 3 4 5 6 7   0 1 2 3 4 5 6 7
resulting 9(2) data:     0 0 0 0 0 0 0 x   x x x x x x x x
```

The x represents a data bit from the Message Section item and the 0 is a binary zero fill bit.

On output, in the above example, the data is fetched from where the x variables appear. This example illustrates the rule of left to right processing from PIC X(n) fields and right to left processing from PIC 9(n) fields.

PRESENT IF Clause

The PRESENT IF clause defines a field within a message to be optionally present based on the value of a control field.

- The control field and the optionally present field must be in the same 01 message structure.
- The control field must precede the optionally present field. The value of the control field determines the presence or absence of the optionally present field.

You can use the PRESENT IF clause only with a message format that allows for fields within a message to be optionally present based upon the value of a previously processed message field.

The PRESENT IF clause allows you to send or receive only the data that is actually present, thus reducing the amount of data that must be transmitted.

```
PRESENT IF [ NOT ] control-field
```

control-field

is a field in the Message Section that describes the presence or absence of the associated optionally present field.

A TRUE condition is produced if *control-field* is a numeric data item that contains a value other than zero or if *control-field* is a nonnumeric data item that contains a character other than blank. When *control-field* is TRUE, the optionally present field is considered to be present.

A FALSE condition is produced if *control-field* is a numeric data item that contains a zero or if *control-field* is a nonnumeric data item that contains a blank. When *control-field* is FALSE, the optionally present field does not exist.

NOT

causes the initial Boolean value for *control-field* to be complemented.

The following rules apply:

- The PRESENT IF clause can be used only with DELIMITED messages. Specification of PRESENT IF with other message formats will result in a syntax error. The DELIMITED message format allows you to process variable-length fields for the data, thus, conserving on data transmission.
- The PRESENT IF clause can be specified at the Message Section group level or at the field level. When the PRESENT IF clause is specified at the group level, it causes the condition to be propagated to the fields that belong to this group. These

fields can still specify their own PRESENT IF condition. The group level PRESENT IF condition is overridden by the field level PRESENT IF condition.

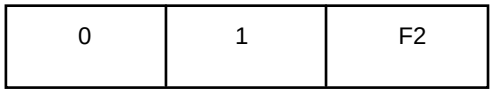
- PRESENT IF clause processing is the same for input and output operations.
- PRESENT IF clauses can be logically nested to any depth. When PRESENT IF clauses are nested, an optionally present field is present only if a second optionally present field is present; the second optionally present field is present only if a third optionally present field is present, and so forth.
- The first level has to be present before you can use the second level, the second level has to be present before you can use the third level, and so forth.
- To conveniently determine if a conditionally present field is present, specify a FIELD STATUS clause on the field. You can test the shadow data item in the FIELD STATUS structure to determine the presence of the field. Refer to the FIELD STATUS clause for details.
- An alternative, although less desirable, method of testing for the presence of a field is to make explicit reference to the Boolean values of the Working-Storage data item associated with the control-field parameter in the PRESENT IF clause.
- You must use caution when setting up the data structures to use the PRESENT IF clause with the DELIMITED message format. It is possible for one data field to be substituted for another in an input message without any possibility of detection of the unintended substitution by the TCP.

The following example shows where such a substitution could take place.

```
WORKING-STORAGE SECTION.
01  WS-MSG.
    02  WS-PIF-1     PIC 9(1) COMP.
    02  WS-PIF-2     PIC 9(1) COMP.
    02  WS-F1        PIC X(8).
    02  WS-F2        PIC X(8).

MESSAGE SECTION.
01  MSG   MESSAGE FORMAT IS DELIMITED
           FIELD-DELIMITER IS OFF.
    02  PIF-1        PIC 1(1)   TO  WS-PIF-1.
    02  PIF-2        PIC 1(1)   TO  WS-PIF-2.
    02  F1           PIC X(8)
                      PRESENT IF PIF-1
                      TO WS-F1.
    02  F2           PIC X(8)
                      PRESENT IF PIF-2
                      TO WS-F2.
```

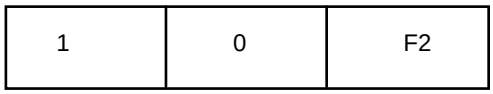
The following input record would result in the value F2 being stored in field WS-F2:



CDT 011CDD

- 0 is stored in WS-PIF-1
- 1 is stored in WS-PIF-2
- F2 is stored in WS-F2

However, suppose that the 0 and 1 were unintentionally reversed in the input record.



CDT 012CDD

- 1 is stored in WS-PIF-1
- 0 is stored in WS-PIF-2
- F2 is stored in WS-F1

In this case, F2 would be stored in field WS-F1, and no indication of the switched data would be given by the TCP. Furthermore, there is no way that such an indication could be given because no TCP rules are broken in either case.

Example of PRESENT IF Clause

You might use the PRESENT IF clause to process the data that is sent by an automated teller machine (ATM). Assume that the data from this ATM is being sent in a bit map message format.

Consider the data that must get transferred based on the ATM buttons that are pushed.

A customer wants to make a deposit into a savings account. The customer also wants to transfer money from a checking account to a credit card account.

The data could be represented by a bit map followed by a stream of data. On the communication line this data would look like a stream of ones and zeros with each bit or group of bits representing a piece of data. The following example shows how a bit map might represent the credit card, savings account, and checking account numbers.

```
Bit Map                 | 1st data | 2nd data | 3rd data
                        | byte     | byte     | byte
                        |          |          |
0  1  1  1  1  0  1  1  | 10110110 | 01101111 | ...
(1) (2) (3)
```

(1) Checking Account Number

(2) Savings Account Number

(3) Credit Card Number

As the customer presses the appropriate buttons on the ATM to conduct the transaction, certain pieces of data will be transferred and others will not. For the pieces of data that are transferred, there will be a 1 in the corresponding bit map. For the pieces that are not transferred, there will be a corresponding zero.

When the customer makes the deposit to the savings account, the first three bits in the above bit map would look like the following:

```
0 1 0
```

and the first data item following the bit map will be the savings account number (the checking account number is not present). When the customer makes the transfer from the checking to the credit card account, the first three bits of the above bit map would look like the following:

```
1 0 1
```

and the first data item following the bit map will be the checking account number. The second data item will be the credit card number.

To process the data, all of the data could be read from the ATM into Working-Storage without regard to message format. The data could then be formatted as it is mapped through the Message Section with the TRANSFORM statement. You would design the Message Section to match the message format, which has a bit map preceding the data. For each of the conditionally present fields within the message, you would include a PRESENT IF clause and a FIELD STATUS clause in the Message Section to check for the presence or absence of the field. If the bit in the bit map is on, the corresponding field is mapped through the message template. If the bit in the bit map is off, there is no corresponding field and the next field in the template is examined.

The code for the Working-Storage Section and the Message Section might look like the following.

```
WORKING-STORAGE SECTION.

01  WS-Input-Output-Record.
    05  WS-Record-Length           PIC 9(4) COMP.
    05  WS-Record                  PIC X(1)
                                   OCCURS 1 TO 100 TIMES
                                   DEPENDING ON
                                      WS-Record-Length.
```

```
01  WS-ATM-Record.
    05  WS-Bit-Map.
        10  WS-Bit-1                PIC 9(1) COMP.
        10  WS-Bit-2                PIC 9(1) COMP.
        10  WS-Bit-3                PIC 9(1) COMP.

01  WS-Savings-Acct.
    05  WS-Savings-Length           PIC 9(2) COMP.
    05  WS-Savings-Data             PIC X(1)
                                    OCCURS 1 TO 15 TIMES
                                    DEPENDING ON
                                       WS-Savings-Length.

01  WS-Checking-Acct.
    05  WS-Checking-Length          PIC 9(2) COMP.
    05  WS-Checking-Data            PIC X(1)
                                    OCCURS 1 TO 15 TIMES
                                    DEPENDING ON
                                       WS-Checking-Length.
01  WS-Credit-Card.
    05  WS-Creditc-Length           PIC 9(2) COMP.
    05  WS-Creditc-Data             PIC X(1)
                                    OCCURS 1 TO 15 TIMES
                                    DEPENDING ON
                                       WS-Creditc-Length.

01  WS-Field-Status-Items.
    05  WS-Field-Status-Savings.
        10 FS-Shadow-Savings        PIC 9(4) COMP.
        10 FS-Error-Savings         PIC 9(4) COMP.

    05  WS-Field-Status-Checking.
        10 FS-Shadow-Checking       PIC 9(4) COMP.
        10 FS-Error-Checking        PIC 9(4) COMP.
    05  WS-Field-Status-Credit-Card.
        10 FS-Shadow-Credit-Card    PIC 9(4) COMP.
        10 FS-Error-Credit-Card       PIC 9(4) COMP.

MESSAGE SECTION.

01  MS-Input-Output-Record.
        MESSAGE FORMAT IS DELIMITED
        FIELD-DELIMITER IS OFF.
    05  MS-Record-Length            PIC X(100)
                                    USING WS-Record
                                    RESULTING COUNT IS
                                       WS-Record-Length.

01  MS-ATM-Record
        MESSAGE FORMAT IS DELIMITED
        FIELD-DELIMITER IS OFF.
    05  MS-Bit-Map.
           10 MS-Bit1-Savings       PIC 1(1)
                                    USING WS-Bit-1.
           10 MS-Bit2-Checking      PIC 1(1)
                                    USING WS-Bit-2.
```

```
        10 MS-Bit3-Credit-Card  PIC 1(1)
                                USING WS-Bit-3.
        10 FILLER               PIC 1(5).
05 MS-Savings.
        10 Savings-Data         PIC X(15)
             PRESENT IF MS-Bit1-Savings
             FIELD STATUS IS WS-Field-Status-Savings
             USING WS-Savings-Data
             RESULTING COUNT IS WS-Savings-Length.
05 MS-Checking.
        10 Checking-Data        PIC X(15)
             PRESENT IF MS-Bit2-Checking
             FIELD STATUS IS WS-Field-Status-Checking
             USING WS-Checking-Data
             RESULTING COUNT IS WS-Checking-Length.
05 MS-Credit-Card.
        10 Credit-Card-Data     PIC X(15)
             PRESENT IF MS-Bit3-Checking
             FIELD STATUS IS WS-Field-Status-Credit-Card
             USING WS-Creditc-Data
              RESULTING COUNT IS WS-Creditc-Length.
```

RESULTING COUNT Clause

On input, the RESULTING COUNT clause obtains the length in bytes of a field or message. For inputting data, you can use the RESULTING COUNT clause with any message format.

On output, the RESULTING COUNT clause creates a variable-length field or message. This length is equal to or less than the declared length. For outputting data, you can use the RESULTING COUNT clause only with the DELIMITED, VARYING1, or VARYING2 message formats.

You must use the RESULTING COUNT clause with the FIELD-DELIMITER clause or with the MESSAGE-DELIMITER clause.

```
[ RESULTING ] COUNT [ IS ] numeric-data-1
```

numeric-data-1

is a numeric elementary data item in either the Working-Storage Section or the Linkage Section.

Resulting Count With Field Delimiter

On input, the RESULTING COUNT clause combined with a field that is delimited causes the field's actual length to be stored in the RESULTING COUNT data item, *numeric-data-1*. The field's contents, of RESULTING COUNT length, is stored into the field's associated Working-Storage data item. The Working-Storage data item is padded with fill characters if it is larger than the actual message field. The field delimiter is not stored in the field's associated Working-Storage data item. The field delimiter is not counted in the RESULTING COUNT length.

On output, the RESULTING COUNT clause combined with the field delimiter permits the length of each field to be equal to or less than the declared field length. A field of RESULTING COUNT length terminated by a field delimiter is created, unless field delimiting has been disabled. When formatting a message of DELIMITED format for output to the device, you can control the length of the overall message by using the RESULTING COUNT clause on a field-by-field basis.

You do not reserve a position for the field delimiter in the Message Section description for the delimited field.

On output of a delimited field, the TCP assembles the field by taking the number of bytes specified by the RESULTING COUNT length from the Working-Storage FROM/USING field and placing those bytes into the message. The TCP will then append the field delimiter, if any, to the field. If the RESULTING COUNT length is 0, only the field delimiter will be output in the message. If the RESULTING COUNT length exceeds the maximum number of bytes defined for the field, an ON ERROR clause will be executed.

Minimum occurrence detection is not enforced. That is, if x > 0 and the delimiter is found before enough data items are processed to satisfy x, an error will not be generated.

Resulting Count With Message Delimiter

For input, the RESULTING COUNT clause combined with the MESSAGE-DELIMITER clause allows you to determine the length of a variable-length message. This is for information only; it does not control the length of the message.

For input, the RESULTING COUNT length at the message level includes the field delimiters.

For output, the only time you can use the RESULTING COUNT clause on the message level is to send a one-field message. With a one-field message, the RESULTING COUNT clause allows you to send a message that is the actual length of the data and remove any padding characters. To output multifield messages, use the RESULTING COUNT clause on each field of the message. This causes the message to be compacted on a field-by-field basis.

Example of Resulting Count on Field Level

Assume that an input data stream consists of from 1 to 101 characters. The last character in the field is a semicolon (;) and acts as a field delimiter. There is, therefore, a maximum of 100 data characters plus one delimiting character.

The following code segment shows the flow of the data on an input operation.

```
WORKING-STORAGE SECTION.
01   WS-Name-Record.
     05 WS-Name-Length     PIC 9(4) COMP.
     05 WS-Name            PIC X(1)
                           OCCURS 1 TO 100 TIMES
                           DEPENDING ON WS-Name-Length.
```

```
MESSAGE SECTION.
01   MSG-Name-Record
         FORMAT IS DELIMITED
         FIELD-DELIMITER IS ";".
     05   MS-Name PIC X(100)
                  USING WS-Name
                  RESULTING COUNT IS WS-Name-Length.
```

If the incoming name field contains:

```
                SMITH John X.;
Start of Field↙
```

The resulting location in Working-StorageWorking-Storage, WS-Name, would contain:

```
                SMITH John X.
Start of Field↙
```

The value in WS-Name-Length would be equal to 13.

TO/FROM/USING Clauses

The TO, FROM, and USING clauses associate a Working-Storage Section or Linkage Section data item with a Message Section item. This association allows the data to be mapped through the Message Section as it is moved into or out of Working-Storage by statements such as: SEND MESSAGE, TRANSFORM, RECEIVE UNSOLICITED MESSAGE, or REPLY TO UNSOLICITED MESSAGE.

```
{ TO    } data-name-1
{ FROM  }
{ USING }
```

data-name-1

identifies the Working-Storage or Linkage Section data area from which message data is received or to which message data is moved. The *data-name-1* item must be described as either alphanumeric (PIC X) or as binary (PIC 9 COMP or PIC S9 COMP); it must not be described as numeric (PIC 9).

TO

specifies that data is to be moved from the message field to the area indicated by *data-name-1*.

FROM

specifies that data is to be moved from the area indicated by *data-name-1* to the message field.

USING

specifies that data is to be moved either from the message field to the *data-name-1* data area or from the *data-name-1* data area to the message field, or both.

A field can be described with both a TO and a FROM clause. If you specify both clauses, each can have a different data name. The particular clause determines the general type of the field (input, output, or input-output) as indicated in Table 5-13.

Table 5-13. Association Clauses and Message-Field Types

Association Clause	Message-Field Type
TO clause only	Input
FROM clause only	Output
USING clause	Input-Output
TO and FROM clause	Input-Output

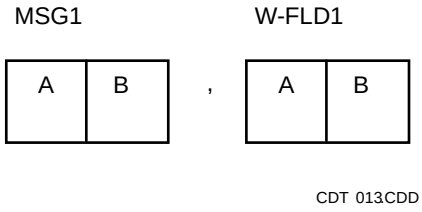
Assume the following Working-Storage and Message Sections:

```
WORKING-STORAGE SECTION.
01     W-FLD1     PIC X(2).
01     W-FLD2     PIC X(2).
01     W-FLD3     PIC 9(18)   COMP.
01     W-FLD4     PIC X(4).

MESSAGE SECTION.
01     MSG1     PIC X(2)   TO W-FLD1.
01     MSG2     PIC X(5)   FROM W-FLD2  MESSAGE FORMAT IS VARYING1.
01     MSG3     PIC X(8)   USING W-FLD3  MESSAGE FORMAT IS VARYING2.
01     MSG4      PIC X(4)  FROM W-FLD1 TO W-FLD4
                           MESSAGE FORMAT IS VARYING1.
```

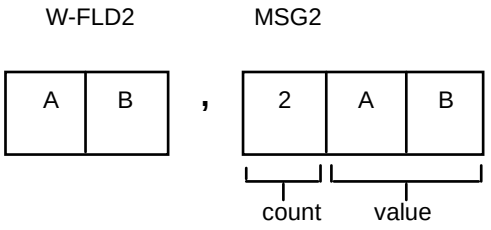
Notice that W-FLD3 contains eight characters, the amount required by a COMP field whose size is between 9(10) and 9(18).

1. The fixed format field MSG1 receives a message containing the two characters AB. These characters are then passed to W-FLD1:



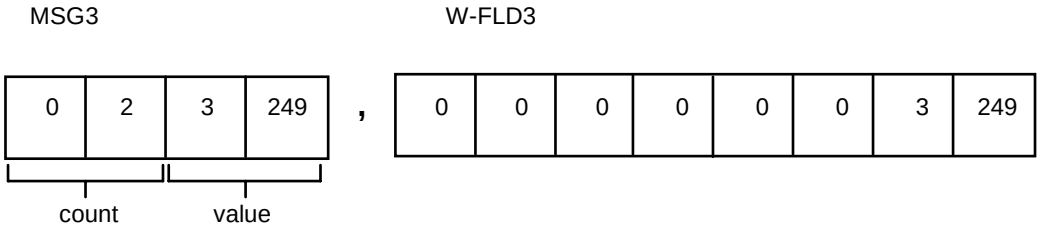
CDT 013.CDD

2. The variable-length field MSG2 contains an output message that originated in W-FLD2. The first byte of MSG2 contains the number of data characters in the message:



CDT 014.CDD

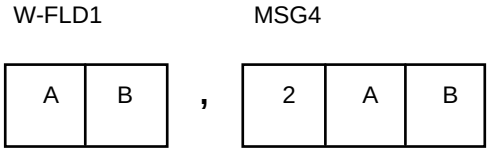
3. The VARYING2 field MSG3 receives a message and passes it to W-FLD3. The first two bytes of MSG3 specify that the message has two characters of data:



CDT 015.CDD

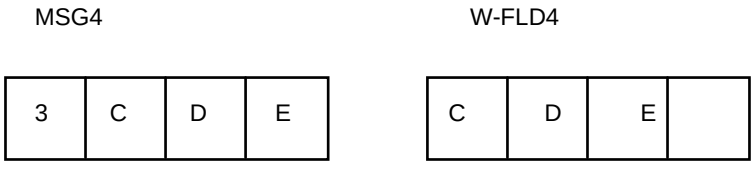
Since W-FLD3 is an eight-byte binary item, the message is right justified in the field and filled with zeros.

4. The VARYING1 field MSG4 sends a message containing two characters of data from W-FLD1:



CDT 016.CDD

MSG4 receives a three-character reply, which it places in W-FLD4:



CDT 017.CDD

Since W-FLD4 is an alphanumeric item, the reply is left justified in the field and filled with blanks.

USER CONVERSION Clause

The USER CONVERSION clause assigns a user-defined number to be passed with the field to a conversion procedure.

```
USER [ CONVERSION ] numeric-literal
```

numeric-literal

specifies the user conversion routine number.

This clause is used only if the application makes use of a user conversion procedure. Refer to the *Compaq NonStop™ Pathway/iTS TCP and Terminal Programming Guide* for details regarding user conversion procedures.

User conversion procedures are provided for input and for output. An input procedure converts incoming data; an output procedure converts outgoing data. The TCP does no standard conversion of data passed between an intelligent device and a message field in the Message Section.

Special Registers

Special registers are data items defined automatically by the SCREEN COBOL compiler, not by the program. Each special register has a particular purpose and should be used only in the manner outlined in its description.

DIAGNOSTIC-ALLOWED Special Register

The DIAGNOSTIC-ALLOWED special register indicates whether diagnostic screens are to be displayed to inform the terminal operator if an error or termination condition occurs. A copy of this register is local to each program unit.

The register is initialized to the value specified by the DIAGNOSTIC parameter of the PATHCOM SET TERM command each time the program unit is called. If the DIAGNOSTIC parameter has not been specified on the SET TERM command, the default value is YES, which enables display of diagnostic screens. The program can move the value NO into the register to disable display of diagnostic screens.

The register has the following implicit declaration:

```
01  DIAGNOSTIC-ALLOWED    PIC AAA.
```

For additional information regarding the use of diagnostic screens, refer to Section 6, Procedure Division and Appendix A, Advisory Messages.

LOGICAL-TERMINAL-NAME Special Register

The LOGICAL-TERMINAL-NAME special register contains the name of the terminal executing the program unit. The name of the terminal is defined through PATHCOM in the ADD TERM command. A single copy of this register is global to the program units. The register is initialized when the terminal is first started.

The register has the following implicit declaration:

```
01  LOGICAL-TERMINAL-NAME    PIC X(16).
```

NEW-CURSOR Special Register

The NEW-CURSOR special register controls placement of the cursor in the next accept operation. A single copy of this register is global to the program units.

If the register value is not a valid screen position when an accept operation begins, the cursor is positioned to the first field of the ACCEPT statement. At the end of any accept operation, the register is set to zero; this causes the default position for the next accept operation to be the first field of that ACCEPT statement.

The register has the following implicit declaration:

```
01  NEW-CURSOR.
    02  NEW-CURSOR-ROW    PIC 9999 COMP.
    02  NEW-CURSOR-COL    PIC 9999 COMP.
```

OLD-CURSOR Special Register

The OLD-CURSOR special register indicates the row and column occupied by the cursor at the last accept operation. A single copy of this register is global to the program units. The register is set by each ACCEPT statement executed by a program unit; the program unit can subsequently access the register.

The register has the following implicit declaration:

```
01  OLD-CURSOR.
    02  OLD-CURSOR-ROW   PIC 9999 COMP.
    02  OLD-CURSOR-COL   PIC 9999 COMP.
```

PW-INPUT-FIELDS-MISSING Special Register

The PW-INPUT-FIELDS-MISSING special register indicates whether the TCP found a field to be absent as the field was mapped through the Message Section on input. You would use this register to determine when special processing is needed to handle absent field situations.

You might encounter absent fields when processing messages of delimited format or messages that contain conditionally present fields. The TCP considers a field to be absent when:

- The message specifies that field delimiters are enabled and optionally that message delimiters are enabled. The TCP finds:
  - Two consecutive delimiters: two field delimiters or a field delimiter followed by a message delimiter
  - A field delimiter as the last character of the message and the associated field as the last field of the message
- The message specifies that a field is conditionally present and the condition is not met so that the TCP finds the field to be absent.

This register is relevant with any statement that uses the Message Section on inbound data, such as: RECEIVE UNSOLICITED MESSAGE, SEND MESSAGE, TRANSFORM. The register contains a YES if one or more fields are absent; otherwise, the register contains a NO. The register is initialized to NO at program startup time. Inbound messages containing absent fields can cause the register's value to change. You can test the register at any place in the program. Testing is usually most meaningful just after an operation that could cause the register to change.

This is a read-only register. Programs attempting to modify this special register will be flagged at compilation with the message:

```
** ERROR 454 ** READ-ONLY SPECIAL REGISTER;
                MAY NOT BE ALTERED
```

The register has the following implicit declaration:

```
01  PW-INPUT-FIELDS-MISSING   PIC AAA.
```

PW-QUEUE-FKEY-UMP Special Register

The PW-QUEUE-FKEY-UMP special register controls the behavior of the ACCEPT...ESCAPE ON UNSOLICITED MESSAGE statement. This register can be altered freely throughout the SCREEN COBOL application and is global to all program units.

The register has the following implicit declaration:

```
01  PW-QUEUE-FKEY-UMP   PIC AAA.
```

The value of this register is either YES or NO. The default value for the register is YES.

If the PW-QUEUE-FKEY-UMP special register equals YES and an unsolicited message arrives during an ACCEPT...ESCAPE ON UNSOLICITED MESSAGE statement, the keyboard remains unlocked during execution of the unsolicited message logic. If the operator presses a function key during this time, the function key value is internally queued in the 6530 terminal. The function key value is read upon execution of the next ACCEPT statement.

If the PW-QUEUE-FKEY-UMP special register equals NO and an unsolicited message arrives during an ACCEPT...ESCAPE ON UNSOLICITED MESSAGE statement, the TCP issues a command to the terminal to lock the keyboard before executing the

unsolicited message logic. The keyboard is unlocked at the next ACCEPT statement. This action prevents 6530 terminals from internally queuing a function key that the operator presses between the start of execution of the unsolicited message logic and the next ACCEPT statement.

PW-QUEUE-FKEY-TIMEOUT Special Register

The PW-QUEUE-FKEY-TIMEOUT special register controls the behavior of the ACCEPT...ESCAPE ON TIMEOUT statement. This register can be altered freely throughout the SCREEN COBOL application and is global to all program units.

The register has the following implicit declaration:

```
01  PW-QUEUE-FKEY-TIMEOUT   PIC AAA.
```

The value of this register is either YES or NO. The default value for the register is YES.

If the PW-QUEUE-FKEY-TIMEOUT special register equals YES and a timeout occurs on an ACCEPT...ESCAPE ON TIMEOUT statement, the keyboard remains unlocked during execution of the timeout logic. If the operator presses a function key during this time, the function key value is internally queued in the 6530 terminal. The key value is read upon execution of the next ACCEPT statement.

If the PW-QUEUE-FKEY-TIMEOUT special register equals NO and a timeout occurs on an ACCEPT...ESCAPE ON TIMEOUT statement, the TCP issues a command to the terminal to lock the keyboard before executing the timeout logic. The keyboard is unlocked at the next ACCEPT statement. This action prevents 6530 terminals from internally queuing a function key that the operator presses between the start of execution of the timeout logic and the next ACCEPT statement.

PW-TCP-PROCESS-NAME and PW-TCP-SYSTEM-NAME Special Registers

The PW-TCP-PROCESS-NAME and PW-TCP-SYSTEM-NAME special registers contain the TCP Guardian process name and the host system name, respectively.

The SCREEN COBOL program uses these registers, in conjunction with the TERMINAL-FILENAME special register, to identify itself to another Guardian process. The Guardian process must be a member of an active Pathway server class.

The SCREEN COBOL program can identify itself to the Guardian process by sending the TCP process-name and system-name, and the SCREEN COBOL program name. The Guardian process can then use the values in an UMP header and communicate with the identified SCREEN COBOL program.

Programs attempting to modify either of these special registers will be flagged at compilation with the message:

```
** ERROR 454 **  READ-ONLY SPECIAL REGISTER;
                  MAY NOT BE ALTERED
```

The PW-TCP-PROCESS-NAME and PW-TCP-SYSTEM-NAME special registers have the following implicit declarations; note that the VALUE clauses are not necessary and are for illustration only.

```
01  PW-TCP-SYSTEM-NAME       PIC X(8) VALUE "\STLOUIS".
01  PW-TCP-PROCESS-NAME      PIC X(6) VALUE "$SWTCP".
```

PW-TERMINAL- ERROR-OCCURRED Special Register

The PW-TERMINAL-ERROR-OCCURRED special register contains an error number when an irrecoverable terminal I/O error occurs and the SCREEN COBOL program contains a USE FOR TERMINAL-ERRORS statement.

The register has the following implicit declaration:

```
01  PW-TERMINAL-ERROR-OCCURRED   PIC 9999 COMP.
```

PW-UNSOLICITED-MESSAGE-QUEUED Special Register

The PW-UNSOLICITED-MESSAGE-QUEUED special register indicates whether any unsolicited messages are currently queued for the SCREEN COBOL program.

The SCREEN COBOL program can test the value of this register or move the value to another field at any time during execution. This register can be used to periodically test for the presence of an unsolicited message as an alternative to issuing an ACCEPT or SEND MESSAGE with the ESCAPE clause.

The value of this register is either YES or NO.

The register has the following implicit declaration:

```
01 PW-UNSOLICITED-MESSAGE-QUEUED   PIC AAA.
```

Programs attempting to modify this special register will be flagged at compilation with the message:

```
** ERROR 454 **  READ-ONLY SPECIAL REGISTER;
                 MAY NOT BE ALTERED
```

The following is an example of how to use this register:

```
IF PW-UNSOLICITED-MESSAGE-QUEUED EQUALS "YES"
   PERFORM process-message
     UNTIL PW-UNSOLICITED-MESSAGE-QUEUED EQUALS "NO".
```

PW-USE-NEW-CURSOR Special Register

The PW-USE-NEW-CURSOR special register preserves the current cursor position following unsolicited message processing.

When a block-mode ACCEPT operation is interrupted by the arrival of an unsolicited message, the user might typically be typing data on the current screen. The SCREEN COBOL program then processes the unsolicited message. The SCREEN COBOL program reissues the interrupted ACCEPT operation.

Usually reissuing the ACCEPT operation allows the cursor to be repositioned as specified in the NEW-CURSOR special register. This would take place if PW-USE-NEW-CURSOR contained the value YES.

If, before reissuing the ACCEPT operation, the SCREEN COBOL program places the value NO in PW-USE-NEW-CURSOR, the screen cursor will be left where the user has positioned it. This prevents the loss of cursor position following unsolicited message processing that involves either no terminal I/O or terminal output only.

The PW-USE-NEW-CURSOR special register is reset to its default value of YES following completion of an ACCEPT, DISPLAY BASE, or CALL operation.

The register has the following implicit declaration and default value:

```
01  PW-USE-NEW-CURSOR        PIC A(3) VALUE "YES".
```

REDISPLAY Special Register

The REDISPLAY special register can prevent unnecessary moving of screen field data to terminal memory. This register indicates to the TCP whether screen field data must be sent to the terminal during processing of a DISPLAY statement or redisplayed from data already in terminal memory. The REDISPLAY register affects data transmission only for the DISPLAY statement and only for areas defined in a base screen but not in an overlay area.

The REDISPLAY register supports T16-6520, T16-6530, and T16-6540 terminals operating in block mode. This register does not support other types of terminals or terminals operating in conversational mode, and it does not support intelligent devices.

A single copy of the REDISPLAY register is global to the SCREEN COBOL program units. The register is set to NO (disables redisplay) when the terminal is first started. The SCREEN COBOL program can move YES (requests redisplay) into the register for specific screens.

For redisplay processing to apply to a DISPLAY statement, these requirements must be met:

- The REDISPLAY register must be set to YES at the time the DISPLAY BASE statement for the current base screen executes. This enables redisplay processing for the screen.
- The REDISPLAY register must be set to YES at the time the affected DISPLAY statement executes.

- The screen field items referred to in the DISPLAY statement must be defined in the base screen and not in an overlay area.

When the REDISPLAY register is set to NO (the default setting), a DISPLAY statement results in the TCP moving the contents of the screen fields to terminal memory. The following events occur:

- The TCP obtains screen field values from associated Working-Storage items and moves the values to terminal memory.
- TCP resets the modified data tags for all screen fields.

When the REDISPLAY register is set to YES and redisplay is enabled for the screen, a DISPLAY statement usually results in redisplaying the screen field data held in terminal memory. The following events occur:

- Data already in terminal memory is used for the display. (The TCP moves no values from Working-Storage items.)
- TCP resets the modified data tags for all screen fields.

If the REDISPLAY register is set to YES and the DISPLAY statement is the first DISPLAY after a DISPLAY BASE statement during which the screen was not in terminal memory, the TCP moves all screen field data to terminal memory. The following events occur in this situation:

- The TCP obtains screen field values from associated Working-Storage items and moves the values to terminal memory.
- All modified data tags for screen fields are reset.

The register has the following implicit declaration:

```
01  REDISPLAY    PIC AAA.
```

An example of the REDISPLAY register is:

```
PROCEDURE DIVISION.
    :
    MOVE "YES" TO REDISPLAY.
    DISPLAY BASE EMPLOYEE-REC-SCREEN.
    DISPLAY      EMPLOYEE-REC-SCREEN.
    :
```

In this example, the REDISPLAY register is set to YES before the DISPLAY BASE statement executes to enable REDISPLAY for the screen. After the DISPLAY BASE statement executes, the REDISPLAY register can be reset as desired for subsequent execution of the DISPLAY screen statement.

RESTART-COUNTER Special Register

The RESTART-COUNTER special register contains the number of times a transaction has been restarted during transaction mode. The first time the BEGIN-TRANSACTION statement executes, the register is set to zero. This number is incremented immediately following each execution of the BEGIN-TRANSACTION statement.

The register has the following implicit declaration:

```
01  RESTART-COUNTER    PIC 9999 COMP.
```

STOP-MODE Special Register

The STOP-MODE special register can prevent interruption of multiple step transactions. A single copy of this register is global to the program units.

The register is set to zero when the terminal is first started, and the value is subsequently under program control. Most programs will continue with a value of zero.

When the value is nonzero, several PATHCOM commands are affected. The effect of the STOP TERM, SUSPEND TERM, and FREEZE SERVER commands is delayed until the register value returns to zero. The SUSPEND and FREEZE commands can be issued in a form that causes the STOP-MODE value to be disregarded.

The register has the following implicit declaration:

```
01  STOP-MODE    PIC 9999 COMP.
```

TELL-ALLOWED Special Register

The TELL-ALLOWED special register can be set by the program to control the issuing of tell messages to a terminal during ACCEPT statement processing. It has no meaning for programs communicating with intelligent devices.

A copy of this register is available to each program unit. The register is initialized to YES each time the program unit is called. The program can move NO into the register to prevent tell messages from being displayed during succeeding accept operations.

When this register is set to YES and a tell message is waiting, the following occurs:

- When the TCP is about to complete an accept operation, it displays the tell message (prefixed by the word MESSAGE:) in the ADVISORY field.
- The TCP waits for any function key from the terminal operator, then resets the field and completes the accept operation.

When this register is set to NO, display of the tell message is postponed.

The register has the following implicit declaration:

```
01  TELL-ALLOWED    PIC AAA.
```

TERMINAL-FILENAME Special Register

The TERMINAL-FILENAME special register contains the internal form of the file name for the terminal executing the program unit. The name of the terminal is defined through PATHCOM in the SET TERM FILE parameter.

A single copy of this register is global to the program units. The register is initialized when the terminal is first started.

The register has the following implicit declaration:

```
01  TERMINAL-FILENAME    PIC X(24).
```

Note. Because the TERMINAL-FILENAME special register is in internal form, it should not be displayed on a terminal screen without first being converted to external form (which should be done in a server). Displaying TERMINAL-FILENAME directly on a terminal screen might cause unpredictable results and cause the terminal to return an I/O error.

TERMINAL-PRINTER Special Register

The TERMINAL-PRINTER special register contains the external form of the file name for the printer that is associated with the terminal executing the program unit. If no associated printer is defined in PATHCOM, this register contains blanks. A single copy of this register is global to the program units. The register is initialized when the terminal is first started.

The register has the following implicit declaration:

```
01  TERMINAL-PRINTER    PIC X(36).
```

TERMINATION-STATUS Special Register

The TERMINATION-STATUS special register communicates the completion status of the following statements: ACCEPT, BEGIN-TRANSACTION, RECEIVE UNSOLICITED MESSAGE, SEND, or SEND MESSAGE. A copy of this register is available to each program unit. The register is initialized to zero each time the program unit is called.

This special register also communicates an error number when the ON ERROR branch of one of the following statements is taken: CALL, RECEIVE UNSOLICITED MESSAGE, REPLY TO UNSOLICITED MESSAGE, SEND, SEND MESSAGE, or TRANSFORM.

The register has the following implicit declaration:

```
01  TERMINATION-STATUS    PIC 9999 COMP.
```

TERMINATION-SUBSTATUS Special Register

The TERMINATION-SUBSTATUS special register communicates an error number further describing the error communicated in TERMINATION-STATUS when the ON ERROR branch of one of the following statements is taken: CALL, RECEIVE UNSOLICITED MESSAGE, REPLY TO UNSOLICITED MESSAGE, SEND, SEND MESSAGE, or TRANSFORM. A copy of this register is local to each program unit.

The register has the following implicit declaration:

```
01  TERMINATION-SUBSTATUS    PIC 9(5) COMP.
```

TRANSACTION-ID Special Register

The TRANSACTION-ID special register contains the value of the transaction identifier that the Compaq Transaction Management Facility (*TMF*) assigns when the BEGIN-TRANSACTION statement executes. TMF assigns a unique identifier to this register for each new or restarted transaction. The register is set to SPACES after either the END-TRANSACTION or the ABORT-TRANSACTION statement executes.

Generally, the contents of this special register should not be displayed on a terminal screen because the associated data item contains binary data. You use this register to locate uniquely identified transactions.

The register has the following implicit declaration:

```
01  TRANSACTION-ID    PIC X(8).
```

6 Procedure Division

The Procedure Division includes all of the processing steps for the program. The steps are organized into SCREEN COBOL statements and sentences, and grouped into paragraphs, procedures, and sections.

The format of the Procedure Division is:

```
PROCEDURE DIVISION [ USING data-name [, data-name ]...] .

[ DECLARATIVES.

{ [ section-name SECTION . ]

   [ paragraph-name . [ sentence ] ... ] ... } ...

{ paragraph-name . [ sentence ] ... } ...

END DECLARATIVES. ]


{ [ section-name SECTION . ]

   [ paragraph-name . [ sentence ] ... ] ... } ...

{ paragraph-name . [ sentence ] ... } ...
```

Division Structure

The division begins with a division header. The format of the header is:

```
PROCEDURE DIVISION [ USING data-name [, data-name ]...] .
```

The header must be terminated with a period separator.

The USING phrase is applicable only in a subprogram that is to execute under control of a CALL statement that also contains a USING phrase. The identifiers in the USING phrase must correspond in structure to the identifiers specified in the USING phrase of the CALL statement. The number of identifiers should also correspond; however, an error indication is given by the TCP only when the number of identifiers in the USING phrase exceeds the number of identifiers specified in the USING phrase of the CALL statement.

Execution begins with the first executable statement after the Procedure Division header, excluding any declarative procedures, and continues on in logical order. The Procedure Division header must be immediately followed by the DECLARATIVES keyword and declarative procedures, or immediately followed by a paragraph or section name.

During execution, control is transferred to a paragraph only at the beginning of the paragraph. Control is passed to a sentence within a paragraph only from the

immediately preceding sentence, unless the immediately preceding sentence is a GO TO statement. When control reaches the end of a paragraph, control passes to the first section of the following paragraph. The only exception is when control reaches the end of a paragraph and that paragraph is the last in the range of a currently active PERFORM operation.

An example of Procedure Division structure is:

```
PROCEDURE DIVISION.
initialization SECTION.
get-started.
    sentence
        statement
        statement.
    sentence
        statement
        :
finish-up-init.
    sentence
        :
main-processing SECTION.
begin-it.
    sentence
        :
process-input-data.
        :
end-of-job.
    EXIT PROGRAM.
```

Declarative Procedures

A special portion of the Procedure Division is reserved for declarative procedures. These procedures are screen recovery routines specified by USE statements. When used, this portion must be coded immediately after the Procedure Division header. The portion begins with keyword DECLARATIVES and ends with keywords END DECLARATIVES. The following example illustrates a declarative procedure:

```
PROCEDURE DIVISION.
DECLARATIVES.
RECOV-SECT-1 SECTION.
    USE FOR RECOVERY ...
        :
END DECLARATIVES.
MAIN SECTION.
begin-my-program.
        :
```

Sections

A section, which is optional, is used to group related paragraphs for processing steps. Reference to a section name in a PERFORM statement, for example, includes all paragraphs in that section in the range of the PERFORM.

The format of the header is:

```
section-name SECTION.
```

A section ends at the next section header, at keywords END DECLARATIVES, or at the physical end of the Procedure Division.

Paragraphs

A paragraph is used to group related sentences and statements. A paragraph usually has at least one sentence, but sentences are not required.

For example:

```
get-all-input.

get-the-first-record.
    ACCEPT my-screen...
```

Reference to a paragraph name permits branching from one area of code to another.

A paragraph begins with a paragraph name in Area A. A paragraph ends immediately before the next paragraph name or section name, or at the physical end of the Procedure Division.

Sentences and Statements

A sentence is a string of one or more statements, ending with a period. A statement is a combination of words and symbols beginning with a SCREEN COBOL verb. For example:

```
chk-report-yy.
   IF current-yy IS LESS THAN 0 OR GREATER THAN 99
     DISPLAY "REPORT YEAR IS NOT BETWEEN 00 AND 99, RE-ENTER "
        "YEAR" IN msg-1
     ACCEPT current-yy UNTIL my-file1
     GO TO chk-report-yy.
```

Sentences can be grouped into three functional categories:

- Imperative—Takes an action unconditionally
- Conditional—Takes an action based on a condition
- Compiler directing—Uses compiler-directing verbs COPY or USE

An imperative sentence is constructed from one or more imperative statements terminated by a period. An imperative sentence can have a GO TO statement or an EXIT PROGRAM statement. If an EXIT PROGRAM statement is present, it must be the last statement in the sentence.

The following examples illustrate imperative sentences:

```
ADD a1 TO b1 GIVING c1, d1, e1.

ADD 25 TO x2,
    GO TO next-image.
```

A conditional sentence tests a conditional item or some relationship between values to determine an action to take.

The following example illustrates a conditional sentence:

```
IF last-tax IS LESS THAN current-tax
    PERFORM higher-tax
ELSE PERFORM lower-tax.
```

Procedures

A procedure consists of a paragraph, a group of successive paragraphs, a section, or a group of successive sections. A procedure name is a paragraph or section name; the name can be qualified.

Procedure Division Statements

Procedure Division statements can be grouped into ten categories. Table 6-1 lists each statement and its category.

Note. Intelligent Device Support (IDS) SCREEN COBOL requester programs interact with external (that is, outside of Pathway) processes which, in turn, control intelligent devices such as automated teller machines, airline reservation terminals, and personal computers. These requester programs are message-oriented. IDS requester programs use the SEND MESSAGE statement and its REPLY to send and receive messages.

The following statements cannot be used in IDS requester programs:

```
ACCEPT                DISPLAY          TURN
CLEAR                 PRINT SCREEN     USE FOR SCREEN RECOVERY
DISPLAY BASE          RESET            USE FOR TERMINAL-ERRORS
DISPLAY OVERLAY       SCROLL
DISPLAY RECOVERY      SET
```

All of the SCREEN COBOL statements are described in alphabetic order in the remainder of this section.

Table 6-1. Categories of Statements

Arithmetic	ADD COMPUTE DIVIDE MULTIPLY SUBTRACT
Conditional	BEGIN-TRANSACTION ... ON ERROR CALL ... ON ERROR IF SEND...ON ERROR
Data Movement	ACCEPT DATE/DAY/TIME MOVE SET
Terminal Input/Output	ACCEPT CLEAR DELAY DISPLAY DISPLAY BASE DISPLAY OVERLAY DISPLAY RECOVERY PRINT SCREEN RECONNECT MODEM RESET SCROLL SEND TURN
Intelligent Device Support (IDS)	SEND MESSAGE TRANSFORM
Unsolicited Message Processing (UMP)	RECEIVE UNSOLICITED MESSAGE REPLY TO UNSOLICITED MESSAGE
Interprogram Communication	CALL CHECKPOINT EXIT PROGRAM
Program Control	EXIT GO TO PERFORM STOP RUN
Compiler Directing	COPY USE
Transaction Management	ABORT-TRANSACTION BEGIN-TRANSACTION END-TRANSACTION RESTART-TRANSACTION

ABORT-TRANSACTION Statement

The ABORT-TRANSACTION statement aborts the transaction of a terminal operating in transaction mode. Transaction mode is an operating mode in which Pathway servers that are configured to run under the Compaq Transaction Management Facility (*TMF*) can lock and update audited files. When this statement executes, all database updates that were made to audited files during the transaction are backed out and no attempt is made to restart the transaction.

```
ABORT-TRANSACTION
```

Execution of this statement causes the terminal to leave transaction mode, and the special register TRANSACTION-ID to be set to SPACES.

If the terminal is not in transaction mode when this statement is executed, the terminal is suspended for pending abort.

If a fatal error occurs while the transaction is being aborted, and the current BEGIN-TRANSACTION statement does not have an ON ERROR phrase, the terminal is suspended for pending abort; the current transaction is backed out. If the current BEGIN-TRANSACTION statement includes an ON ERROR phrase, the ON ERROR branch is executed, and the terminal is not suspended.

For additional information about programming for TMF, see the *Compaq NonStop™ Pathway/iTS TCP and Terminal Programming Guide*.

ACCEPT Statement

The ACCEPT statement operates differently for terminals in block mode from terminals in conversational mode. It cannot be used to communicate with intelligent devices.

If the terminal associated with the SCREEN COBOL program is operating in block mode, ACCEPT performs the following:

- Waits for response from the terminal
- Receives data from the terminal
- Returns only valid data to the program, determining the validity of the data from the definitions in the Screen Section of the Data Division
- If invalid data is entered and an ADVISORY field is defined for the base screen, displays an error message and enhances the field in error so the data can be corrected or reentered.

If the terminal associated with the SCREEN COBOL program is operating in conversational mode, ACCEPT performs the following:

- Displays the prompt value defined for the first screen field described with a PROMPT clause. The prompt value is always displayed in the first column of the screen line.
- Waits for response from the terminal. If the TIMEOUT phrase is used, ACCEPT waits the time limit specified in this phrase.

- Receives input from the terminal and stores the data into the associated Working-Storage items of the program data area. Input can be accepted from the terminal one screen field at a time, one field per line. However, the capability referred to as typeahead enables data entry for more than one field on the same line.
- Returns only valid data to the program, determining the validity of the data from the definitions in the Screen Section of the Data Division.
- If invalid data is entered and an ADVISORY field is defined for the base screen, displays an error message and redisplays the prompt for the field in error so the data can be reentered. If no ADVISORY field is defined, it redisplays the prompt but does not display an error message.

```
ACCEPT [ screen-identifier ] ...

{ UNTIL { [()] { comp-condition-1 }... [)] ESCAPE [ ON ] } }
{       {    { [()] { comp-condition-2 }... [)] }        } }
{       {                                                } }
{       { [()] { comp-condition-1 }... [)]               } }
{                                                          }
{ ESCAPE [ ON ] { [()] { comp-condition-2 }... [)] }       }
```

screen-identifier

specifies the screen fields from which data is accepted; ACCEPT statement can have a maximum of 127 screen fields . Each *screen-identifier* can name an entire screen, a screen group, or an elementary input item of any base or overlay screen that is currently displayed. If *screen-identifier* is a group, all subordinate elementary items that have a TO or USING clause in their definition are included in the reference. *Screen-identifier* cannot be a subscripted item.

If data is to be accepted from fields defined for both a base screen and an overlay screen, the *screen-identifier* list must include the overlay screen identifier or the identifiers of items included in the overlay declaration. It is not sufficient to specify the base screen without the overlay screen specification.

The order in which fields appear in the *screen-identifier* list is the order in which they are checked and converted.

If this parameter is omitted, the completion condition in either the UNTIL or ESCAPE clause determines when the statement is to terminate. No data is accepted from the screen, and no Working-Storage item is altered.

In block mode, if a screen contains only filler items and a DISPLAY statement is followed by an ACCEPT statement without a screen identifier, the screen remains until a function key signals the termination of the ACCEPT statement. A typical instance for omitting the identifier would be during the display of a help screen.

UNTIL and ESCAPE

specify the conditions under which the statement is to complete. In block mode, these conditions are typically the names of the terminal function keys that the

terminal operator can use. At least one of these two clauses must be present with a completion condition. If both clauses are present, any one completion condition can appear in only one of the two clauses.

comp-condition-1

specifies the completion conditions under which the statement is to terminate with input of data.

comp-condition-2

specifies the completion conditions under which the statement is to terminate without input of data.

Language elements that can appear as completion conditions (*comp-condition-1* or *comp-condition-2*) are:

ABORT

indicates that the abort input-control characters were entered to terminate the ACCEPT statement. This phrase is effective only for terminals in conversational mode. Refer to the ABORT-INPUT clause described in Section 5.

ABORT is allowed only in the ESCAPE clause. If this phrase is executed, the data items in Working-Storage are not changed.

If the Break key is enabled by Pathway system configuration commands (either the PATHCOM SET PROGRAM TYPE or SET TERM command) for the program unit or for the terminals at which the program unit runs, the ABORT clause must be specified for the Break key to work.

If the terminal is in block mode, ABORT is treated as a comment.

INPUT

indicates that the ACCEPT statement terminates with valid screen input. This phrase is effective only for terminals in conversational mode.

If the terminal is in block mode, INPUT is treated as a comment.

TIMEOUT timeout-value

specifies a time limit in seconds that the terminal operator has to complete the data entry. The *timeout-value* can be a numeric literal or a numeric data item; valid values are 0 through 32,767 seconds. If the operator does not respond in the specified number of seconds, TERMINATION-SUBSTATUS is set to 40, and the ACCEPT operation is cancelled. TIMEOUT can appear only in the ESCAPE clause. The maximum timeout value is the largest value that will fit in a 32-bit field.

If this phrase is not specified, there is no time limit.

Use of the PW-QUEUE-FKEY-TIMEOUT special register allows locking the keyboard before processing the TIMEOUT logic. For further information see Special Registers on page 5-93..

Note. Do not use enclosing parentheses for a function key range following a TIMEOUT clause if a Working-Storage item is used for the TIMEOUT value and the item is not indexed; if you include the parentheses, a SCREEN COBOL compiler error occurs.

`UNSOLICITED [ MESSAGE ]`

indicates that the ACCEPT statement is to be aborted on the arrival of an unsolicited message. You detect the receipt of an unsolicited message by checking for the appropriate condition code value in TERMINATION-STATUS following completion of the ACCEPT statement. UNSOLICITED MESSAGE can appear only in the ESCAPE clause.

Use of the PW-QUEUE-FKEY-UMP special register allows locking the keyboard before processing the unsolicited message logic. For further information see Special Registers on page 5-93.

Note. In conversational mode, all comp-condition phrases except ABORT, INPUT, TIMEOUT, and UNSOLICITED MESSAGE are ignored.

mnemonic-name

indicates the ACCEPT operation completes when the terminal operator presses the associated function key. This assumes *mnemonic-name* has been associated with a terminal function key; the association is specified by an IS phrase in the SPECIAL-NAMES paragraph of the Environment Division. Because of terminal characteristics, certain keys can be used only in the ESCAPE clause. Table 4-1 lists the system names of function keys that can be used only in the ESCAPE clause.

mnemonic-name-1 THROUGH *mnemonic-name-2*

indicates a set of function keys as a single condition. The *mnemonic-names* must be associated with the keys from the same range (shifted or unshifted) of function keys; for example, F2 THROUGH F15.

Use of this option can produce unexpected results if you change function key mapping in the SPECIAL-NAMES paragraph. With function-key remapping, you should list the individual mnemonic names of all keys to be included in the UNTIL and ESCAPE clauses without using the THROUGH or THRU option.

Block Mode Accept Operation

The ACCEPT statement enables the terminal keyboard and waits for input from the terminal. When a valid control key code is received from the terminal, the keyboard is disabled and a RESET TEMP is executed automatically; this causes the removal of any temporary field attributes or data from the display regardless of whether they were originally displayed explicitly by the program or implicitly through the ACCEPT statement. If termination is caused by the completion condition specified in the ESCAPE clause, the ACCEPT statement terminates at this point.

If a prompt is used and the field named in the PROMPT clause is an output field, the ACCEPT statement causes the current value for the output field to be displayed before reading the data input from the terminal. An output field named in the PROMPT clause must be defined as FILLER or defined with a FROM or USING clause.

The data entered from the terminal is checked against the requirements given for the field by its definition in the Screen Section of the Data Division. The TCP checks only those fields referred to by the screen-identifier list in the ACCEPT statement.

If errors are discovered and the terminal is in block mode during the data checking, the following occurs:

- If a field with the ADVISORY clause is defined for the current screen, a DISPLAY TEMPORARY of the advisory field automatically occurs using the standard error message for the first error detected.
- If the terminal is equipped with an audible alarm, the alarm sounds provided it was not suppressed in the SCREEN-CONTROL paragraph of the Environment Division.
- The first field in error has a temporary modification of its display attribute with the standard error enhancement as declared in the SCREEN-CONTROL paragraph. The program can specify that all fields in error are enhanced (refer to the Input-Output Section in Section 4).
- The statement is restarted following these display operations.

If no data errors are found during the checking, the following occurs:

- The validated data from all screen fields referred to and present, including all required fields, is converted and moved into the TO or USING data items in Working-Storage associated with the screen fields.
- Absent screen input fields do not change the associated Working-Storage data items unless specifically requested with the WHEN ABSENT field-characteristic clause.
- All SHADOWED fields associated with the input fields of the ACCEPT statement have their ENTER and RETURN bits set appropriately.

If the completion is through an ESCAPE clause, none of the TO or USING data items is affected. Data variables retain their values, and SHADOWED ENTER and RETURN bits are undefined.

At the end of any accept operation, the NEW-CURSOR special register is set to zero (row 0, column 0). This controls the placement of the cursor for the next accept operation and causes the default position to be the first field of the current ACCEPT statement.

The ACCEPT statement indicates the condition that caused completion by storing the condition code value into the TERMINATION-STATUS special register. Each completion condition is assigned a code value according to its position in the UNTIL or ESCAPE clauses. The codes are assigned by considering the conditions of the UNTIL and ESCAPE clauses to be a single list, and assigning each condition the code value that corresponds to its position in the list. When several conditions are grouped together with parentheses, they are all considered to occupy the same position; that is, all the conditions within the parentheses receive the same code value, and the next condition

following the group receives the code value that is one greater than that assigned to the conditions in the group.

In the following example, the value of TERMINATION-STATUS is 1 if the Enter key is pressed, 2 for the CLEAR key, 2 for the PA1 key, and 3 for the PF1 key.

```
ACCEPT CUSTOMER-SCREEN UNTIL ENTER
    ESCAPE ON (CLEAR, PA1), PF1
```

In the next example, the value of TERMINATION-STATUS is 1 if F1 is pressed, 2 if Shift-F16 is entered, 3 if there is a timeout, and 4 if an unsolicited message is received.

```
GET-OPER-INPUT.
  ACCEPT my-screen
    UNTIL f1-key
          sf16-key
    ESCAPE ON
          TIMEOUT 300
          UNSOLICITED MESSAGE.

  PERFORM ONE OF f1-key-action
                 sf16-key-action
                 timed-out
                 unsol-msg-arrival
  DEPENDING ON TERMINATION-STATUS.
  GO TO get-oper-input.
```

Conversational Mode Accept Operation

The ACCEPT statement displays the prompt value for the first screen field described with a PROMPT clause, enables the keyboard, and waits for data to be entered from the terminal. (If no screen field description contains a PROMPT clause, the ACCEPT statement begins at the first column of the screen.) If termination is caused by a completion condition specified in the ESCAPE clause, the ACCEPT statement terminates at this point with no changes to the Working-Storage data items. The ACCEPT statement always displays the prompt value in the first column of the screen line and positions the cursor at the end of the prompt field regardless of the positions specified for the field in the screen description.

When the terminal is enabled for input, data can be accepted for each input field a line at a time or accepted for more than one field on the same line. If the typeahead capability is used, field or group separators delimit the screen fields such that multiple fields of data are accepted in a single buffer. When typeahead is used, only the prompt value for the first field is displayed. Then, no other prompts appear until the end of the input is indicated by either a carriage return or an input-control character.

The ACCEPT statement processes input data in the order the data is received from the terminal. The input data is associated with the screen fields in the sequence the fields are defined in the Screen Section. The data is accepted until there is no more input, the abort-input character is entered, or an error is detected. The sequence in which the screen identifiers are processed is from top to bottom and from left to right as follows:

1. The screen field with a lower row (line) number is processed before a screen field with a higher row number.

2. Within the same row, the screen field with a lower column number is processed before a screen field with a higher column number.

The input data is checked against the requirements given for a field by the field definition in the Screen Section. Only those fields referred to by the screen-identifier list are checked. During ACCEPT statement processing, the input data is scanned for input-control characters that identify the input fields and indicate an abort, end-of-input, or restart operation. Mnemonic names (except BELL and HIDDEN) are not recognized in conversational mode; therefore, function keys have no effect.

A field error affects only the data in the field that contains the error; fields containing data entered before the error was detected remain valid. Fields containing data entered after the error was detected are ignored.

If an error is discovered during the data checking, the following occurs:

- Only the first field having an error is detected and enhanced. The BELL attribute is the only recognized error enhancement in conversational mode.
- If a field with the ADVISORY clause is defined for the current screen, the advisory field is displayed on the next line following the line with the error.
- ACCEPT processing restarts after the error display operation. The prompt for the field containing the error is redisplayed, and the cursor is positioned to accept the correct input.

Not all errors are detected immediately. If an error is detected after subsequent screen fields have been entered and processed, an error message is displayed and the ACCEPT statement is restarted at the beginning. This is the same action that occurs when a restart-input character is processed.

If no data errors are found during the checking, the following occurs:

- The validated data from each screen field referred to is converted and moved as the field is received from the terminal. The converted data is placed in either the TO or USING data item in Working-Storage associated with the screen field. The characteristics defined for a screen field such as PICTURE, UPSHIFT, and so forth, apply to the converted value.
- Absent screen input fields do not change the associated Working-Storage data items unless specifically requested with the WHEN ABSENT field-characteristic clause.
- All SHADOWED fields associated with the input fields of the ACCEPT statement have their ENTER and RETURN bits set appropriately. If these bits are checked by a comparison statement, the ENTER and RETURN bits should be checked together.

ACCEPT statement processing stores a condition code into the TERMINATION-STATUS special register. A code value is assigned to each completion condition in the same way as described previously for block mode.

The Break key for conversational terminals can be enabled to terminate an ACCEPT operation. The Break key is enabled through the PATHCOM commands SET TERM or SET PROGRAM TYPE in the Pathway system configuration. Also, for the Break key to work on an ACCEPT, the SCREEN COBOL program must include an ESCAPE ON ABORT clause in the ACCEPT statement. If the Break key is enabled and pressed

during an ACCEPT operation, the key has the same effect as entering the abort input-control characters.

The following example illustrates an ACCEPT statement for conversational mode. The value of TERMINATION-STATUS is 1 if valid input is entered, 2 for ABORT, and 2 for TIMEOUT.

```
ACCEPT EMPLOYEE-SCREEN UNTIL INPUT
    ESCAPE ON (ABORT, TIMEOUT 180).
PERFORM ONE OF
    300-CHECK-NULL-NAME
    200-EXIT-ROUTINE
    DEPENDING ON TERMINATION-STATUS.
```

Modified Data Tag (MDT)

There is a modified data tag (MDT) associated with each nonliteral screen field. The MDT is a 1-bit field that indicates whether data in the screen field has been modified and, therefore, should be transmitted to the TCP upon completion of the ACCEPT statement. The terminal sends data only for fields with MDT set on. If the screen field has not been modified, MDT is not set and data in the screen field is not transmitted to the TCP.

You can manipulate the MDT bit programmatically. In SCREEN COBOL, the field attributes MDTON and MDTOFF also control field data transmission. The application programmer has a choice of having the data in a screen field transmitted in either of the following circumstances:

- Unconditionally upon every ACCEPT statement (MDTON)
- When the screen field has been modified (MDTOFF)

The field's MDT bit is not reset after the completion of an ACCEPT statement. Once the MDT bit is set, it stays set until the next DISPLAY BASE, TURN, RESET, or CLEAR INPUT operation. Repeated ACCEPT statements, without any of these operations in between the statements, cause previously sent data to be retransmitted. Although retransmission of data might be desirable for some applications, you can programmatically avoid resending it if you wish.

The following example illustrates the recommended use of the CLEAR INPUT verb if you do not want retransmission of terminal data.

```
START-PROGRAM.
   DISPLAY BASE screen-id.
LOOP.
   DISPLAY screen-id.
   ACCEPT screen-id UNTIL F1-KEY
   ESCAPE ON SF16-KEY.
   .
   .
   .
   CLEAR INPUT.
   GO TO LOOP.
```

The CLEAR INPUT statement resets the MDT bits and displays null values in all unprotected fields of the screens currently displayed. RESET ATTR or TURN MDTOFF can be used instead if blanking out the input fields is not desired.

You must consider another MDT convention: the TCP turns a field's MDT bit on in the following operations:

- When a TURN TEMP statement selects an input field for changing display attributes, the MDT bit is always set.
- When a RESET TEMP statement selects an input field for resetting display attributes, the MDT bit is set, regardless of the initial MDT attribute of the field.

These two exceptions apply only to the TURN and RESET statements that have the TEMP modifier.

This MDT convention allows fields to be handled correctly when they contain errors. When an error is detected in a field, a TURN TEMP statement of a display attribute is normally performed on that field, whether explicitly by the program or implicitly by the action of the ACCEPT statement. As indicated by the preceding rules, the MDT is set also, thus guaranteeing that the data from the field will again be sent from the terminal on the next read operation. After that next read operation, a RESET TEMP operation is performed, which removes the flagging display attribute while again turning the MDT bit on. The latter setting of the MDT is necessary because a subsequent read of the same data might be performed if another field is found to be in error, and the data in the field that was RESET must be sent once again to be properly accepted.

ACCEPT DATE/DAY/TIME Statement

The ACCEPT DATE/DAY/TIME statement causes the TCP to obtain the current operating system settings for date, day, and time and return them to your program data area.

```
ACCEPT accept-name FROM { DATE [YYYYMMDD] | DAY [YYYYDDD] |
TIME }
```

accept-name

is the identifier of the data item where DATE, DAY, or TIME is stored. DATE, DAY, and TIME are typically defined as:

```
PIC 9(8)   for DATE YYYYMMDD
PIC 9(6)   for DATE
PIC 9(7)   for DAY YYYYDDD
PIC 9(5)   for DAY
PIC 9(8)   for TIME
```

DATE YYYYMMDD

is the current date expressed as an 8-digit number *yyyymmdd* where *yyyy* is the year, *mm* is the month, and *dd* is the day. For example, November 25, 2002, would be returned as 20021125.

DATE

is the current date expressed as a 6-digit number *yymmdd* where *yy* is the year, *mm* is the month, and *dd* is the day. For example, November 25, 1997, would be returned as 971125.

DAY YYYYDDD

is the current Julian date expressed as a 7-digit number *yyyyddd* where *yyyy* is the year and *ddd* is the day of the year. For example, February 25, 2001, would be returned as 2001056.

DAY

is the current Julian date expressed as a 5-digit number *yyddd* where *yy* is the year and *ddd* is the day of the year. For example, March 3, 1998, would be returned as 98062.

TIME

is the current time based on a 24-hour clock, expressed as an 8-digit number *hhmmsscc* where *hh* is the hour, *mm* the minutes, *ss* the seconds, and *cc* the hundredths of seconds. For example, the time 2:41 P.M. would be returned as 14410000. The range of values allowed is 00000000 through 23595999.

The following sentence stores the current date (*yymmdd*) in *todays-date*, the Julian date (*yyddd*) in *julian-date*, and the current time (*hhmmsscc*) in *time-right-now*:

```
WORKING-STORAGE SECTION.

01  date-and-time-fields.
    05  todays-date-full PIC 9(8)  VALUE ZERO.
    05  todays-date      PIC 9(6)  VALUE ZERO.
    05  julian-date-full PIC 9(7)  VALUE ZERO.
    05  julian-date      PIC 9(5)  VALUE ZERO.
    05  time-right-now   PIC 9(8)  VALUE ZERO.
    :
PROCEDURE DIVISION.
    :
    ACCEPT todays-date-full FROM DATE YYYYMMDD
    ACCEPT todays-date FROM DATE
    ACCEPT julian-date-full FROM DAY YYYYDDD
    ACCEPT julian-date FROM DAY
    ACCEPT time-right-now FROM TIME
```

ADD Statements

The ADD statements sum numeric values and store the results in one or more data items. When defining a field to hold a total, the size of the field should be considered. The receiving field must be large enough to hold the result and thus avoid truncation of nonzero digits. The forms of the ADD statements are:

```
ADD TO
ADD GIVING
ADD CORRESPONDING
```

Each form is described in the following paragraphs.

ADD TO Statement

The ADD TO statement adds together all values specified and then adds that sum to the current value in each data item specified.

```
ADD { value } ,... TO { result } ,...
```

value

is either a numeric literal or the identifier of an elementary numeric data item.

result

is the identifier of a numeric data item to which *value*, or the sum of the values, is added.

ADD GIVING Statement

The ADD GIVING statement adds together all values specified and then replaces the current value of each data item specified with the sum.

```
ADD { value } ,... GIVING { result } ,...
```

value

is either a numeric literal or the identifier of an elementary numeric data item.

result

is the identifier of a numeric data item into which the sum of the values is stored.

ADD CORRESPONDING Statement

The ADD CORRESPONDING statement adds together elementary items in one group to any corresponding items in another group and then stores the totals in the second group used for the addition. Items correspond when they have the same names and

qualifiers up to but not including the group item name specified in the ADD CORRESPONDING statement.

```
ADD { CORR          } group-1 TO group-2
    { CORRESPONDING }
```

group-1 and *group-2*

are the identifiers of group items in which some or all of the elementary items are numeric.

The totals are placed in the *group-2* items.

The following conventions apply to data items used with the CORRESPONDING phrase:

- A REDEFINES or OCCURS clause can be specified in the data description entry of any data item.
- Data items can be subordinate to a data description entry with a REDEFINES or OCCURS clause.
- No data item can be defined with a level number 66, 77, or 88.

Subordinate data items in two different groups correspond to each other according to the following rules:

- Both data items must have the same data name.
- All possible qualifiers for the sending data item, up to but not including a group name, must be identical to all possible qualifiers for the receiving data item up to but not including the receiving group name.
- Only elementary numeric data items are considered.
- Any data item subordinate to a data item that is not eligible for correspondence is ignored.
- FILLER data items are ignored.

In the following example, all item names except staples and paper correspond. Those two items are skipped in the add operations. Notice that correspondence depends on the names of the items (and qualifiers other than the highest level ones) and not on their physical order.

```
WORKING-STORAGE SECTION.
01  cabinet-supplies.
    05  writing-tools.
        10  pencils      PIC 99.
        10  pens         PIC 99.
        10  erasers      PIC 99.
    05  paper-clips      PIC 99.
    05  staples          PIC 99.
```

```
01  stockroom-supplies.
    05  writing-tools.
        10  pencils      PIC 99.
        10  erasers      PIC 99.
        10  pens         PIC 99.
    05  paper-clips      PIC 99.
    05  paper            PIC 99.
     :
PROCEDURE DIVISION.
     :
    ADD CORRESPONDING cabinet-supplies TO stockroom-supplies.
```

In the following example, only one item (6-12-years) corresponds between the groups:

```
01  test-group-1.
    05  children.
        10  1-5-years
        10  6-12-years
    05  teen-agers.
        10  13-15-years
        10  16-19-years
    05  adults
        10  women
        10  men

01  test-group-2.
    05  children.
        10  1-3-years
        10  4-5-years
        10  6-12-years
    05  teen-agers
```

Assuming all items are numeric, the following statement sums 6-12-years of children of test-group-1 with 6-12-years of children of test-group-2:

```
ADD CORRESPONDING test-group-1 TO test-group-2
```

BEGIN-TRANSACTION Statement

The BEGIN-TRANSACTION statement marks the beginning of a sequence of operations that are to be treated as a single transaction. When this statement executes, the terminal enters transaction mode. Transaction mode is an operating mode in which Pathway servers that are configured to run under the Transaction Management Facility (TMF) can lock and update audited files.

TMF starts a new transaction and assigns a transaction-ID number to the terminal. This number is placed in the special register TRANSACTION-ID. Two other special registers are set: RESTART-COUNTER is set to 0 to indicate that the transaction is being started for the first time, and TERMINATION-STATUS is set to 1 to indicate that the transaction has started.

```
BEGIN-TRANSACTION [ ON ERROR imperative-statement ]
```

ON ERROR

provides a point of control if an error is encountered. No test is made against the transaction restart limit; the transaction is restarted and the ON ERROR branch is taken.

imperative-statement

is the statement to be executed if an error occurs or the transaction is being restarted. If the ON ERROR phrase is omitted and the number of restarts equals the transaction restart limit, the terminal is suspended, but can be restarted.

If the transaction fails for any reason while the terminal is in transaction mode, TMF backs out any updates performed on the data base for the current transaction. If the transaction was not terminated deliberately by execution of the ABORT-TRANSACTION statement, terminal execution is restarted at the BEGIN-TRANSACTION statement under these conditions:

- ON ERROR phrase is specified
- ON ERROR phrase is not specified, but the number of restarts has not exceeded the transaction restart limit. The maximum number of times a logical transaction can be automatically restarted is specified with the MAXTMFRESTARTS parameter of the PATHCOM SET PATHWAY command.

On a transaction restart, TMF assigns a new transaction-ID number to the terminal; the TCP marks the screen for screen recovery and increments by 1 the special register RESTART-COUNTER. The special register TERMINATION-STATUS remains at 1 (which indicates that a transaction is started or restarted). Working-storage items are restored to the values at execution of BEGIN-TRANSACTION. If the ON ERROR phrase is specified, the imperative statement is executed.

If the terminal is in transaction mode when the BEGIN-TRANSACTION statement is executed, the current transaction is backed out and the terminal is suspended for a pending abort. Terminal execution cannot be resumed.

The special register TERMINATION-STATUS is set by the BEGIN-TRANSACTION statement to indicate the result of execution. Table 6-2 lists the possible values of TERMINATION-STATUS.

Table 6-2. BEGIN-TRANSACTION Statement Errors

TERMINATION-STATUS	Meaning
1	The transaction is started or restarted.
2	TMF is not installed. Action without the ON ERROR phrase: the terminal is suspended for pending abort.
3	TMF is not running. Action without the ON ERROR phrase: the terminal is suspended, but can be restarted.
4	A fatal error was encountered while attempting to start the transaction. Action without the ON ERROR phrase: the terminal is suspended for pending abort.

CALL Statement

The CALL statement transfers control from one SCREEN COBOL program to another SCREEN COBOL program.

```
CALL { data-name         } [ USING { identifier } ,... ]
     { program-unit-name }

  [ ON ERROR imperative-statement ]
```

data-name

is a nonnumeric data item in the Working-Storage Section or Linkage Section; the value of the data item gives the PROGRAM-ID of another SCREEN COBOL program, as specified in the Identification Division of that program. The *data-name* specification allows the PROGRAM-ID of the called program to be specified dynamically.

program-unit-name

is a nonnumeric literal or an identifier of a nonnumeric literal that gives the PROGRAM-ID of another SCREEN COBOL program, as specified in the Identification Division of that program. A nonnumeric literal is defined in the SCREEN COBOL language as being enclosed in quotation marks. However, a search on the name of the program unit will find the program-unit name even if it is not enclosed in quotation marks. The program can be compiled for any terminal type; for more information, see Compatibility for a Called Program on page 6-27.

If a literal specifying the called program unit is also the name of a section in the calling program, an error results at compile time. This situation may be avoided by specifying the called program unit with a Working-Storage identifier of a nonnumeric literal rather than a literal.

USING

passes data to the program called. A USING phrase must be specified in the Procedure Division header of the called program. The number of identifiers for this phrase in the CALL statement must be at least as great as that specified for the USING phrase in the called program.

identifier

is the name of an argument passed to the called program. This identifier cannot exceed 12,288 bytes; it must be an 01 or 77 level data item in the Working-Storage Section or Linkage Section of the program that is calling the other program. The identifiers in the USING phrase must correspond exactly in number and structure to the number and structure of the identifiers specified in the USING phrase of the Procedure Division header of the called program. Correspondence is by position in the USING lists. See the description of the Linkage Section on page 5-3 for more information about correspondence between the Linkage Section and the CALL statement.

`ON ERROR`

provides a point of control if an error is encountered in a descendant program unit.

If a suspend class error is encountered, control is returned to the next higher level program unit having a CALL statement containing an ON ERROR clause. (A suspend class error condition is one that, without the use of the CALL...ON ERROR feature, causes the terminal to become suspended.) If a program unit containing an ON ERROR clause does not exist, the terminal is suspended at the statement where the error occurred. If the terminal is in transaction mode when a suspend class error occurs, and the point-of-control CALL...ON ERROR is beyond the scope of the current transaction, the current transaction is aborted.

If you include an ON ERROR clause in a CALL statement and it is executed, no error message is written to the Pathway log file.

`imperative-statement`

is the statement to be executed if an error occurs.

The data area of a program is initialized each time the program is called; therefore, variables do not retain their values between calls.

If the ON ERROR branch is taken, the special register TERMINATION-STATUS contains an error code describing the error, and the special register TERMINATION-SUBSTATUS contains a value or an error code further describing the error. Table 6-3 lists TERMINATION-STATUS and corresponding TERMINATION-SUBSTATUS error codes. The error code in TERMINATION-SUBSTATUS is dependent upon the error.

Table 6-3. CALL Statement Errors (page 1 of 6)

	TERMINATION-STATUS	TERMINATION-SUBSTATUS
0001	INVALID PSEUDOCODE DETECTED	(%*p-reg*)
0002	DEPENDING VARIABLE VALUE TOO BIG	
0003	INVALID SUBSCRIPT VALUE	
0004	SCREEN RECOVERY EXECUTED ILLEGAL INSTRUCTION	
0005	CALL: ACTUAL NUMBER OF PARAMETERS MISMATCHES FORMAL	
0006	CALL: ACTUAL PARAMETER SIZE MISMATCHES FORMAL	
0007	SCREEN OPERATION DONE WITHOUT BASE DISPLAYED	
0010	INTERNAL ERROR IN TERMINAL FORMAT ROUTINES	(%*p-reg*)
0011	ILLEGAL TERMINAL TYPE SPECIFIED	(%*p-reg*)
0012	SCREEN REFERENCED BUT NOT DISPLAYED	(*screen-number*)

Table 6-3. CALL Statement Errors (page 2 of 6)

	TERMINATION-STATUS	TERMINATION-SUBSTATUS
0013	OVERLAY SCREEN DISPLAYED IN TWO AREAS	
0014	ILLEGAL TERMINAL IO PROTOCOL WORD	
0015	ARITHMETIC OVERFLOW	
0016	TERMINAL STACK SPACE OVERFLOW	(*bytes*)
0017	ERROR DURING TERMINAL OPEN	(*errnum*)
0018	ERROR DURING TERMINAL IO	(*errnum*)
0019	WRONG TRANSFER COUNT IN TERMINAL IO	
0020	CALLED PROGRAM UNIT NOT FOUND	
0021	TRANSACTION MESSAGE SEND FAILURE	
0022	SEND: SERVER CLASS NAME INVALID	
0023	PSEUDOCODE SIZE TOO BIG	
0024	TCLPROG DIRECTORY ENTRY IS BAD	
0025	TERMINAL INPUT DATA STREAM INVALID	
0027	TRANSACTION MODE VIOLATION	
0028	TRANSACTION I/O ERROR	(*errnum*)
0029	TRANSACTION RESTART LIMIT REACHED	
0030	TMF NOT CONFIGURED	
0031	TMF NOT RUNNING	
0032	TMF TFILE OPEN FAILURE	(*errnum*)
0035	INSUFFICIENT TERMBUF CONFIGURED	
0036	CANNOT CALL PU WITH TERMINAL-ERRORS DECLARATIVE	
0037	ILLEGAL ACCEPT VARIABLE TIMEOUT VALUE	
0040	INVALID NUMERIC ITEM	(*number*)
0041	INVALID PRINTER SPECIFICATION	
0042	DEVICE REQUIRES INTERVENTION	(*errnum*)
0043	PRINTER I/O ERROR	(*errnum*)
0050	TERMINAL STOPPED BY PENDING REQUEST	
0051	TERMINAL SUSPENDED BY PENDING REQUEST	
0052	TERMINAL STOPPED BY PROGRAM	
0053	INVALID NUMERIC ITEM - INSTRUCTION ADDRESS	(*%address*)
0054	IO PROTOCOL DENIED	(*protocol-number*)

Table 6-3. CALL Statement Errors (page 3 of 6)

	TERMINATION-STATUS	TERMINATION-SUBSTATUS
0055	INVALID I/O PROTOCOL VALUE FROM PATHMON	(*protocol-number*)
0056	6540 CACHE ERROR; TERMINAL SUSPENDED	
0057	6540 CACHE ERROR; REVERTING TO T16-6530 EMULATION	
0058	PROGRAM UNIT OR SCOBOL CONSTRUCT REQUIRES NEWER VERSION OF TCP	
0059	I/O ERROR ON DEVICEINFO	(*errnum*)
0060	DEVICE DOESN'T SUPPORT DOUBLEBYTE CHARACTERS	
0061	DBCS TRANSLATION SUPPORT NOT INSTALLED	
0062	INVALID KATAKANA OR DBCS DATA	
0063	INVALID KATAKANA OR DBCS DATA	
0064	TRUNCATION OCCURRED DURING DISPLAY OF DBCS DATA	
0065	DEVICE DOES NOT SUPPORT KATAKANA	
0066	DEVICE DOES NOT SUPPORT KATAKANA	
0067	FIELD CONTAINS OTHER THAN DBCS DATA	
0068	FIELD CONTAINS OTHER THAN DBCS DATA	
0069	UNILATERAL ABORT: BACKUP TASK STATE NOT VALID	
0070	BACKUP TASK ERROR: UNABLE TO CHECKOPEN TERMINAL	
0071	REQUESTED DEVICE COLOR/HIGHLIGHT NOT AVAILABLE	
0072	RUN-TIME ATTRIBUTE SETTING INVALID	
0073	INSUFFICIENT TERMPOOL FOR REQUEST	(*bytes*)
0074	ILLEGAL DELAY VALUE	
0100	ERROR DURING SERVER OPEN	(*errnum*)
0101	TCLPROG DIRECTORY FILE OPEN ERROR	(*errnum*)
0102	TCLPROG CODE FILE OPEN ERROR	(*errnum*)
0103	ILLEGAL TCLPROG DIRECTORY FILE	
0104	ILLEGAL TCLPROG CODE FILE	
0106	PARAMETERS FOR TCP CONFIGURATION TOO LARGE	
0107	SWAP FILE CREATION ERROR	(*errnum*)

Table 6-3. CALL Statement Errors (page 4 of 6)

	TERMINATION-STATUS	TERMINATION-SUBSTATUS
0108	SWAP FILE OPEN ERROR	(*errnum*)
0109	SWAP FILE I/O ERROR	(*errnum*)
0110	NO ROOM FOR NEW SERVER CLASS IN TCP	
0112	REPLY NUMBER NOT KNOWN TO PROGRAM	
0113	TRANSACTION MESSAGE SIZE EXCEEDS LIMIT	
0114	MAXIMUM REPLY SIZE EXCEEDS LIMIT	
0115	TRANSACTION REPLY SIZE INVALID	(*bytes-length*)
0116	ERROR DURING SERVER I/O	(*errnum*)
0117	SERVER CLASS UNDEFINED	
0118	REQUEST INVALID FOR SERVER STATE	
0119	NO SPACE FOR NEW SERVER PROCESS	
0121	TCLPROG DIRECTORY FILE ERROR	(*errnum*)
0122	TCLPROG CODE FILE ERROR	(*errnum*)
0123	NO SERVER PROCESS LINKED TO	
0124	APPLICATION DEFINED ERROR--EXIT PROG WITH ERROR	(*termination-status*)
0125	MULTIPLE UNSOLICITED MESSAGES REJECTED DUE TO TERM STOP/SUSPEND	
0161	I/O ERROR	(*errnum*)
0162	RECEIVED MESSAGE SMALLER THAN EXPECTED	(*byte-length*)
0163	RECEIVED MESSAGE LARGER THAN EXPECTED	
0164	CODE OF RECEIVED MESSAGE UNDEFINED	
0165	EDIT ERROR OCCURRED ON MESSAGE INPUT	
0166	RECEIVED MESSAGE EXCEEDS MAXIMUM ALLOWABLE SIZE	(*byte-length*)
0167	MESSAGE TO SEND EXCEEDS MAXIMUM ALLOWABLE SIZE	(*byte-length*)
0168	DEVICE SUBCLASS INVALID	
0169	ILLEGAL TIMEOUT VALUE	(*timeout*)
0170	INVALID END OF MESSAGE CHARACTER ENCOUNTERED	
0171	FIELD LENGTH EXCEEDS MAXIMUM ALLOWABLE LENGTH	(*byte-length*)
0172	MESSAGE LENGTH EXCEEDS MAXIMUM ALLOWED	(*byte-length*)

Table 6-3. CALL Statement Errors (page 5 of 6)

	TERMINATION-STATUS	TERMINATION-SUBSTATUS
0173	I/O ERROR ON CONTROL-26 OPERATION	(*errnum*)
0174	CONTROL-26 OPERATION DID NOT COMPLETE IN TIME	
0175	EDIT ERROR OCCURRED ON MESSAGE OUTPUT	
0176	ATTEMPT TO RECEIVE UNSOLICITED MESSAGE WITH ONE NOT YET REPLIED TO	
0177	NO UNSOLICITED MESSAGE TO REPLY TO	
0178	ATTEMPT TO RECEIVE UNSOLICITED MESSAGE WHEN TERM MAXINPUTMSGS = 0	
0179	DATA LEFT OVER ON SCATTER TO WORKING STORAGE	
0180	NOT ENOUGH DATA FOR SCATTER TO WORKING STORAGE	
0181	VARIABLE FIELD SIZE WOULD EXCEED DECLARED FIELD SIZE	
0182	DELIMITER IS NOT BYTE ALIGNED	
0183	DEPENDING VALUE IS OUT OF BOUNDS	
0184	CONFLICT OF DATA TYPES DURING 'PRESENT IF' DETERMINATION	
0185	FIELD OCCURRENCE EXCEEDS WORKING STORAGE MAXIMUM OCCURRENCE	
0200	INVALID FORMAT MESSAGE RECEIVED BY TCP	
0201	SYNCID VIOLATION IN MESSAGE RECEIVED BY TCP	
0202	TERMINAL IDENTIFIER NOT KNOWN TO TCP	
0203	FUNCTION UNIMPLEMENTED	
0204	TCP STATE DOES NOT ALLOW OPERATION	
0205	TERMINAL STATE DOES NOT ALLOW OPERATION	
0206	TCP CANNOT HANDLE MORE TERMINALS	
0207	REQUEST PENDING	
0208	NO ROOM FOR ANOTHER ALTERNATE TCLPROG	
0209	PROCESSOR DOES NOT HAVE PATHWAY MICROCODE	
0210	BACKUP PROCESSOR DOWN	
0211	BACKUP NEWPROCESS FAILURE	(*process-creation-detail*))

Table 6-3. CALL Statement Errors (page 6 of 6)

	TERMINATION-STATUS	TERMINATION-SUBSTATUS
0212	BACKUP FAILED	
0213	OPEN OF FILE TO BACKUP FAILED	(*errnum*)
0214	FILE-SYSTEM ERROR DURING CHECKPOINT	(*errnum*)
0215	ERROR IN BACKUP	< nested message >
0216	TCP INTERNAL ERROR	(%*p-reg*)
0217	TCP TRAP	(%*p-reg*)
0218	TAKEOVER BY BACKUP	
0219	TCP MEMORY DUMP TAKEN	*file-name*
0220	INSPECT NOT ENABLED FOR TCP	
0221	INSPECT TERMINAL TABLE FULL	
0222	INSPECT BREAKPOINT TABLE FULL	
0223	REQUEST NOT ALLOWED WHILE AT BREAKPOINT	
0224	I/O ERROR WITH IMON OR INSPECT	(*errnum*)
0225	TASK ALREADY USING ANOTHER INSPECT TERMINAL	
0226	REQUESTED FUNCTION NOT SUPPORTED IN THIS RELEASE	
0227	SEND: EXTERNAL PATHMON NAME INVALID	
0228	SEND: EXTERNAL SYSTEM NAME INVALID	
0229	SEND: EXTERNAL SYSTEM NAME NOT DEFINED	
0230	SEND: NO ROOM FOR NEW EXTERNAL PATHMON IN TCP	
0231	SEND: ERROR DURING I/O TO EXTERNAL PATHMON	(*errnum*)
0232	MAXTERMDATA IN TCP CONFIGURATION TOO SMALL	
0233	SERVER PROCESS UNKNOWN	
0239	GUARDIAN-LIB INCOMPATIBLE WITH TCP	
0240	VALUE FOR MAXINPUTMSGS TOO LARGE	
0241	UNSOLICITED MESSAGE REJECTED BY TCP	(*errnum*)
0242	MULTIPLE UNSOLICITED MESSAGES REJECTED	
0243	PATHTCP2 CANNOT EXECUTE ON THIS RELEASE OF GUARDIAN	

The TERMINATION-STATUS error numbers related to CALL ... ON ERROR correspond directly to the Pathway error messages generated by the TCP in the 3000 through 3999 range. For example, TERMINATION-STATUS error 114 corresponds to Pathway error message 3114. For descriptions of the messages, see the TCP messages in the *Compaq NonStop™ Pathway/iTS System Management Manual.*

Note that TERMINATION-STATUS becomes undefined when TERMINATION-STATUS is set for reasons other than CALL ... ON ERROR return.

Refer to the EXIT PROGRAM statement for additional information on programmatic control of error conditions.

Compatibility for a Called Program

Programs compiled for a particular terminal type are allowed to call programs compiled for any other terminal type. You must ensure that the called program is compatible with the calling program so that the results are predictable. The following are considerations regarding compatibility between a called and a calling program.

- A conversational mode program running on a terminal with block mode capabilities can call a block mode program that uses only features available on the terminal.
- A program compiled with no terminal type specified (TERMINAL clause of the OBJECT-COMPUTER paragraph) assumes a very limited set of screen display attributes and can be called by nearly any other block mode program.
- The screen display attributes used in the called program must be applicable to the calling program; otherwise, the results are unpredictable.
- The function keys defined as system names in the called program must be applicable to the calling program; otherwise, the results are unpredictable.
- The character sets used in the programs must be compatible. Note that if a called program is compiled for a 6530 terminal and uses the default character set specified in the terminal's configuration file, the USASCII character set is used until the first DISPLAY BASE operation occurs.
- During a DISPLAY BASE operation, a program unit is aborted if the terminal does not support the attributes requested in the SET MINIMUM-ATTR or SET MINIMUM-COLOR statement.
- Termination status 71 indicates insufficient support for the color, highlight, or outline display attributes requested.

For characteristics of different terminal types, see the *Compaq NonStop™ Pathway/iTS TCP and Terminal Programming Guide*.

CHECKPOINT Statement

The CHECKPOINT statement causes the current context for the terminal, such as Working-Storage items, to be checkpointed.

```
CHECKPOINT
```

This statement causes an additional checkpoint. Automatic checkpointing occurs as follows:

- At execution of a BEGIN-TRANSACTION statement, the TCP performs a full context checkpoint.
- At execution of an END-TRANSACTION statement, the TCP performs a full context checkpoint.
- For a SEND statement to a server not using TMF (PATHCOM command SET SERVER TMF OFF), the TCP performs a checkpoint before and after the SEND.
- At execution of a RECONNECT MODEM statement, the TCP performs a full context checkpoint.

No automatic checkpointing occurs on a SEND statement to a TMF server (SET SERVER ON). For more information about TCP checkpointing, see the TCP checkpointing strategy description in the *Compaq NonStop™ Pathway/iTS TCP and Terminal Programming Guide*.

If the CHECKPOINT statement is issued while the terminal is in transaction mode, the terminal is suspended for pending abort.

CLEAR Statement

The CLEAR statement prepares the terminal for a new set of input. This statement cannot be used by programs communicating with intelligent devices.

```
CLEAR INPUT
```

This statement stores null values into all unprotected fields of the screens currently displayed and resets the modified data tag (MDT) bits of all unprotected fields on terminals that use MDT. Except for the MDT, the attributes of the fields are not affected.

The CLEAR statement differs from the RESET statement as follows:

- CLEAR affects all unprotected fields on the display screen; RESET affects only those fields specified in the statement, whether the fields are protected or not.
- CLEAR causes all unprotected fields to become blank; RESET returns all fields to the initially declared values.
- CLEAR does not affect field attributes, although the MDT bits are cleared; RESET returns all fields to their initial values.

- CLEAR requires only a short data sequence; RESET requires a data sequence from the TCP for each field referred to in the statement.

COMPUTE Statement

The COMPUTE statement evaluates an arithmetic expression and then stores the result in one or more data items.

```
COMPUTE { result } ,... = expression
```

result

is the identifier of a numeric elementary item.

expression

is an arithmetic expression calculated according to precedence rules described in Section 2, SCREEN COBOL Source Program.

As with other arithmetic operations, consider truncation situations and how they should be handled.

The following example illustrates the COMPUTE statement:

```
WORKING-STORAGE SECTION.

77  compute-result      PIC 999     VALUE ZEROS.
77  ws-result           PIC S9(9)   VALUE ZEROS.
77  ws-99               PIC S99     VALUE 99.
77  ws-five-ones        PIC S9(5)   VALUE 11111.
01  exponent            PIC 9(5)    VALUE ZERO   COMP.
    :
    :
    COMPUTE compute-result = (((24.0 + 1) * (60 - 10)) / 125).

(compute-result = 10)
```

COPY Statement

The COPY statement inserts sections of code into a program for use at compile time. This allows code that is common to several programs to be written once and be maintained easily.

```
COPY copy-text [ { OF } library-name ] .
               [ { IN }              ]
```

copy-text

is a unique section name in a SCREEN COBOL copy library file.

library-name

is the file containing the text to be copied. The name is expanded to a full file name using the default subvolume in effect for the compilation. If you specify the library name with a subvolume and a file name, you must enclose the entry in quotation marks. For example:

```
"subvol.afile".
```

If *library-name* is omitted and *copy-text* exists, the default library name COPYLIB is used for the compilation.

Even though the COPY statement is described as a Procedure Division statement, the statement can be included in a SCREEN COBOL program wherever a character string or separator can appear; the only exception is within another COPY statement. The keyword COPY cannot be split over two lines, but text that follows the keyword can be continued.

Library text is copied into the source program. The SCREEN COBOL copy library must be in the correct format, and each *copy-text* must be written in correct SCREEN COBOL syntax.

A copy library must be an EDIT disk file in the following form:

```
?SECTION copy-text-1 [ , { ANSI   } ]
                     [   { TANDEM } ]
text-1

?SECTION copy-text-2 [ , { ANSI   } ]
                     [   { TANDEM } ]
text-2
```

Each SECTION line identifies the beginning of a *copy-text*; the question mark must be in column 1. The content of the text is arbitrary and can be any length. No text line can begin with ?SECTION.

The compiler assumes the source format (ANSI standard reference format or Tandem standard reference format) of the library text is the same as that of the line containing the COPY statement. When the format option is specified, the format overrides the compiler's assumption, permitting a library text to be copied irrespective of the format of the source program. Also, the library text itself can have compiler commands, which are executed when the text is copied. Note that after copying is complete, the compiler always reverts to the format in effect when it encountered the COPY statement. See Section 7, Compilation for information on restrictions on the use of the Compaq *Inspect* debugging tool when the ANSI compiler command is set.

During program compilation, *copy-text* is found by locating the SECTION command whose *copy-text* name matches *copy-text* in the COPY statement. Text is copied starting at the line after the SECTION line and continues until either another SECTION line is recognized or end-of-file is reached. In the following example, text-0 has no SECTION command and could never be copied:

```
text-0
```

```
?SECTION copy-text-1
text-1
```

When a library file begins like this, this text could be comments about the library contents.

In the following example, notice that employee-detail of the COPY statement is not qualified because the copy library, named COPYLIB, resides on the default volume and subvolume for the compilation.

The contents of the copy library COPYLIB are as follows:

```
?SECTION employee-detail
01  emp-data-in.
   05  emp-no      PIC X(05).
   05  emp-name    PIC X(20).
   05  dept        PIC X(03).
   05  job-class   PIC X(05).
   05  hourly-rate  PIC 9(3)V99.
   05  deductions   PIC 9(3)V99.
   05  salary      PIC 9(7)V99.
```

The SCREEN COBOL source code is as follows:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
 COPY employee-detail.
```

In the compile listing in the example below, all lines from a copy library are marked with a <.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
  COPY employee-detail.

< 01  emp-data-in.
<     05  emp-no       PIC X(05).
<     05  emp-name     PIC X(20).
<     05  dept         PIC X(03).
<     05  job-class    PIC X(05).
<     05  hourly-rate  PIC 9(3)V99.
<     05  deductions   PIC 9(3)V99.
<     05  salary       PIC 9(7)V99.
```

DELAY Statement

The DELAY statement delays program execution for a specified period of time.

```
DELAY { numeric-literal }
      { identifier      }
```

numeric-literal

is a numeric value representing one-second units; the maximum value is 21474836. The value must be a positive integer.

identifier

is the identifier of an integer data item representing one-second units; the maximum value is 2,147,483,647 seconds (31 bits). This value must be a positive integer.

Note. The maximum values are enforced only at run-time; the compiler will not generate an error for a value greater than 2,147,483,647.

This statement is intended for use in situations where an error has occurred (such as a terminal I/O error because power to the terminal is off) and the operation encountering the error is to be retried periodically.

An alternate use of the DELAY statement (DELAY 0) causes immediate execution. For example, this statement can be specified after a DISPLAY statement (for block mode) to cause an immediate output of the contents of the terminal buffer to the screen.

The following example illustrates the DELAY statement:

```
* highest level program-unit.
 loop.
     CALL menu ON ERROR PERFORM analyze-error.
     IF retry = 1
* delay five minutes, then retry.
        DELAY 300
        GO TO loop
     ELSE
        GO TO giveup.

 analyze-error.
     IF TERMINATION-STATUS = 18 AND
        ( TERMINATION-SUBSTATUS = 171 OR
          TERMINATION-SUBSTATUS = 173 )
        MOVE 1 TO retry
     ELSE
        MOVE 0 TO retry.

* suspend.
 giveup.
     EXIT PROGRAM WITH ERROR.
```

DEVICEINFO Statement

The DEVICEINFO statement obtains information about the intelligent device to which the SCREEN COBOL program is currently sending messages. This statement performs the same function as the DEVICEINFO procedure.

```
DEVICEINFO USING deviceinfo-rec
```

deviceinfo-rec

is an 01 group defined as follows:

```
01  deviceinfo-rec.
    02 file-name                PIC X(16).
    02 device-type              PIC 9(4) comp.
    02 device-subtype           PIC 9(4) comp.
    02 physical-record-length   PIC 9(4) comp.
```

The SCREEN COBOL compiler does not check the data types in the *deviceinfo-rec* group item. It assumes that you have defined it correctly.

DEVICEINFO returns the file name in internal network format. Refer to the *Guardian Programmer's Guide* for a description of this format.

This statement could be used as follows:

```
MOVE TERMINAL-FILENAME TO file-name OF deviceinfo-rec.
DEVICEINFO USING deviceinfo-rec.
```

DISPLAY BASE Statement

The DISPLAY BASE statement formats the specified screen from the screen definition in the Screen Section. The statement selects the screen description for subsequent operations but does not display the screen.

The DISPLAY BASE DYNAMIC statement performs the same function as the two-statement sequence DISPLAY BASE and DISPLAY. (DISPLAY follows DISPLAY BASE.)

The DISPLAY BASE statement operates differently for terminals in block mode from terminals in conversational mode. It cannot be used for communication with intelligent devices.

For terminals in block mode, DISPLAY BASE formats data for screen literals, VALUE clauses, null characters, and fill characters, and establishes attributes for screen fields. DISPLAY BASE also clears the current screen display. DISPLAY BASE does not cause current values from Working-Storage items to appear in output fields of the screen; DISPLAY formats these values.

For terminals in both block and conversational mode, DISPLAY BASE establishes the current screen, which serves as the foundation for all other screen operations. Therefore, this statement must execute before other display operations execute. A

second DISPLAY BASE can execute at any time to establish a new screen, or to reestablish the same screen.

```
DISPLAY BASE [DYNAMIC] base-screen-name
```

DYNAMIC

specifies that nonliteral screen fields can acquire their initial contents from their FROM or USING Working-Storage data item. A subsequent DISPLAY statement is not needed to display the initial Working-Storage values.

In addition, the DYNAMIC modifier provides the capability of changing the screen field attribute settings at run-time by using the contents of individual attribute elements in an associated control structure.

base-screen-name

is the name of the base screen.

When a shadowed screen field has an associated Working-Storage item with its SELECT bit set to 1, the screen field acquires its initial screen contents from its FROM or USING Working-Storage data item. Otherwise, the initial screen contents are acquired from the compile-time literal value or the default initial value.

DISPLAY BASE (without the DYNAMIC modifier) does not physically display the screen. The screen data is displayed when one of the following events occurs:

- Terminal buffer (TERMBUF) fills up
- One of the following statements is executed: ACCEPT; BEGIN-TRANSACTION, END-TRANSACTION, RESTART-TRANSACTION, or ABORT-TRANSACTION; CALL; CHECKPOINT; DELAY; EXIT PROGRAM; PRINT SCREEN; or SEND
- The DISPLAY statement is executed for a conversational terminal

During execution of the first DISPLAY BASE statement for a SCREEN COBOL program, the I/O startup messages prepare the terminal for Pathway/iTS operation. The program can then act on any terminal I/O errors through the CALL ON ERROR clause.

For the DISPLAY BASE statement in block mode, the input and output fields of the screen are filled with the values specified in the VALUE clauses for the fields (unless the DYNAMIC modifier is used and the field’s SELECT bits are set to 1, in which case the values are taken from the associated Working-Storage data items). A field that has no VALUE clause is filled with the fill character. For variable-length tables (defined with OCCURS DEPENDING ON), the table is filled to the maximum length possible as specified in the definition, regardless of the current value of the table's controlling variable.

Using DISPLAY BASE

A running SCREEN COBOL program has at most one current base screen; the current base screen is defined by the most recently executed DISPLAY BASE statement. The program can have at most one current overlay screen associated with each of the overlay areas of the current base screen; the current overlay screen is defined by the most recently executed DISPLAY OVERLAY statement for each of the areas. With the exception of the DISPLAY BASE and DISPLAY OVERLAY statements, all screen operations must deal only with the current screens.

The definition of a screen is local to a SCREEN COBOL program; therefore, a program cannot use a current screen that was established by another program, even if the declaration of the current screen is identical to the declaration of the screen in the currently executing program. Consequently, the program must perform a DISPLAY BASE operation to use a screen.

If a program has current screens defined and calls another program that has screen declarations, the current screens become undefined for the first program. If the first program is to make use of the screens it previously displayed, the first program must execute DISPLAY BASE/OVERLAY statements after the call to the program has completed.

If a program calls another program that has no screen declarations or does not exist, the definitions of the current screens remain unchanged.

When a program that has defined the current screens executes an EXIT PROGRAM statement, the current screens become undefined. The program must display the screens again to make use of them even if no intervening screen operations have occurred since its exit.

DISPLAY OVERLAY Statement

The DISPLAY OVERLAY statement formats the specified overlay screen from the screen definition in the Screen Section and associates the overlay screen with the overlay area in the current base screen. The statement also selects the overlay screen description for subsequent operations, but does not display the screen.

The DISPLAY OVERLAY DYNAMIC statement performs the same function as the two-statement sequence DISPLAY OVERLAY and DISPLAY. (DISPLAY follows DISPLAY BASE.)

The DISPLAY OVERLAY statement operates differently for terminals in block mode from terminals in conversational mode. It cannot be used for communication with intelligent devices.

For terminals in block mode, DISPLAY OVERLAY formats data for screen literals, VALUE clauses, null characters, and fill characters, and establishes attributes for screen fields. The overlay screen replaces any previous screen in the overlay area. DISPLAY OVERLAY does not transmit current values from Working-Storage items to output fields of the screen; DISPLAY transmits these values.

For terminals in both block and conversational mode, DISPLAY OVERLAY establishes the current overlay screen and must execute before other screen operations using the overlay screen.

```
DISPLAY OVERLAY { [DYNAMIC]overlay-screen-name } AT overlay-
                { SPACES                       }      area
```

DYNAMIC

specifies that nonliteral screen fields can acquire their initial contents from their FROM or USING Working-Storage data item. A subsequent DISPLAY statement is not needed to display the initial Working-Storage values.

In addition, the DYNAMIC modifier provides the capability of changing the screen field attribute settings at run-time using the contents of individual attribute elements in an associated control structure.

overlay-screen-name

is the name of the overlay screen to be displayed.

AT *overlay-area*

is the name of the overlay area of the currently displayed base screen into which the overlay screen is to be placed.

SPACES

causes the overlay area to become blank and restores the area to the state it was in immediately after the base screen was displayed. Any association of an overlay screen with the overlay area is broken.

When a shadowed screen field has an associated Working-Storage item with its SELECT bit set to 1, the screen field acquires its initial screen contents from its FROM or USING Working-Storage data item. Otherwise, the initial screen contents are acquired from the compile-time literal value or the default initial value.

The DISPLAY BASE statement must appear before the DISPLAY OVERLAY statement, or else an error is generated.

The overlay area must be at least as large as the overlay screen. An overlay screen cannot be displayed in more than one overlay area at the same time.

DISPLAY RECOVERY Statement

The DISPLAY RECOVERY statement initiates screen recovery. A program can use this statement to implement a request by a terminal operator for screen recovery, thus eliminating duplication of code for recovery actions.

This statement cannot be used by programs communicating with intelligent devices.

```
DISPLAY RECOVERY
```

When DISPLAY RECOVERY executes, the standard error recovery procedure is executed. The recovery process performs the equivalent of a DISPLAY BASE statement for the current base screen followed by a DISPLAY OVERLAY operation for all currently active overlay screens. The screen recovery process then executes any screen recovery declarative procedures that have been provided in the SCREEN COBOL program.

DISPLAY Statement

The DISPLAY statement formats data for selected output fields for transmission to the screen.

The DISPLAY BASE statement operates differently for terminals in block mode than it does for terminals in conversational mode. It cannot be used for communication with intelligent devices.

```
DISPLAY [ TEMP      ] [ nonnumeric-literal IN ]
        [ TEMPORARY ]

  { screen-identifier } ,...

  [ DEPENDING [ ON ] identifier ]
  [ SHADOWED                    ]
```

`TEMP` or `TEMPORARY`

marks the fields so that they will be reset to their default values when the next RESET TEMP or ACCEPT statement has completed executing.

If the terminal is operating in conversational mode, this phrase is ignored and DISPLAY performs normally. To change, temporarily, the value of a screen item, the current value of the associated Working-Storage item must be saved, the value changed, the new value displayed, and the previous current value restored.

`nonnumeric-literal`

is a value that is sent to the terminal for each selected field. The value is not converted; it is truncated or extended with the fill character if necessary. The value must be in quotation marks.

If this clause is omitted, the data for a selected screen field is obtained from the Working-Storage data item specified in the FROM or USING clause of the screen field description. The data is converted and edited according to the screen field declaration, and those characters are placed in the field on the terminal display.

`screen-identifier`

is a screen, screen group, or elementary output item of any active screen; the maximum is 127 items per DISPLAY statement. When *`screen-identifier`* is not an elementary item, it refers to all subordinate elementary items that have a VALUE, FROM, or USING clause in their definitions.

DEPENDING ON *identifier*

selects either zero or one *screen-identifier* from the list of screen fields. The statement whose position in the *screen-identifier* list is the same as the value in *identifier* is selected. If the value in *identifier* is less than 1 or greater than the number of screen identifiers, no *screen-identifier* is selected.

SHADOWED

selects from the *screen-identifier* list only those fields that have SHADOWED items in which the SELECT bit is set. Fields in the *screen-identifier* list that do not have SHADOWED items are not selected.

Note. If neither the DEPENDING ON modifier nor the SHADOWED modifier is specified, all fields in the list are selected.

The DEPENDING ON clause for the DISPLAY statement is analogous to the DEPENDING ON clause for the PERFORM ONE statement. The following example illustrates this.

```
WORKING-STORAGE SECTION.
    :
77  ws-screen-status    PIC 9(4)    COMP VALUE 1.
01  ws-fld1             PIC x(10).
01  ws-fld2             PIC x(10).
01  ws-fld3             PIC x(10).

SCREEN SECTION.
    :
01  MENU1 SIZE 24, 80.
      05 screen-fld1  at 4, 20
                      PIC X(10)
                      from fld1.
      05 screen-fld2  at 5, 40
                      PIC X(10)
                      from fld2.
      05 screen-fld3  at 6, 60
                      PIC X(10)
                      from fld3.

PROCEDURE DIVISION.
    :
BODY-PARAGRAPH.
    :
  DISPLAY BASE MENU1.
  DISPLAY SCREEN-FLD1,
          SCREEN-FLD2,
          SCREEN-FLD3,
  DEPENDING ON WS-SCREEN-STATUS.
```

If WS-SCREEN-STATUS equals 1, SCREEN-FLD1 is displayed. If WS-SCREEN-STATUS equals 2, SCREEN-FLD2 is displayed, and so on. It is not considered erroneous if WS-SCREEN-STATUS < 1 or WS-SCREEN-STATUS > 3. Control just

falls through (execution continues with the next statement) and no screen field is displayed.

The execution of DISPLAY in block mode does not cause a physical write to the screen but causes data to be written to the terminal buffer. A physical write to the screen occurs when one of the following events occurs:

- Terminal buffer (TERMBUF) fills up
- Execution of one of the following statements: ACCEPT; BEGIN-TRANSACTION, END-TRANSACTION, RESTART-TRANSACTION, or ABORT-TRANSACTION; CALL; CHECKPOINT; DELAY; EXIT PROGRAM; PRINT SCREEN; or SEND
- Execution of DISPLAY for a conversational terminal

For terminals operating in conversational mode, the DISPLAY statement presents output in order by rows. A screen field value appears on the screen at the column number position specified in the screen field description. Blank lines are not generated (for formatting purposes), so screen lines generally do not correspond with the line numbers specified in the Screen Section.

To display fully line-formatted screens, define at least one item for every line (row) of the screen. If a row of spacing is required, define the screen item for that row with a VALUE clause specifying blanks; for example, VALUE " ". Then, prepare the entire screen buffer by specifying the screen name as the screen identifier in the DISPLAY statement.

For terminals operating in conversational mode, the DISPLAY statement performs as follows:

- The DISPLAY statement places screen items on the output line in the column location specified in the Screen Section. If another screen item has the same line number description, but is not named in the DISPLAY statement, that screen item appears in the screen display.
- If you specify a screen group name to display multiple screen fields, each screen field appears in the column described for that field. However, the screen fields are on consecutively numbered lines regardless of the screen field descriptions.
- Any nonfiller screen item must be defined with a TO, FROM, or USING clause in the Screen Section. If a screen item is defined with both a VALUE clause and a TO, FROM, or USING clause, the literal in the VALUE clause is never displayed. The DISPLAY statement output is always from the associated Working-Storage data items.
- If *nonnumeric-literal* is listed in a DISPLAY statement and the screen-identifier list contains more than one field, the literal appears in each of the screen fields named in the list.

The Break key for conversational terminals can be enabled to terminate a DISPLAY operation. The Break key is enabled through the PATHCOM commands SET TERM or SET PROGRAM TYPE in the Pathway system configuration. If the Break key is

enabled and is pressed during a DISPLAY operation, the DISPLAY terminates and no terminal error condition results.

The value of the TERMINATION-SUBSTATUS special register on the DISPLAY operation is:

- 1 if the BREAK key is pressed
- 0 if the BREAK key is not pressed

DIVIDE Statements

The DIVIDE statements divide one data item into another and store the results in one or more data items. The forms of the DIVIDE statements are:

```
DIVIDE INTO
DIVIDE GIVING
DIVIDE BY GIVING
```

Each form is described in the following paragraphs.

DIVIDE INTO Statement

The DIVIDE INTO statement divides one data item into one or more other data items.

```
DIVIDE divisor INTO { dividend } ,...
```

divisor

is either a numeric literal or the identifier of an elementary numeric data item.

dividend

is the identifier of an elementary numeric data item that is the dividend and receiving field for the quotient.

DIVIDE GIVING Statement

The DIVIDE GIVING statement divides one data item into another and stores the quotient in one or more data items.

```
DIVIDE divisor INTO dividend GIVING { quotient } ,...
```

divisor

is either a numeric literal or the identifier of an elementary numeric data item.

dividend

is either a numeric literal or the identifier of an elementary numeric data item.

quotient

is the identifier of an elementary numeric data item where the quotient is stored.

DIVIDE BY GIVING Statement

The DIVIDE BY GIVING statement is the same as DIVIDE GIVING, except the dividend is specified first.

```
DIVIDE dividend BY divisor GIVING { quotient } ,...
```

dividend

is either a numeric literal or the identifier of an elementary numeric data item.

divisor

is either a numeric literal or the identifier of an elementary numeric data item.

quotient

is the identifier of an elementary numeric item where the quotient is stored.

The following example illustrates the DIVIDE BY GIVING statement:

```
WORKING-STORAGE SECTION.
77  leap-year        PIC 9   VALUE ZERO.
77  divide-result    PIC 99  VALUE ZERO.
01  invoice-date.
    05  inv-month    PIC 99.
    05  inv-day      PIC 99.
    05  inv-year     PIC 99.
    :
PROCEDURE DIVISION.
    :
    DIVIDE inv-year BY 4 GIVING divide-result.
    :
```

END-TRANSACTION Statement

The END-TRANSACTION statement marks the completion of a sequence of operations that are treated as a single transaction. When this statement executes, the terminal leaves transaction mode. Transaction mode is an operating mode in which Pathway servers that are configured to run under the Transaction Management Facility (TMF) can lock and update audited files.

```
END-TRANSACTION
```

If TMF accepts this statement, any database updates made during the transaction become committed, the terminal leaves transaction mode, and the special register TRANSACTION-ID is set to SPACES. If TMF rejects this statement, transaction restart occurs.

If the terminal is not in transaction mode when the END-TRANSACTION statement is executed, the terminal is suspended for a pending abort.

EXIT Statements

The EXIT statements mark the end of a procedure or the exiting point of a subprogram. The forms of the EXIT statements are:

```
EXIT
EXIT PROGRAM
```

Each form is described in the following paragraphs.

EXIT Statement

The EXIT statement marks the end of a procedure. The statement performs no operation. However, if an EXIT statement is referenced by a previous PERFORM statement, the EXIT statement (like any other statement in a SCREEN COBOL program) provides the instructions for the return linkage to the statement after the PERFORM statement.

```
EXIT .
```

EXIT PROGRAM Statement

The EXIT PROGRAM statement marks the logical end of a called program. When this statement is executed in a called program, control returns to the calling program. If the program executing the EXIT PROGRAM statement is the initial program used when the terminal was started, the terminal is stopped.

```
EXIT PROGRAM [ WITH ERROR ] .
```

`WITH ERROR`

is an option that provides a way to reassert the error condition described by special registers TERMINATION-STATUS and TERMINATION-SUBSTATUS. The error condition states that if a suspend class error is encountered, control is returned to the next higher level program unit having a CALL statement with an ON ERROR clause; if a program unit with the CALL...ON ERROR feature does not exist, the terminal is suspended without possibility of restart.

Note that the program can change the contents of TERMINATION-STATUS and TERMINATION-SUBSTATUS before executing the EXIT PROGRAM WITH ERROR statement. Values for TERMINATION-STATUS must be in the range of 0 through 255.

The EXIT PROGRAM statement must appear in a sentence by itself and must be the only sentence in the paragraph.

GO TO Statements

The GO TO statements pass control from one part of the Procedure Division to another. The forms of the GO TO statements are:

```
GO TO
GO TO DEPENDING
```

Each form is described in the following paragraphs.

GO TO Statement

The GO TO statement unconditionally passes control from one part of the Procedure Division to another.

```
GO [ TO ] procedure-name
```

procedure-name

is the name of the procedure to which control is transferred.

GO TO DEPENDING Statement

The GO TO DEPENDING statement passes control to one of several procedures depending on a variable data item.

```
GO TO { procedure-name } ,... DEPENDING [ ON ] depend
```

procedure-name

is a series of procedure names. Only one is chosen, based on the value of *depend*.

depend

is the identifier of an elementary numeric integer data item. This item acts like an index because its value selects the procedure name to which the program branches. If the value of *depend* is outside the range of *procedure-name*, no branching occurs and control passes to the next statement.

The following example illustrates the GO TO DEPENDING statement:

```
procedure-branch.
     GO TO proc-1,
           proc-2,
           proc-3, DEPENDING ON branch-flag.
     MOVE 0 to branch-flag.
```

- If branch-flag is 1, control passes to proc-1.
- If branch-flag is 2, control passes to proc-2.
- If branch-flag is 3, control passes to proc-3.

- If branch-flag is less than 1 or greater than 3, control passes to the statement immediately following the GO TO DEPENDING statement.

IF Statement

The IF statement evaluates a condition and then transfers control depending on whether the value of the condition is true or false.

```
IF condition { statement-1   } [ ELSE { statement-2   } ]
             { NEXT SENTENCE } [      { NEXT SENTENCE } ]
```

condition

is any conditional expression.

statement-1, statement-2

are imperative or conditional statements. Each statement can contain an IF statement, in which case the statement is referred to as a nested IF statement.

NEXT SENTENCE

is a substitution for *statement-1* or *statement-2*. The phrase performs no operation, but is used to preserve the syntactical structure or to emphasize that one value of condition elicits no action.

IF statements within IF statements are considered as paired IF and ELSE statements, proceeding from left to right. An ELSE is assumed to apply to the immediately preceding IF that has not already been paired with an ELSE.

The following conventions apply to the IF statement:

- If *condition* is true, *statement-1* is executed; if NEXT SENTENCE has been substituted for *statement-1*, no operation is performed.
- If *condition* is false, *statement-2* is executed; if NEXT SENTENCE has been substituted for *statement-2* or if the ELSE clause has been omitted, no operation is performed.
- If a GO TO statement that causes a transfer of control is executed as part of *statement-1* or *statement-2,* control is unconditionally transferred to the target of the GO TO statement.
- If control is not unconditionally transferred by execution of a GO TO statement as part of *statement-1* or *statement-2,* control passes to the next executable statement following the IF statement after all statements executed as part of the IF statement have completed.
- Comparisons (using GREATER THAN, LESS THAN, EQUAL, and so on) of a PIC N data item or literal with a numeric data item (PIC 9) are not allowed. All other comparisons are allowed and are done on a byte-by-byte basis.

The following example illustrates a simple IF statement:

```
IF julian-days IS GREATER THAN 59,
   ADD leap-year TO julian-days.
```

The following example illustrates a simple IF ELSE statement:

```
IF tally GREATER THAN 0
   MOVE 0 TO tally
   MOVE 3 TO msg-index
   PERFORM print-error-routine
ELSE
   MOVE 1 TO flag.
```

The following example illustrates nested IF statements:

```
IF employee-number NOT EQUAL TO SPACES
    PERFORM read-routine
    IF no-error
       PERFORM list-record-out
       IF yes
          PERFORM delete-master
          IF no-error
             ADD 1 TO delete-count
          ELSE
             NEXT SENTENCE
       ELSE
          MOVE 0 TO flag
    ELSE
       NEXT SENTENCE
ELSE
   MOVE 1 TO flag.
```

IF ... DOUBLEBYTE Statement

The IF ... DOUBLEBYTE statement tests for the existence of only double-byte characters in an alphanumeric data item.

```
IF data-name [ IS ] [ NOT ] DOUBLEBYTE
```

Aligned double spaces are seen as %H2020 and are valid double-byte characters. A single space or a nonaligned space is not a double-byte character.

IF ... WITHIN Statement

The IF ... WITHIN statement determines whether the cursor, positioned at the row and column values stored in OLD-CURSOR, NEW-CURSOR, or an appropriate Working-Storage data item, is within an elementary screen item or an elementary screen overlay item.

```
IF [ NOT ] data-item [ NOT ] WITHIN screen-item

 [ AT screen-area ]
```

data-item

specifies the OLD-CURSOR or NEW-CURSOR special register or specifies a Working-Storage data item defined as follows:

```
01  WS-USER-SAVE-CURSOR.
    02  WS-USER-SAVE-ROW     PIC 9(4) COMP.
    02  WS-USER-SAVE-COL     PIC 9(4) COMP.
```

screen-item

specifies an elementary screen item defined in the Screen Section, or specifies an elementary screen item within a defined screen overlay item. The *screen-item* cannot include an OCCURS clause.

screen-area

specifies the name of a screen overlay item in a base screen definition. You include *screen-area* so the compiler can calculate the offset position of the overlay elementary screen item relative to the base screen.

NOT

specifies negation.

- Using either one of the NOT options causes the compiler to issue the same code and the statements following the IF ... WITHIN statement are executed only if the row and column values in *data-item* are not the same as the defined row and volume values in *screen-item* in the Screen Section.
- Using both of the NOT options cancels their effect and is equivalent to using neither one.

Note. For screen fields that wrap, only the first row is tested. If you test a screen field that wraps using the IF ... WITHIN statement, a warning is issued.

MOVE Statements

The MOVE statements transfer data from one data item to one or more other data items in accordance with editing rules. The forms of the MOVE statements are:

```
MOVE
MOVE CORRESPONDING
```

MOVE Statement

The MOVE statement transfers data from a data item to one or more data items.

```
MOVE data-name-1 TO { data-name-2 } [,] ...
```

data-name-1

is the sending item. The item can be an identifier or a literal. Any subscripting or indexing for *data-name-1* is evaluated only once, immediately before data is moved to the first receiving item.

data-name-2

is the receiving item. The item is an identifier. The following example:

```
MOVE item-1(b) TO item-2, item-3(b)
```

is equivalent to:

```
MOVE item-1(b) TO temp
MOVE temp TO item-2
MOVE temp TO item-3(b).
```

The following example illustrates a number of MOVE statements:

```
WORKING-STORAGE SECTION.
01  record-in.
    05  item-a    PIC X(5).
    05  item-b    PIC 99V99.
    77  temp1     PIC X(4).
    77  temp2     PIC X(8).
    77  temp3     PIC 9(5)V999.
    77  temp4     PIC 9V9.
PROCEDURE DIVISION.
begin-processing.
  MOVE item-a TO temp1.    (1)
  MOVE item-a TO temp2.    (2)
  MOVE item-b TO temp3.    (3)
  MOVE item-b TO temp4.    (4)
  MOVE SPACES TO record-in.
  MOVE ZEROS TO item-b.
```

(1) *Item-a* is truncated to fit *temp1*.

(2) The remainder of *temp2* is blank filled.

(3) The remainder of *temp3* is zero filled.

(4) If the value in *item-b* is greater than 9.9, this move causes the TCP to suspend the program unit with an arithmetic overflow error.

MOVE CORRESPONDING Statement

The MOVE CORRESPONDING statement moves selected data items of one group to corresponding data items of another group.

```
MOVE { CORR          } group-1 TO group-2
     { CORRESPONDING }
```

group-1

is the group name of sending data items.

group-2

is the group name of receiving data items.

group-1 and *group-2* must be defined in the Working-Storage Section or the Linkage Section, not in the Screen Section.

The following conventions apply to data items used with the CORRESPONDING phrase:

- *group-1* and *group-2* are the group names specified in the MOVE CORRESPONDING statement. Either or both of these data items can be described with the REDEFINES or OCCURS clause or be subordinate to items described with these clauses.
- Any subordinate data item of *group-1* or *group-2* which contains the REDEFINES, RENAMES, OCCURS or USAGE IS INDEX clause will not be moved.
- No data item can be defined with a level number 66, 77, or 88.

Subordinate data items in two different groups correspond to each other according to the following rules:

- Both data items must have the same data name.
- All possible qualifiers for the sending data item, not including the group name, must be identical to all possible qualifiers for the receiving data item, not including the receiving group name.
- At least one of the corresponding sending/receiving items must be elementary. The class of any corresponding pair of data items can differ.
- Any data item subordinate to a data item that is not eligible for correspondence is ignored.
- FILLER data items are ignored.

The following examples show corresponding items.

Example 1

All items in the following two groups correspond:

```
01  detail-in.                  01  report-line.
    05  social-security            03  social-security
    05  employee-name              03  FILLER
    05  address                    03  employee-name
        10  street                 03  FILLER
        10  city                   03  address
        10  state                      05  street
        10  zip-code                   05  FILLER
                                       05  city
                                       05  FILLER
                                       05  state
                                       05  FILLER
                                       05  zip-code
```

The following sentence would fill in report-line:

```
MOVE CORRESPONDING detail-in TO report-line
```

Example 2

Only pencils items in the following groups correspond; even though all other elementary names are alike, they do not have the same qualifiers:

```
01  stock-items.                01  shelf-items.
    05  erasers                    05  pens
        10  gum                        07  felt-tip-pens
        10  pink                       07  ball-point-pens
        10  ink                        07  fountain-pens
    05  pencils                    05  eradicators
        10  mechanical                 10  ink
        10  non-mechanical             10  pink
    05  felt-tip-pens                  10  gum
    05  ball-point-pens            05  pencils
    05  fountain-pens                  10  mechanical
                                       10  non-mechanical
```

The following sentence would move only the data items of the pencils group:

```
MOVE CORRESPONDING stock-items TO shelf-items
```

MOVE Restrictions

Move operations between the following types of data items will cause a compilation error and should therefore not be attempted:

- An alphabetic data item or the figurative constant SPACE to a numeric data item
- A numeric literal, a numeric data item, or the figurative constant ZERO to an alphabetic data item
- A noninteger numeric literal or a noninteger numeric data item to an alphanumeric data item
- A numeric data item to a numeric data item that does not have at least the same number of positions to the left of the decimal position
- Numeric integer(s), numeric noninteger(s), and numeric edited data item to a data item that allows only double-byte (PIC N) data.
- A data item or literal that allows only double-byte (PIC N) data to a numeric integer, a numeric noninteger, or a numeric edited data item.

MOVE Conventions

Data is converted and stored according to the data category of the receiving field. The conventions are as follows:

- Alphanumeric or alphabetic receiving data item
  - Data is stored beginning at the leftmost position in the receiving field
  - If the data in the sending item is shorter, the data is filled with spaces in the receiving field according to the standard alignment rules described in Section 2, SCREEN COBOL Source Program
  - If the data in the sending item is longer, the data is truncated on the right to the length of the receiving field.
  - If the sending item is described as signed numeric, the operational sign is not moved to the receiving field; this applies whether the sign is a part of the data item or is a separate character.
- Numeric receiving item
  - Data is aligned by decimal point and is filled with zeros as necessary.
  - If the receiving field is signed and the sending field is signed, the sign is moved and converted; if the sending field is not signed, the value is signed as positive.
  - If the receiving field is not signed, the absolute value of the sending field data is moved.
  - If the sending field is alphanumeric, the value of the sending field is treated as an unsigned numeric integer.

Group moves are treated as alphanumeric to alphanumeric moves, with no data conversion. The receiving area is filled without regard to individual or subgroup items in either the sending or receiving items.

Table 6-4 summarizes MOVE conventions.

Table 6-4. MOVE Summary Table

	Category of Receiving Data Item				
Category of Sending Data Item	**Alpha-betic**	**Alpha-numeric**	**Alpha-numeric Edited**	**Numeric Integer, Numeric Non-integer, or Numeric Edited**	**Double-Byte Character**
Alphabetic	Yes	Yes	No	No	Yes *
Alphanumeric	Yes	Yes	No	Yes	Yes *
Alphanumeric Edited	Yes	No	No	No	Yes *
Numeric Integer	No	No	No	Yes	No
Numeric Noninteger	No	No	No	Yes	No
Numeric Edited	No	No	No	No	No
Double-Byte Character	No	Yes	Yes	No	Yes

* Such MOVE operations move string data byte by byte and no editing or conversion is done.

MULTIPLY Statements

The MULTIPLY statements multiply two or more numeric items and place the result in a specified data item. A multiply operation can easily produce a value that does not fit into the receiving field; when defining a receiving field, thought should be given to the size of that field.

MULTIPLY BY Statement

The MULTIPLY BY statement multiplies one numeric data item by one or more other numeric data items. The product replaces the value of each multiplier.

```
MULTIPLY value BY { multiplier } ,...
```

value

is the multiplicand, which is a numeric literal or an identifier of an elementary numeric data item.

multiplier

is the identifier of an elementary numeric data item. The result of the multiply operation is stored as the new value of *multiplier*. The sum of the number of digits in *value* and *multiplier* must not exceed 18.

MULTIPLY GIVING Statement

The MULTIPLY GIVING statement multiplies two numeric data items and stores the product in one or more other data items.

```
MULTIPLY value BY multiplier GIVING { result } ,...
```

value

is the multiplicand, which is a numeric literal or an elementary numeric data item.

multiplier

is a numeric literal or the identifier of an elementary numeric data item. The sum of the number of digits of *value* and *multiplier* must not exceed 18.

result

is the identifier of an elementary numeric data item into which the product is stored.

PERFORM Statements

The PERFORM statements execute one or more procedures in a program. When a single paragraph or section name is specified, control passes to the first statement of the paragraph or section; when execution of the paragraph or section completes, control passes to the PERFORM statement. If a group of paragraphs or procedures is specified, control passes to the first statement of the first paragraph or section; when execution of the last paragraph or section completes, control returns to the PERFORM statement.

The forms of the PERFORM statement are:

```
PERFORM
PERFORM TIMES
PERFORM UNTIL
PERFORM VARYING
PERFORM ONE
```

In each of these forms, two parameters, *proc-1* and *proc-2*, appear. *Proc-1* and *proc-2* have no special relationship; they represent a consecutive sequence of operations to be executed beginning at *proc-1* and ending with the execution of *proc-2*. GO TO and PERFORM statements can occur within the range of *proc-1* and *proc-2*. If two or more logical paths lead to the return point, *proc-2* could be a paragraph consisting of an EXIT statement, to which all of these paths must lead.

If control passes to these procedures by a means other than a PERFORM statement, control passes through the last statement of the procedure to the next executable statement as if no PERFORM statement referred to these procedures.

The range of a PERFORM statement is logically all those statements that are executed as a result of the PERFORM statement, through the transfer of control to the statement following the PERFORM statement. The range includes all statements executed as a result of a GO TO, PERFORM, or CALL statement in the range of the original PERFORM statement, as well as all statements in the Declaratives Section that might be executed. Statements in the range of a PERFORM are not required to appear consecutively.

If a sequence of statements referred to by a PERFORM statement includes another PERFORM statement, the sequence of procedures for the nested PERFORM must be either totally included in, or totally excluded from, the logical sequence referred to by the original PERFORM statement. Thus, an active PERFORM statement whose execution point begins within the range of another active PERFORM statement must not allow control to pass to the exit of the other active PERFORM statement. Furthermore, two or more such active PERFORM statements must not have a common exit.

PERFORM Statement

The PERFORM statement executes a procedure, or group of procedures as established by the THROUGH phrase, one time. When execution completes, control passes to the statement following the PERFORM statement.

```
PERFORM proc-1 [ { THROUGH } proc-2 ]
               [ { THRU    }        ]
```

proc-1 and *proc-2*

are the procedure paragraphs or sections to be executed.

The following example illustrates a PERFORM of one paragraph:

```
IF report-a
   PERFORM do-report-a.
```

The following example illustrates a PERFORM of several paragraphs:

```
IF reports
     PERFORM do-reports THRU do-reports-exit.
     :
do-reports.
     :
  (several paragraphs to create the reports)
     :
do-reports-exit.
  EXIT.
```

PERFORM TIMES Statement

The PERFORM TIMES statement executes a procedure, or group of procedures as established by the THROUGH phrase, a specified number of times. When the specified number of executions complete, control passes to the statement following the PERFORM TIMES statement.

```
PERFORM proc-1 [ { THROUGH } proc-2 ] count TIMES
               [ { THRU    }        ]
```

proc-1 and *proc-2*

are the procedure paragraphs or sections to be executed.

count

is an integer literal or the identifier of an integer data item. The procedure, or group of procedures, is executed as many times as the value of *count*.

The following example illustrates the PERFORM TIMES statement:

```
PERFORM list-transactions 2 TIMES.
```

PERFORM UNTIL Statement

The PERFORM UNTIL statement executes a procedure, or group of procedures as established by the THROUGH phrase, based on a condition. The condition is checked before each PERFORM cycle. When the condition is met, control passes to the statement following the PERFORM UNTIL statement.

```
PERFORM proc-1 [ { THROUGH } proc-2 ] UNTIL condition
               [ { THRU    }        ]
```

proc-1 and *proc-2*

are the procedure paragraphs or sections to be executed.

condition

is any conditional expression.

The following example illustrates the PERFORM UNTIL statement:

```
WORKING-STORAGE SECTION.
01  flag              PIC 9.
    88  bad                 VALUE 0.
    88  good                VALUE 1.
    88  no-more-adds        VALUE 1.
          :
PROCEDURE DIVISION.
          :
   PERFORM add-routine UNTIL no-more-adds.
          :
```

```
add-routine.
   MOVE 0 to flag.
          :
          : Once the add routine is successful, a 1 is moved
          : to flag; otherwise, flag remains 0.  As long as
          : flag is 0, the procedure is reexecuted.
          :
delete-routine.
```

PERFORM VARYING Statement

The PERFORM VARYING statement executes a procedure, or group of procedures as established by the THROUGH phrase, while varying a data item until specified conditions are true. When AFTER phrases are specified, the range is within a nested loop. The innermost loop is defined by the last AFTER phrase; the outermost loop is defined by the first set of parameters in the VARYING clause. When execution completes, control passes to the statement following the PERFORM VARYING statement.

```
PERFORM proc-1 [ { THROUGH } proc-2 ]
               [ { THRU    }        ]

 VARYING vary-1 FROM base-1 BY step-1 UNTIL cond-1

 [ AFTER vary-2 FROM base-2 BY step-2 UNTIL cond-2 ] ...
```

proc-1 and *proc-2*

are the procedure paragraphs or sections to be executed.

vary-1 and *vary-2*

are the identifiers of integer numeric data items.

base-1 and *base-2*

are integer numeric literals or identifiers of numeric data items.

step-1 and *step-2*

are integer numeric literals or identifiers of numeric data items. Their value must not be zero.

cond-1 and *cond-2*

are any conditional expressions.

The following example illustrates the PERFORM VARYING statement:

```
WORKING-STORAGE SECTION.
01  command-data.
    05  FILLER  PIC X(36)  VALUE "ADD     - ADD A NEW RECORD".
    05  FILLER  PIC X(36)  VALUE "DELETE  - DELETE A RECORD".
    :
```

```
01  command-table REDEFINES command-data.
    05 command-entry  PIC X(36)  OCCURS 10 TIMES.

77  no-of-commands    PIC 99    VALUE 9.
77  command-index     PIC 99    VALUE 1  COMP.

PROCEDURE DIVISION.
    :
    PERFORM list-commands VARYING command-index FROM 1 BY 1
        UNTIL command-index GREATER THAN no-of-commands.

list-commands.
    :
```

PERFORM ONE Statement

The PERFORM ONE statement executes just one procedure, or one group of procedures as established by the THROUGH phrase, as determined by the value of an identifier.

```
PERFORM ONE [ OF ] { proc-1 [ { THROUGH } proc-2 ] } ,...
                   {        [ { THRU    }        ] }

  DEPENDING [ ON ] identifier
```

proc-1 and *proc-2*

are the procedure paragraphs or sections to be executed; the maximum is 255 paragraphs.

identifier

is an integer numeric literal or the identifier of an integer data item. The value determines which procedure, or group of procedures, is to be performed.

Each procedure, or group of procedures, in the list is assigned an index value that indicates the relative position of the procedure, or group of procedures, within the list. The index values begin at 1 and increment by 1, up to 255; the list cannot contain more than 255 procedures or groups of procedures.

If the value of the identifier matches one of these indexes, the procedure, or group of procedures, with that index is executed. When execution completes, control passes to the statement following the PERFORM ONE statement. If the value of *identifier* does not match any procedure index, no procedures are executed.

The following example illustrates the PERFORM ONE statement:

```
PERFORM ONE OF
        A THRU B
        C
        D
    DEPENDING ON I.
```

PRINT SCREEN Statement

The PRINT SCREEN statement causes the current screen image to be printed on an attached or nonattached printer.

PRINT SCREEN cannot be used by programs communicating with intelligent devices.

```
PRINT SCREEN [ ON ERROR imperative-statement ]
```

`ON ERROR`

provides a point of control if an error occurs while attempting the print operation. The special register TERMINATION-STATUS is set with an error code indicating the type of error; the *`imperative-statement`* is then executed.

If the clause is omitted and an error occurs, default system action is taken.

`imperative-statement`

is the statement to be executed if an error is detected.

If an attached printer has been specified via the PATHCOM SET TERM command for the 6520 terminal, the screen image is directed to a printer attached directly to the terminal (the 6524 terminal includes an attached printer).

If an attached printer has been specified via the PATHCOM SET TERM command for the IBM3270 terminal, the screen image is directed to the device specified in the special register TERMINAL-PRINTER; this device must be attached to the same control unit as the terminal.

If a printer is not attached, printing occurs on the operating system file specified in the special register TERMINAL-PRINTER. Device types terminal, printer, and process are supported.

If a printer is not attached, a screen image that is printed with PRINT SCREEN can be different from the screen image that is displayed, because the print operation does not read the screen to form the screen image. Instead, the print operation forms the screen image by using the screen description contained in the SCREEN COBOL object file and the Working-Storage items associated with the screen.

The following items indicate how screen reconstruction from Working-Storage items can cause differences in the displayed and printed screens (for printers that are not attached):

- A screen field defined as HIDDEN in the Screen Section is not printed even though the field is changed to be displayed during execution. If you want to have a screen field initially hidden and subsequently printed, do not define the item as hidden. Turn the HIDDEN attribute on and off by using a TURN statement during processing.
- An unexpected value might be printed because the value in the Working-Storage item has never been displayed or contains data that is different from what is presently displayed on the screen.

- Direct displays, such as DISPLAY "XYZ" IN SCREEN-FIELD, are not printed because these displays have no effect on Working-Storage.
- If a screen field has an associated FROM field and a different associated TO field, an anomaly exists. The PRINT SCREEN statement resolves this by assigning the following precedence when selecting associated Working-Storage items:

```
highest -> USING association -> TO association
lowest  -> FROM  association
```

Table 6-5 lists the TERMINATION-STATUS error codes set by the PRINT SCREEN statement.

Table 6-5. PRINT SCREEN Statement Errors

TERMINATION-STATUS	Meaning
0	No error has occurred.
1	The base screen is not displayed. Default system action: The terminal is suspended without possibility of restart, and a message describing the error is logged to the log file.
2	The printer specification is invalid for the terminal type, or the printer device type is not supported: the IS-ATTACHED modifier of the PATHCOM SET TERM PRINTER command is specified for 6510 terminals; the IS-ATTACHED modifier is specified for IBM3270 and the printer is not attached to the same controller as the terminal; or a file having a device type other than a terminal, printer, or process has been specified. Default system action: The terminal is suspended without possibility of restart, and a message describing the error is logged to the log file.
3	The printer requires attention (for example, it is in NOT READY state). During I/O to the printing device, an operating system file error code indicating the device requires human intervention was returned. Default system action: If special register DIAGNOSTIC-ALLOWED is set toYES a diagnostic screen informing the terminal operator of the condition is displayed. If the terminal operator presses the 6510 or 6520 terminal F1 key (or equivalent IBM3270 key), the operator has corrected the condition; screen recovery is invoked. The copy is restarted from the beginning. If the special register DIAGNOSTIC-ALLOWED is set to NO, the terminal is suspended with the possibility of restart, and a message describing the error is logged to the log file.
4	A fatal error has occurred. During I/O to the device, an operating system file error code indicating a fatal error condition was returned. Default system action: The terminal is suspended with the possibility of restart, and a message describing the error is logged to the log file.

I/O Performed by the PRINT SCREEN Statement

The PRINT SCREEN I/O sequence begins with a top-of-form operation. Each screen line is written in a separate record; trailing blanks and trailing null values are suppressed. Printing starts with the line at the top of the screen and proceeds through the line at the bottom of the screen.

Diagnostic Screens

A diagnostic screen, which is described in Appendix A, Advisory Messages, can be displayed when an error occurs during a PRINT SCREEN sequence. An example of the default diagnostic screen is:

```
PATHWAY ERROR REPORT: 04MAY92,12:42

TERMINAL: TERM-1

PRINTER REQUIRES ATTENTION
  PRINTER: $LP
  PRESS F1 TO RETRY, F2 TO ABORT
```

IBM3270 Attached Printers

To permit a screen on an IBM3270 terminal to be printed, an input field must be declared starting at screen position 1,2. If a protected field is in this position, the screen is locked for screen copy operations and a PRINTER I/O ERROR (179) occurs.

The destination device of a PRINT SCREEN operation must have a device type of 10 and must use the CRT protocol of the AM3270 access method. Refer to the *Device-Specific Access Methods—AM3270/TR3271* manual for additional information.

RECEIVE UNSOLICITED MESSAGE Statement

The RECEIVE UNSOLICITED MESSAGE statement reads an unsolicited message into Working-Storage. The message is either read directly into Working-Storage or is mapped through the Message Section.

```
RECEIVE UNSOLICITED [ MESSAGE ]

[ CODE FIELD [ IS ] code-field ]

{ YIELDS rcv-message                                          }
{                                                             }
{ { CODE rcv-code [,rcv-code ]... YIELDS rcv-message }... }

[ TIMEOUT timeout-value ]

[ ON ERROR imperative-stmt ]
```

YIELDS *rcv-message*

identifies an 01 level message name in either the Working-Storage Section or the Message Section.

CODE FIELD [IS] *code-field*

defines the location, length, and data type of the *rcv-code* field in the unsolicited message. The absence of this clause causes the default:

- offset—0 bytes offset from beginning of the record
- length—2 bytes
- data type—COMPUTATIONAL numeric data item

You need to specify this clause when the location, length, and data type of the *rcv-code* field is other than the default.

The *code-field* parameter in this clause specifies a field name in the Working-Storage Section. The offset of the specified field from the beginning of its record dictates the offset of the *rcv-code* field from the beginning of the unsolicited message.

The length of the specified field dictates the length of the *rcv-code* field.

The data type of the specified field dictates the data type of the *rcv-code* field.

CODE *rcv-code*

identifies a literal or a data item that specifies which unsolicited message is expected.

The position of the *rcv-code* in the CODE *rcv-code* clause corresponds to a TERMINATION-STATUS value. One or more *rcv-code* values can be associated with each unsolicited message.

A nonnumeric literal must be enclosed within quotation marks.

TIMEOUT *timeout-value*

specifies a time limit in seconds that the RECEIVE UNSOLICITED MESSAGE operation will wait for an unsolicited message. The *timeout-value* can be a numeric literal or a numeric data item; valid items are 0 through 32,767 seconds.

If the unsolicited message is not received in the specified number of seconds, TERMINATION-SUBSTATUS is set to 40, the operation is cancelled, and any ON ERROR clause is executed.

If this clause is omitted, there is no time limit.

ON ERROR *imperative-stmt*

specifies action to be taken if an error occurs in receiving the message. If an error occurs, *imperative-stmt* is executed. The TERMINATION-STATUS special register contains a value indicating the cause.

The following rules apply:

- The RECEIVE UNSOLICITED MESSAGE statement completes immediately if the unsolicited message queue for the SCREEN COBOL program contains a message; otherwise, the RECEIVE UNSOLICITED MESSAGE statement will wait for the arrival of an unsolicited message or a timeout, if a timeout was specified.
- The execution of the RECEIVE UNSOLICITED MESSAGE is the beginning step in processing any unsolicited message. The RECEIVE UNSOLICITED MESSAGE is executed in the following cases:

  1. The ESCAPE ON UNSOLICITED MESSAGE clause of an ACCEPT statement executes due to the arrival of an unsolicited message.
  2. A SEND MESSAGE statement is interrupted and its ESCAPE ON UNSOLICITED MESSAGE clause is executed.
  3. You test the PW-UNSOLICITED-MESSAGE-QUEUED special register and find it equals YES.
  4. The SCREEN COBOL program executes RECEIVE UNSOLICITED MESSAGE and waits for the arrival of an unsolicited message.

  For case 4, you might want to include a TIMEOUT clause if there is a possibility that the SCREEN COBOL program could wait indefinitely for an unsolicited message to arrive.

Consider the following SCREEN COBOL example with multiple receive codes:

```
PROCEDURE DIVISION.

UNSOLICITED-MESSAGE-HANDLER.
   RECEIVE UNSOLICITED MESSAGE
         CODE RC-1 YIELDS R-MSG-1
         CODE RC-2 YIELDS R-MSG-2
       TIMEOUT MAX-TIME
       ON ERROR GOTO ERROR-HANDLER.

   PERFORM ONE OF MSG-1-RCD
                  MSG-2-RCD
   DEPENDING ON TERMINATION-STATUS.

ERROR-HANDLER.
   IF TERMINATION-STATUS    =  1 AND
      TERMINATION-SUBSTATUS = 40
         PERFORM RCV-TIMED-OUT
   ELSE
         PERFORM ANALYZE-ERROR.
```

This code executes as follows:

1. The RECEIVE UNSOLICITED MESSAGE statement waits for the arrival of an unsolicited message. If the unsolicited message queue for this SCREEN COBOL program contains a message, the operation completes immediately; otherwise, the RECEIVE UNSOLICITED MESSAGE statement waits for the arrival of an unsolicited message or a timeout.

2. When the unsolicited message arrives, the RECEIVE UNSOLICITED MESSAGE statement moves the message to R-MSG-1 or R-MSG-2 depending on the receive code value. The statement also moves a number 1 or 2 into the TERMINATION-STATUS register depending on the position of the receive code in the statement. If there are no errors and the RECEIVE UNSOLICITED MESSAGE statement does not time out, the special register TERMINATION-STATUS is set as follows:

```
RECEIVE CODE              TERMINATION-STATUS

   RC-1                              1
   RC-2                              2
```

3. If the RECEIVE UNSOLICITED MESSAGE statement times out or there is an error, the statement sets the TERMINATION-STATUS and TERMINATION-SUBSTATUS registers accordingly and performs the procedure ERROR-HANDLER.

   Refer to Table 6-5 for an explanation of the error numbers contained in TERMINATION-STATUS.

RECONNECT MODEM Statement

The RECONNECT MODEM statement gives a SCREEN COBOL program control of the connection to a Pathway/iTS terminal or intelligent device across a dial-in switched line (a standard communication line used by the public telephone system). Pathway/iTS does not support a dial-out capability over a switched line.

```
RECONNECT MODEM
```

If the connection to the Pathway/iTS terminal or to the intelligent device is over a switched line, the RECONNECT MODEM statement breaks the connection with the SCREEN COBOL program and causes the program to wait for another incoming call. After the next incoming call completes connection to the terminal or device, the SCREEN COBOL program resumes execution at the next program instruction.

If a RECONNECT MODEM statement is executed but the Pathway/iTS terminal or device is not connected over a switched line, the program resumes immediately at the next program instruction.

After a RECONNECT MODEM statement is executed, all terminal screen definitions are lost. A DISPLAY BASE statement must precede the next screen operation.

RECONNECT MODEM lets a SCREEN COBOL program perform the following operations for Pathway/iTS terminals or intelligent devices connected over switched lines:

- Disconnect the terminal or device at the end of a session (the caller logs off)
- Recover from a modem error (an accidental disconnection), and wait for the next terminal or device to call

The SCREEN COBOL program must be in a consistent state when accessed by an incoming call. Initialize local variables and complete previous transactions before executing RECONNECT MODEM.

The RECONNECT MODEM statement causes a full context checkpoint. If Pathway/iTS is running under TMF and a terminal or intelligent device is in transaction mode, this statement backs out the current transaction and suspends the terminal or device so that it cannot be resumed. If an ABORT-TRANSACTION statement precedes the RECONNECT MODEM statement, Pathway/iTS attempts to resume communication with the terminal or device after a modem error.

The following example illustrates the RECONNECT MODEM statement:

```
 START-PROGRAM.
     CALL SEARCH-PROGRAM ON ERROR GO TO VERIFY-RECONNECT.
     RECONNECT MODEM.
     GO TO START-PROGRAM.

 VERIFY-RECONNECT.
     IF TERMINATION-STATUS IS = 18 AND
        TERMINATION-SUBSTATUS IS = 140
*  This is a modem error - return to a consistent state
*  and wait for the next terminal caller.
        DELAY 10
        RECONNECT MODEM
        GO TO START-PROGRAM.
*  Processes other error conditions.

     DISPLAY BASE SEARCH-SCREEN.

        or

     SEND MESSAGE REPLY YIELDS MESSAGE-IN.

      :
```

REPLY TO UNSOLICITED MESSAGE Statement

The REPLY TO UNSOLICITED MESSAGE statement sends a reply to a message previously received by a RECEIVE UNSOLICITED MESSAGE statement. After a RECEIVE UNSOLICITED MESSAGE statement is issued, all other RECEIVE UNSOLICITED MESSAGE statements are rejected as errors and all ESCAPE ON UNSOLICITED MESSAGE clauses are disallowed until a REPLY TO UNSOLICITED MESSAGE statement is executed.

```
REPLY [ TO ] UNSOLICITED [ MESSAGE ] [ WITH ] reply-message

  [ ON ERROR imperative-stmt ]
```

reply-message

identifies an 01 level message field in the Message Section or an 01 or 77 level data item in the Working-Storage Section. The *reply-message* contains data to be sent in response to a previously received unsolicited message.

ON ERROR *imperative-stmt*

specifies action to be taken should an error occur in sending the message. If an error occurs, *imperative-stmt* is executed. The TERMINATION-STATUS special register contains a value indicating the cause.

Consider the following SCREEN COBOL example that uses the REPLY TO UNSOLICITED MESSAGE statement:

```
PROCEDURE DIVISION.

MESSAGE-RESPONSE.
   REPLY TO UNSOLICITED MESSAGE WITH MY-REPLY
         ON ERROR PERFORM ERROR-HANDLER.

ERROR-HANDLER.
   IF TERMINATION-STATUS    =  1
         PERFORM ANALYZE-GUARDIAN-ERROR
   ELSE
         PERFORM ANALYZE-ERROR.
```

The REPLY TO UNSOLICITED MESSAGE statement executes as follows:

1. The statement sends the contents of MY-REPLY in response to the last unsolicited message.
2. If there is an error, ERROR-HANDLER is performed. If there is a file system error, the TERMINATION-STATUS register contains a 1 and the TERMINATION-SUBSTATUS register contains the error number; otherwise, TERMINATION-STATUS contains the error number.

   Refer to later in this section for an explanation of the error numbers contained in TERMINATION-STATUS.

RESET Statement

The RESET statement restores the display attributes and the data of screen fields to the compile-time definition. The statement restores only the terminal display, not the internal data.

RESET cannot be used by programs communicating with intelligent devices.

```
RESET [ TEMP      ] [ ATTR ] { screen-identifier } ,...
      [ TEMPORARY ] [ DATA ]

  [ DEPENDING [ ON ] identifier ]
  [ SHADOWED                    ]
```

TEMP or TEMPORARY

specifies that the selected fields are to be reset only if they have received their current values or attributes from a DISPLAY TEMP or TURN TEMP statement.

Note. During execution of a SCREEN COBOL program, the TCP controls the MDTs (modified data tag) in the same way it controls display attributes; with two important exceptions:

- When a TURN TEMP statement selects an input field for changing display attributes, the MDT bit is always set.
- When a RESET TEMP statement selects an input field for resetting of attributes, the MDT bit is set, regardless of the initial MDT attribute of the field.

These two exceptions apply only to the TURN and RESET statements that have the TEMP modifier. Note also that the field's MDT bit is not reset after the completion of the ACCEPT statement. Once the MDT bit is set, it stays set until the next DISPLAY BASE, TURN, RESET, or CLEAR INPUT operation.

ATTR

resets the display attributes of the selected fields to the value specified in the screen definition.

DATA

resets the characters displayed in the selected fields to the value specified in the VALUE field-characteristic clause of the field. If a value is not specified, the standard fill character fills the field.

If neither ATTR nor DATA is specified, both the attributes and data of the selected fields are reset to initial values.

For a terminal operating in conversational mode, RESET DATA has no effect; you can specify either RESET ATTR or RESET TEMP ATTR to reset the display attributes of fields on conversational-mode terminals.

screen-identifier

specifies the fields to be reset; the maximum is 127 screen fields per RESET statement. Each *screen-identifier* can name an entire screen, a screen group, or an elementary item of any base or overlay screen that is currently displayed. If *screen-identifier* is a group, all subordinate elementary items that have a TO, FROM, or USING clause in their definitions are reset.

DEPENDING ON *identifier*

selects zero or one *screen-identifier* from the list. The statement whose position in the *screen-identifier* list is the same as the value in *identifier* is selected. If the value in *identifier* is less than 1 or greater than the number of screen identifiers, no *screen-identifier* is selected.

SHADOWED

selects from the *screen-identifier* list only those fields that have SHADOWED items in which the SELECT bit is set; fields that do not have SHADOWED items are not selected.

Note. If neither the DEPENDING ON modifier nor the SHADOWED modifier is specified, all fields in the *screen-identifier* list are selected.

When the RESET statement is executed, the attributes and data of the selected fields are reset to their initial values.

RESET does not cause a physical write to the screen but causes data to be written to the terminal buffer. A physical write to the screen occurs when one of the following events occurs:

- TERMBUF fills up
- Execution of one of the following statements:

  ```
  ACCEPT
  BEGIN-TRANSACTION
  END-TRANSACTION
  RESTART-TRANSACTION
  ABORT-TRANSACTION
  CALL
  CHECKPOINT
  DELAY
  EXIT PROGRAM
  PRINT SCREEN
  SEND
  ```

- Execution of DISPLAY for a conversational terminal

The DEPENDING ON clause for the RESET statement is analogous to the DEPENDING ON clause for the PERFORM ONE statement. The following example illustrates this.

```
WORKING-STORAGE SECTION.
    :
77  ws-screen-status    PIC 9(4)    COMP VALUE 1.
01  ws-table.
    05  ws-table-item   PIC X(10)
                        OCCURS 4 TIMES.

SCREEN SECTION.
    :
01  MENU1 SIZE 24, 80.
    05 screen-table   at 3, 4
                       PIC X(10)
                       OCCURS ON 4 LINES SKIPPING 1
                       VALUE "0000000000"
                       USING ws-table-item.

PROCEDURE DIVISION.
    :
```

```
BODY-PARAGRAPH.
    :
  DISPLAY BASE MENU1.
  TURN REVERSE IN MENU1.
  DISPLAY MENU1.
    :
  RESET SCREEN-TABLE(1),
        SCREEN-TABLE(2),
        SCREEN-TABLE(3),
        SCREEN-TABLE(4),
  DEPENDING ON WS-SCREEN-STATUS.
```

If WS-SCREEN-STATUS equals 1, the displayed value and video attribute for SCREEN-TABLE(1) are reset to the states declared in the screen definition. If WS-SCREEN-STATUS equals 2, the displayed value and video attribute for SCREEN-TABLE(2) are reset to the states declared in the screen definition, and so on. It is not considered erroneous if WS-SCREEN-STATUS < 1 or WS-SCREEN-STATUS > 3. Control just falls through (execution continues with the next statement) and no screen field is reset.

RESTART-TRANSACTION Statement

The RESTART-TRANSACTION statement restarts the transaction of a terminal operating in transaction mode. Transaction mode is an operating mode in which Pathway servers that are configured to run under the Transaction Management Facility (TMF) can lock and update audited files.

```
RESTART-TRANSACTION
```

Execution of this statement indicates the current attempt to perform the transaction failed because a transient problem occurred.

The statement requests TMF to back out any updates made on a database during this transaction; terminal execution resumes at the BEGIN-TRANSACTION statement. TMF assigns a new transaction-ID to the transaction; the TCP marks the screen for screen recovery and increments by 1 the special register RESTART-COUNTER. The special register TERMINATION-STATUS remains at 1 (which indicates that the transaction is started or restarted). Working-storage items are restored to the values they had at execution of BEGIN-TRANSACTION. If the BEGIN-TRANSACTION statement includes the ON ERROR phrase, the ON ERROR branch is executed.

The execution of this statement can cause suspension of a terminal for a pending abort for two reasons:

- The terminal is not in transaction mode when this statement executes.
- A fatal error occurs while attempting to back out the updates made on the database.

SCROLL Statement

The SCROLL statement moves the contents of an overlay area up or down. This statement can be used only with the 6510 terminal; it cannot be used for communicating with other terminals or with intelligent devices.

```
SCROLL { UP   } overlay-area-name
       { DOWN }
```

UP

moves the data displayed in the overlay area of the screen up one line toward the top of the screen. A blank line appears at the bottom of the overlay area, and the top line in the overlay area is lost.

DOWN

moves the data displayed in the overlay area of the screen down one line toward the bottom of the overlay. A blank line appears at the top of the overlay area, and the last line in the overlay area is lost.

overlay-area-name

is the name of the screen overlay area. The overlay screen associated with the area can contain only output or literal fields. Literal fields are displayed only when the overlay screen is initially displayed in the area.

SEND Statement

The SEND statement declares the data structure associated with each valid reply code value. The SEND statement sends a transaction request message to a server process and receives a reply from that server process. The SEND statement includes the message and a list of reply specifications.

In processing the SEND statement, the TCP retains the reply code values to use when the server sends the reply. Upon receipt of a reply, the TCP compares the reply code value to the list of reply code values and determines which reply was received and, consequently, determines the structure of the data. The TCP then copies the reply into the SCREEN COBOL program.

```
SEND [ identifier-1 ] ,... TO server-class-name

[ UNDER PATHWAY pathmon-name ]

[ AT SYSTEM system-name ]

[ REPLY { CODE { reply-code-value } ...
      YIELDS [ VARYING ] {identifier-2 } ... } ...
      CODE OTHER YIELDS [ VARYING ] { identifier-2 } ...

[ ON ERROR imperative-statement ]
```

identifier-1

is a data item to be sent to the server. The data item represented by this identifier cannot exceed 32000 bytes. If *identifier-1* is a variable-length data item, the SEND statement sends only the currently defined occurrence (a variable-length data item is an item defined with an OCCURS DEPENDING ON clause).

If this parameter is omitted, zero bytes are sent to the server. A reply will still be returned from the server.

server-class-name

identifies the server class for which a message is intended. The *server-class-name* can be a nonnumeric literal or a data item. The size of the field containing *server-class-name* can be 1 through 15 characters.

This is the logical server class name used in the PATHCOM ADD SERVER command.

pathmon-name

is the name of the PATHMON process that controls the links to the server class named in the *server-class-name* parameter. The *pathmon-name* can be a nonnumeric literal or a data item. The field containing *pathmon-name* can have up to 15 characters, but the TCP passes only the first 5 characters in network communications.

A SEND statement directed through an external PATHMON must specify a valid network process name. The value specified for *pathmon-name* must begin with a $ and can be followed by 1 to 4 alphanumeric characters.

If this parameter is omitted, the PATHMON that controls the server class is assumed to have the same name as the PATHMON process that controls the TCP.

system-name

is the name of the Compaq *NonStop™ Himalaya* system on which the named PATHMON process is running. The *system-name* can be a nonnumeric literal or a data item. The field containing *system-name* can have up to 15 characters, but the TCP passes only the first 8 characters in network communications.

The value specified for *system-name* must be a valid network system name that begins with a \ and can be followed by 1 to 7 alphanumeric characters. If this parameter is omitted, the NonStop™ Himalaya system name of the PATHMON that controls the server class is assumed to be the same as the system name of the PATHMON that controls the TCP.

reply-code-value

is an integer literal or integer data item that specifies an expected reply code from the server. The maximum number of reply code values associated with one YIELDS clause is 255.

TERMINATION-STATUS can be used to identify the actual reply code received as described later in this section.

`VARYING`

can be used to control error logging when variable-length replies are expected. When the ON ERROR clause is present, the keyword VARYING suppresses the logging of error 3115 (TRANSACTION REPLY SIZE INVALID) when the reply is shorter than the YIELDS buffer. Error 3115 is still logged whenever the reply length exceeds the length of the YIELDS buffer. The keyword VARYING has no effect unless the ON ERROR clause is present.

`identifier-2`

is a data name into which a portion of the contents of the reply message is to be placed.

`CODE OTHER`

is used to ensure that a match on the reply code will always occur, thus preventing the logging of error 3112 (REPLY NUMBER NOT KNOWN TO PROGRAM) when the actual reply code is not found in the list of expected reply codes. Note that the CODE OTHER clause can appear by itself when no explicit reply codes are expected.

The CODE OTHER clause must be the last CODE clause in the SEND verb; if it is followed by a CODE <value> clause, SCREEN COBOL will report error 616 ('CODE OTHER' MUST BE LAST STMT IN REPLY CODE LIST OF SEND).

Reply code values must not appear in the same CODE clause as the keyword OTHER. The sequence CODE 1 2 OTHER will produce error 44 (SYNTAX ERROR DETECTED AT TOKEN) and error 48 (PARSING RESUMED AT TOKEN); the sequence CODE OTHER 1 2 will produce the same error.

`ON ERROR`

provides a point of control if an error occurs in sending the message.

If this clause is omitted and an error is detected, standard system action is performed. Depending on the error, system action involves either waiting for a resource to become available or suspending execution of the program.

Errors that occur during execution of a SEND statement can be looked up by number in the TERMINATION-STATUS special register to determine the meaning and recommended action. For some errors, additional information is reported in the TERMINATION- SUBSTATUS special register.

At the end of this section is a list of the error numbers found in the TERMINATION-STATUS special register.

`imperative-statement`

is the statement to be executed if an error is detected.

Use of Reply Codes and Termination Status

A message containing a variable-length data item cannot be easily decomposed by a server written in COBOL; the only exception is when the variable-length data item is the last item of the message.

If a server is to process more than one type of message, a data item of the message should contain a field that identifies the type of transaction unless the content of the data itself determines the transaction type.

Specifying *reply-code-value* after the CODE keyword identifies the structure of the reply. When the send operation receives the reply from the server, the first two bytes are interpreted as a 16-bit integer. This code must match one of the CODE reply code values. The entire reply is then distributed to the items in the *identifier-2* list associated with *reply-code-value*. The special register TERMINATION-STATUS is set to a number corresponding to the position of the particular reply code value in the list, providing the TCP did not find an error while executing the SEND statement.

Each *reply-code-value* corresponds to a unique number setting for TERMINATION-STATUS whether or not a reply code value yields the same Working-Storage data item. If there is no match or if the reply message data does not exactly fill the data items in the *identifier-2* list, an error is indicated.

In the SEND statement, the position of a reply code affects the value set for the TERMINATION-STATUS special register as illustrated in the following example:

```
SEND HEADER, LASTNAME OF EMP-REC TO "PERS-DEPT"
  REPLY CODE 1, 21, 31 YIELDS R-CODE, NEW-SALARY
        CODE 2, 42, 62 YIELDS NEW-RATE, STOCK-OPTION, BENEFIT
        CODE 0, 200    YIELDS TERMINATION-NOTICE
  ON ERROR PERFORM SERVER-LIST.
```

In this example, the positions of the reply codes cause these corresponding values to be set for TERMINATION-STATUS:

Reply Code	TERMINATION-STATUS
1	1
21	2
31	3
2	4
42	5
62	6
0	7
200	8

Consider the following example of the SEND statement:

```
77  YEARLY-REVIEW PIC 999 VALUE 3.
    :
    MOVE YEARLY-REVIEW TO TRANSCODE OF HEADER.
    SEND HEADER, LASTNAME OF PERSONAL-REC TO "SALARY-UPDATE"
         REPLY CODE 1 YIELDS R-CODE, NEW-SALARY-CODE
               CODE 2 YIELDS R-CODE, NEW-SALARY, STOCK-OPTION
               CODE 0 YIELDS R-CODE, TERMINATION-NOTICE
         ON ERROR PERFORM SERVER-DUMB.
```

The example is executed as follows:

1. The transaction message is constructed using the values of HEADER and LASTNAME from the SCREEN COBOL program data area. This message is sent to a server process of the server class SALARY-UPDATE, and the requester waits for a reply.
2. When the reply arrives, the reply is identified and moved into NEW-SALARY-CODE, NEW-SALARY and STOCK-OPTION, or TERMINATION-NOTICE, depending on the reply code. The number moved into special register TERMINATION-STATUS will be 1, 2, or 3, depending on the reply code received from the server.
3. The ON ERROR clause takes special action if a problem occurs in sending the message. The possible problems include a freeze on the server class, the unavailability of an appropriate server, and an unrecognizable reply from the server. If such a condition arises, TERMINATION-STATUS is set to a value indicating the type of error, and the imperative statement PERFORM SERVER-DUMB is executed.
4. If the ON ERROR clause had not been included and an error occurred, the standard system action would be performed (see Sending to an External PATHMON Process on page 6-76 for a list of SEND ERROR numbers)

Variable Reply Length

The ON ERROR clause prevents the requester from being suspended when the reply length does not match the YIELDS buffer length. In this case, when the TCP executes the ON ERROR clause, it aslo sets the SCREEN COBOL special registers TERMINATION-STATUS and TERMINATION-SUBSTATUS as follows:

```
TERMINATION-STATUS    =  11
TERMINATION-SUBSTATUS =  min (<rcvlen>,
                         1 + max(<sndlen>,<maxyldlen>) )
```

where <rcvlen> = actual length of message from server, <sndlen> = length of message sent to server, <maxyldlen> = longest total reply area of any YIELDS clause in this SEND verb.

For a SEND the TCP allocates a WRITEREAD buffer of length 1 + max(<sndlen>,<maxyldlen>). The value of TERMINATION-SUBSTATUS depends on the length of the actual reply, the length of the YIELDS buffer, and the length of the WRITEREAD buffer as follows:

- Case 1: server reply shorter than YIELDS buffer. In this case TERMINATION-SUBSTATUS = server reply length.
- Case 2: server reply length = YIELDS buffer length. In this case TERMINATION-SUBSTATUS is undefined. This case does not cause the TCP to invoke the ON ERROR clause.
- Case 3: server reply longer than YIELDS buffer but shorter than WRITEREAD buffer. In this case TERMINATION-SUBSTATUS = server reply length.
- Case 4: server reply longer than YIELDS buffer but = WRITEREAD buffer length. In this case TERMINATION-SUBSTATUS = server reply length.
- Case 5: server reply longer than YIELDS buffer and longer than WRITEREAD buffer. In this case TERMINATION-SUBSTATUS = WRITEREAD buffer length.

Unspecified Reply Codes

In the following example a reply code other than 1 or 2 (in the first two bytes of the reply) causes error 3112 (REPLY NUMBER NOT KNOWN TO PROGRAM) to be logged, and causes the terminal to be suspended. If a reply code of 1 or 2 is received, but the reply length does not match the length of rply-1 or rply-2, respectively, error 3115 (TRANSACTION REPLY SIZE INVALID) is logged, and the terminal is suspended because there is no ON ERROR clause.

```
SAMPLE-PARAGRAPH-1.
   SEND msg-a TO srvr-x
   REPLY CODE  1  YIELDS rply-1
         CODE  2  YIELDS rply-2.
```

Occasionally an application may choose not to specify all reply codes. For example, the following requester accepts any value in the first two bytes of the reply without logging error 3112.

```
PERFORM SAMPLE-PARAGRAPH-2.
   :
SAMPLE-PARAGRAPH-2.
   SEND msg-a TO srvr-x
   REPLY CODE OTHER YIELDS VARYING rply-a
      ON ERROR
        PERFORM ERR-FILTER.
ERR-FILTER.
   IF TERMINATION-STATUS = 11 PERFORM var-len-reply-rcd
   ELSE                       PERFORM genuine-reply-error.
VAR-LEN-REPLY-RCD.
   MOVE TERMINATION-SUBSTATUS TO rply-length.
GENUINE-REPLY-ERROR.
   DISPLAY TERMINATION-STATUS.
```

This example is executed as follows:

1. Regardless of the actual reply code, if the reply length equals the length of the Working-Storage item rply-a, the reply is stored in rply-a, and the program branches back to the paragraph that performed SAMPLE-PARAGRAPH-2.
2. If the reply is shorter than rply-a, the reply is stored in rply-a, the actual length of the reply is stored in the special register TERMINATION-SUBSTATUS, and the TERMINATION-STATUS value of 11 causes the program to branch to the paragraph VAR-LEN-REPLY-RCD.
3. If the reply is longer than rply-a, the reply is truncated to the length of rply-a and stored in rply-a, TERMINATION-STATUS is set to 11, TERMINATION-SUBSTATUS is set to the actual reply length up to a maximum of 1 + max(len(msg-a),len(rply-a)), an error 3115 is logged, and the program branches to VAR-LEN-REPLY-RCD.

Additional Information on Variable Reply Length

In the following example, the reply is received into two Working-Storage items. The first two bytes are stored into a common reply code, while the rest are stored into a reply-specific buffer. The common reply code identifies which buffer contains the reply, in the event that a length mismatch sets TERMINATION-STATUS to 11.

This requester accepts only a reply code of 15 or 37; any other reply code causes error 3112 to be logged.

```
PERFORM SAMPLE-PARAGRAPH-3.
   :
SAMPLE-PARAGRAPH-3.
    MOVE no-err TO err-flag.
    MOVE 0 TO rp-len.
    PERFORM send-paragraph.

    IF err-flag = no-err
       PERFORM ONE OF
          set-rp-1-len
          set-rp-2-len
       DEPENDING ON termination-status
       PERFORM good-reply-paragraph
    ELSE IF err-flag = length-mismatch
            MOVE termination-substatus TO rp-len
* if excess length, truncate to buffer length
            PERFORM adjust-rp-len
* subtract 2 bytes for reply code
            SUBTRACT 2 FROM rp-len
         ELSE IF err-flag = genuine-err
                 PERFORM genuine-reply-err-paragraph.

 SEND-PARAGRAPH.
    SEND msg-a TO srvr-x
    REPLY CODE 15    YIELDS VARYING rp-cd rp-buf-1
          CODE 37    YIELDS VARYING rp-cd rp-buf-2
       ON ERROR PERFORM err-paragraph.
```

```
ERR-PARAGRAPH.
   IF TERMINATION-STATUS = 11
      MOVE length-mismatch TO err-flag
   ELSE
      MOVE genuine-err TO err-flag.

ADJUST-RP-LEN.
   IF      rp-cd = 15 PERFORM set-rp-1-len
   ELSE IF rp-cd = 37 PERFORM set-rp-2-len.

SET-RP-1-LEN.
   IF rp-len > max-rp-1-len
      MOVE max-rp-1-len TO rp-len.

SET-RP-2-LEN.
   IF rp-len > max-rp-2-len
      MOVE max-rp-2-len TO rp-len.

GOOD-REPLY-PARAGRAPH.
   IF rp-cd = 15
      PERFORM rp-1-paragraph
   ELSE IF rp-cd = 37
           PERFORM rp-2-paragraph
        ELSE
           PERFORM logic-err-paragraph.
```

The example is executed as follows, assuming that the REPLY CODE is either 15 or 37:

1. If the reply length matches the length of the corresponding YIELDS items, the reply is stored in the corresponding items, and the ON ERROR branch is not taken.
2. If the reply length is less than the length of the corresponding YIELDS items, the reply is stored in the corresponding items, the length of the reply is stored in the special register TERMINATION-SUBSTATUS, and the program performs ERR-PARAGRAPH with TERMINATION-STATUS = 11. No error 3115 is logged because VARYING was specified.
3. If the reply is longer than the corresponding YIELDS items, the reply is truncated and stored in the corresponding items, TERMINATION-STATUS is set to 11, TERMINATION-SUBSTATUS is set to the actual reply length up to a maximum of 1 + max(len(msg-a),len(rp-cd)+len(rp-buf-1),len(rp-cd)+len(rp-buf-2)), and the program performs ERR-PARAGRAPH. In this case an error 3115 is logged.

Sending to an External PATHMON Process

The following program example illustrates the two ways you can use the SEND statement to access a server class controlled by an external PATHMON process (a PATHMON process in a different PATHMON environment than that of the requesting TCP).

```
DATA DIVISION.
WORKING-STORAGE SECTION.
    :
01  WS-DEFAULT-NAMES.
    05 WS-DEFAULT-SERVER       PIC X(15) VALUE "SERV-1".
    05 WS-DEFAULT-PATHMON      PIC X(5) VALUE "$PWT".
    05 WS-DEFAULT-SYSTEM       PIC X(8) VALUE "\TS".
    :
01  WS-SCRN1-FIELDS.
    05 WS-SERV-NAME            PIC X(15) VALUE " ".
    05 WS-SCRN-PATHMON         PIC X(5) VALUE " ".
    05 WS-SCRN-SYSTEM          PIC X(8) VALUE " ".
    :

PROCEDURE DIVISION.
    :
    :
    SEND MSGID, EMPLOYEE-REC TO "SERV-1"
         UNDER PATHWAY "$PWT"
         AT SYSTEM "\TS"
         REPLY CODE 1 YIELDS R-CODE, EMPLOYEE-REC
               CODE 2 YIELDS R-CODE, HIRE-DATE
         ON ERROR PERFORM 899-SEND-ERROR.
         :
         :
    MOVE WS-DEFAULT-SERVER TO WS-SERV-NAME.
    MOVE WS-DEFAULT-PATHMON TO WS-SCRN-PATHMON.
    MOVE WS-DEFAULT-SYSTEM TO WS-SCRN-SYSTEM.
         :
         :
    SEND MSGID, EMP-TRANSFER TO WS-SERV-NAME
         UNDER PATHWAY WS-SCRN-PATHMON
         AT SYSTEM WS-SCRN-SYSTEM
         REPLY CODE 1 YIELDS R-CODE, EMPLOYEE-LOC
               CODE 2 YIELDS R-CODE,
               CODE 3 YIELDS R-CODE
         ON ERROR PERFORM 899-SEND-ERROR.
         :
```

The following list of messages contains an explanation of the error numbers in the TERMINATION-STATUS special register.

TERMINATION-STATUS 1

```
SERVER CLASS FROZEN
```

Cause. The server class to which the message is directed is frozen.

Action Without ON ERROR Clause. The system waits until the server class is thawed by execution of a PATHCOM THAW SERVER command, then continues with the processing of the SEND statement.

TERMINATION-STATUS 2

```
RESOURCE UNAVAILABLE
```

Cause. A free server class control block cannot be found. You should increase the value of the TCP configuration parameter MAXSERVERCLASSES.

Action Without ON ERROR Clause. The system waits until the resource is available, then continues with the processing of the SEND statement.

TERMINATION-STATUS 3

```
RESOURCE UNAVAILABLE
```

Cause. A free server process control block cannot be found. You should increase the value of the TCP configuration parameter MAXSERVERPROCESSES.

Action Without ON ERROR Clause. The system waits until the resource is available, then continues with the processing of the SEND statement.

TERMINATION-STATUS 4

```
LINK DENIED
```

Cause. The request to PATHMON for a link to the server class has been denied for indeterminate reasons, and the TCP has no previously established links to the class.

Action Without ON ERROR Clause. The system periodically rerequests a link and, when successful, continues with the processing of the SEND statement.

TERMINATION-STATUS 5

```
SERVER CLASS UNDEFINED
```

Cause. The request to PATHMON for a link to the server class has been denied because no class of that name has been defined.

Action Without ON ERROR Clause. The system periodically rerequests a link. When the server class is added to the configuration, the system continues with the processing of the SEND statement.

TERMINATION-STATUS 6

```
ILLEGAL SERVER CLASS NAME
```

Cause. The value given for the name of the server class does not have the format of a valid server class name.

Action Without ON ERROR Clause. The system suspends the terminal with a fatal error.

TERMINATION-STATUS 7

```
MESSAGE TOO LARGE
```

Cause. The message to the server is larger than allowed by the TCP configuration.

TERMINATION-STATUS 8

```
MAXIMUM REPLY TOO LARGE
```

Cause. The size of one or more replies specified in the SEND statement is larger than allowed by the TCP configuration. The maximum number of bytes permitted for an outgoing SEND message or a server reply message is specified using the MAXREPLY parameter of the PATHCOM SET TCP command, which should be set to the larger of:

- The longest outgoing message from any SEND statement from any terminal controlled by the TCP
- The longest reply possible from any server replying to a SEND statement from any terminal controlled by the TCP

TERMINATION-STATUS 10

```
UNDEFINED REPLY
```

Cause. The reply code found in the reply from the server does not match any codes specified in the SEND statement.

Action Without ON ERROR Clause. The system suspends the terminal. If terminal execution is resumed when the operator issues a PATHCOM RESUME command, the SEND statement is retried. If the terminal is in transaction mode, the transaction is backed out and the terminal is suspended. If terminal execution is resumed, the transaction is restarted.

TERMINATION-STATUS 11

```
REPLY LENGTH INVALID
```

Cause. The length of the reply received from the server is not equal to the length specified by the selected YIELDS list.

Note. The TERMINATION-SUBSTATUS special register contains the following additional information: reply length received (in bytes).

TERMINATION-STATUS 12

```
I/O ERROR
```

Cause. A file system error occurred during the WRITEREAD to the server, or a timeout on the server occurred.

Note. The TERMINATION-SUBSTATUS special register contains the following additional information: file-system error number.

TERMINATION-STATUS 13

```
TRANSACTION MODE VIOLATION
```

Cause. A SEND to a server not using TMF was attempted while the terminal was in transaction mode.

Action Without ON ERROR Clause. The system suspends the terminal for pending abort.

TERMINATION-STATUS 14

```
NO PMCB AVAILABLE
```

Cause. The request requires the TCP to communicate with an external PATHMON process, but the maximum number of PATHMON processes the TCP can communicate with has been reached. The value specified in the SET TCP MAXPATHWAYS command sets this limit.

Action Without ON ERROR Clause. The system suspends the terminal and an error message is sent to the PATHMON log file.

TERMINATION-STATUS 15

```
UNDEFINED SYSTEM
```

Cause. The system name given is not known to the network.

TERMINATION-STATUS 16

```
ILLEGAL SYSTEM NAME
```

Cause. The value given for the system name does not have the correct format. (For example, the first character is not a \.)

TERMINATION-STATUS 17

```
ILLEGAL PATHMON NAME
```

Cause. The value given for the name of PATHMON process does not have the correct format. (For example, the first character is not a $.)

TERMINATION-STATUS 18

```
PATHMON I/O ERROR
```

Cause. An I/O error occurred during the OPEN or WRITEREAD message to an external PATHMON.

TERMINATION-STATUS 19

```
PATHMON I/O ERROR DUE TO LICENCING PROBLEMS
```

Cause. An external Pathmon process rejects the OPEN message due to licensing problems.

Note. The TERMINATION-SUBSTATUS special register contains the following additional information: file-system error number.

SEND MESSAGE Statement

The SEND MESSAGE statement sends a message to an intelligent device, receives a reply from that device, or both sends a message and receives a reply. The message data structure can be identified by either a message template in the Message Section or a level 01 data item in the Working-Storage Section.

```
SEND MESSAGE  { send-message                }
              { [ send-message ] reply-spec }


  [ ESCAPE ON UNSOLICITED [ MESSAGE ] ]

  [ USER [ CONVERSION ] numeric-literal ]

  [ TIMEOUT timeout-value ]

  [ ON ERROR imperative-statement ]


reply-spec syntax:

REPLY [ CODE FIELD [ IS ] code-field ]

{YIELDS  reply-message                                    }
{{CODE reply-code [,reply-code]... YIELDS reply-message}...}
```

send-message

specifies a message template in the Message Section or an 01 level data item in the Working-Storage Section. In either case, it specifies data to be sent to an intelligent device.

If you omit *send-message*, no data is sent, but a reply message is returned.

CODE FIELD [IS] *code-field*

defines the location, length, and data type of the *reply-code* field in the *reply-message*.

The absence of this clause causes the default to be used:

- Offset—0 bytes offset from beginning of the record
- Length—2 bytes
- Data type—COMPUTATIONAL numeric data item

You need to specify this clause when the location, length, and data type of the *reply-code* field is other than the default.

The *code-field* parameter in this clause specifies a field name in the Working-Storage Section.

The offset of the specified field from the beginning of its record dictates the offset of the *reply-code* field from the beginning of the *reply-message*.

The length of the specified field dictates the length of the *reply-code* field.

The data type of the specified field dictates the data type of the *reply-code* field.

The *reply-code* in all *reply-messages* must be the same length and have the same offset; reply codes need not have the same name.

YIELDS *reply-message*

specifies either a message template in the Message Section or an 01 level data item in Working-Storage. It describes the data expected from the intelligent device in reply to the SEND MESSAGE statement. A *reply-message* can be associated with one or more *reply-code* values.

If *reply-message* and *reply-code* are both omitted, no reply is expected.If only one *reply-message* is specified and *code-field* is omitted, any reply is expected to have the format of the *reply-message*.

CODE *reply-code*

specifies a literal or a data item that indicates which *reply-message* is expected.

No *reply-code* is needed if there is only one *reply-message*.

The position of the *reply-code* in the CODE *reply-code* clause corresponds to a TERMINATION-STATUS value. One or more *reply-code* values can be associated with each *reply-message*.

A nonnumeric literal must be enclosed within quotation marks.

ESCAPE ON UNSOLICITED [MESSAGE]

specifies that the SEND MESSAGE statement is to be aborted on the arrival of an unsolicited message. You detect the receipt of an unsolicited message by checking for the appropriate condition code value in TERMINATION-STATUS following the completion of the SEND MESSAGE statement. This is an accepted way of completing a SEND MESSAGE operation and is not handled in the ON ERROR clause.

The TERMINATION-SUBSTATUS special register has a value following a SEND MESSAGE that completes for an unsolicited message. This value specifies the state of the front-end process at the time that the unsolicited message was received.

TIMEOUT *timeout-value*

specifies a time limit in seconds for output to the intelligent device. The *timeout-value* can be a numeric literal or a numeric data item; valid values are 0 through 32,767 seconds.

When a timeout error occurs, it is logged to the Pathway log. If the TCP attribute SENDMSGTIMEOUT is set to OFF, only the first SENDMSGTIMEOUT timeout error is logged for the TCP. When the ON ERROR clause is used with a SEND

MESSAGE statement, only the first 3161 timeout encountered by each TCP configured under a PATHMON process is logged. An additional 3161 timeout is logged when the primary TCP fails and the backup TCP takes over and encounters the timeout. The TCPs filter out all other 3161 timeout errors. If the ON ERROR clause is not used and a 3161 timeout occurs, the corresponding terminal is suspended after the timeout is logged.

If the output does not complete in the specified number of seconds, TERMINATION- STATUS is set to 1, TERMINATION-SUBSTATUS is set to 40, the operation is cancelled, and any ON ERROR clause is executed.

When the SCREEN COBOL program that contains the SEND MESSAGE statement is communicating with devices using a queued (CONTROL 26) protocol, the TERMINATION-SUBSTATUS special register has a value following a SEND MESSAGE that completes for a TIMEOUT.

If this clause is omitted, there is no time limit.

`USER [ CONVERSION ] numeric-literal`

specifies a user conversion procedure associated with either a *send-message* or a *reply-message*. Refer to the *Compaq NonStop™ Pathway/iTS TCP and Terminal Programming Guide* for a detailed discussion of user conversion procedures.

`ON ERROR imperative-statement`

specifies action to be taken should an error occur in sending the message. If an error is detected, the *imperative-statement* is executed.

If ON ERROR is omitted and an error is detected, the system takes standard action. Depending on the error, the system either waits for a resource to become available or suspends execution of the program.

If ON ERROR is included in the SEND MESSAGE statement, TERMINATION-STATUS is set to a specified number.

If there are no errors, TERMINATION-STATUS contains the relative position of the matching reply. Thus, you should write error processing routines to execute only when an error occurs.

Appendix D, Errors for Message Section Statements, lists errors that can occur during execution of the SEND MESSAGE statement.

Note. Error messages can be placed in the log by the PATHMON process even if an ON ERROR clause is included in a SEND MESSAGE statement.

The following rules apply:

- A *send-message* or *reply-message* cannot exceed 32,000 bytes.
- *send-message* considerations:
  - If *send-message* identifies a message template in the Message Section, the data is obtained from the data item specified in the FROM or USING clause of

the message, is converted and edited according to the message definition, and is then sent to the intelligent device.

- If *send-message* identifies an 01 level data item in Working-Storage, the data is neither validated nor converted but is sent as is to the device, unless the USER CONVERSION clause is included.
- If *send-message* identifies a variable-length data item (described with an OCCURS DEPENDING ON clause), it sends only the number of characters up to and including the current number of occurrences.

- *reply-message* considerations:
  - If *reply-message* is the name of a message template, the value of the message is validated according to the message-field definition, converted to the format of the Working-Storage data item in the TO or USING clause, and then moved to that data item.
  - If *reply-message* is the name of a level 01 Working-Storage item, the data from the intelligent device is neither validated nor converted but is moved as is to the data item, unless you specify a USER CONVERSION clause.
  - All the *reply-messages* in any one SEND MESSAGE statement must have the same format: FIXED, VARYING1, VARYING2, DELIMITED, or FIXED-DELIMITED.
  - If there is only one *reply-message*, you need not specify a *reply-code*.
  - When the TCP receives a reply from the intelligent device, it checks for a *reply-code* in the message or data item. The *reply-code* value indicates which *reply-message* was returned.
  - The *reply-code* is located either at the location indicated by *code-field* or, if *code-field* is omitted, in the first two bytes of the message or data item.
  - Each *reply-code* is associated with a value in the special register TERMINATION-STATUS. TERMINATION-STATUS is set to a number corresponding to the position of each *reply-code* in the list of *reply-code* values, starting at position number 1.
  - A *reply-message* can be associated with more than one *reply-code*. Each *reply-code* corresponds to a unique setting for TERMINATION-STATUS regardless of whether the *reply-code* yields the same or a different *reply-message*.
  - If a *code-field* is specified, a *reply-message* is selected by comparing each *reply-code* with the *code-field* according to the standard SCREEN COBOL rules for equality (refer to Section 2, SCREEN COBOL Source Program). The test is performed exactly as if it were written:

    ```
    IF code-field EQUALS reply-code
    ```

- Execution considerations:
  - Pathway/iTS does not buffer multiple messages. Each message is transmitted to the intelligent device when SEND MESSAGE is executed.
  - SCREEN COBOL programs handle synchronization of messages between the program and the intelligent device. This means that the program must detect duplicate messages from the intelligent device.
  - Pathway/iTS does no checkpointing when SEND MESSAGE is executed.

  Consider the following SCREEN COBOL example with multiple reply messages:

```
DATA DIVISION.

WORKING-STORAGE SECTION.
77 INQUIRY  PIC X(12).
01 MESSAGE-REPLY-OK.
   03  REPLY-CODE-FIELD   PIC X(2).
   03  REPLY-OK           PIC X(130).
01 MESSAGE-REPLY-NOGOOD.
   03  REPLY-CODE-FIELD   PIC X(2).
   03  REPLY-NG           PIC X(14).

MESSAGE SECTION.
    01 INTELLIGENT-MESSAGE-INQ  PIC X(12) FROM INQUIRY.

PROCEDURE DIVISION.

SEND MESSAGE INTELLIGENT-MESSAGE-INQ
  REPLY CODE FIELD IS REPLY-CODE-FIELD OF MESSAGE-REPLY-OK
        CODE "A1", "B1", "C1"  YIELDS MESSAGE-REPLY-OK
        CODE "A2", "B2", "C2"  YIELDS MESSAGE-REPLY-NOGOOD
  ON-ERROR PERFORM DEVICE-DUMB.
```

  This SEND MESSAGE statement executes as follows:

  1. It constructs the message using the message template called INTELLIGENT-MESSAGE-INQ from the Working-Storage data item INQUIRY. It sends the message to the intelligent device or process and waits for a reply.
  2. When the reply arrives, it moves the reply either to MESSAGE-REPLY-OK or to MESSAGE-REPLY-NOGOOD, depending on the reply code value. It moves a number from 1 to 6 into the TERMINATION-STATUS register based on the position of the reply code in the SEND MESSAGE statement. In this example, the special register TERMINATION-STATUS is set as follows:

```
REPLY CODE       TERMINATION-STATUS

    A1                   1
    B1                   2
    C1                   3
    A2                   4
    B2                   5
    C2                   6
```

3. If the SEND MESSAGE statement detects an error, it sets TERMINATION-STATUS to a value indicating the type of error and performs the procedure DEVICE-DUMB.

   The reply code values could also be numeric. Consider the following example:

```
SEND MESSAGE INQUIRY
  REPLY CODE FIELD IS REPLY-CODE-FIELD OF MESSAGE-REPLY-OK
        CODE 1, 21, 31 YIELDS MESSAGE-REPLY-OK
        CODE 2, 42, 62 YIELDS MESSAGE-REPLY-NOGOOD
  ON ERROR PERFORM DEVICE-DUMB.
```

   In the example, the special register TERMINATION-STATUS is set as follows:

```
REPLY CODE      TERMINATION-STATUS

    1                 1
   21                 2
   31                 3
    2                 4
   42                 5
   62                 6
```

   Consider the following SCREEN COBOL example of processing the ESCAPE ON UNSOLICITED MESSAGE clause:

```
CONTINUE-ACTION.
  SEND MESSAGE REQUEST-MESSAGE
      REPLY CODE "AA" YIELDS reply-aa
            CODE "XX" YIELDS reply-xx
    ESCAPE ON UNSOLICITED MESSAGE
    TIMEOUT FIVE-MINUTES
    ON ERROR
      PERFORM ERROR-ANALYSIS.

* Normal reply processing:

  PERFORM ONE OF AA-RPLY-ACTION
                 XX-RPLY-ACTION
                 UNSOL-MSG-ARRIVED
  DEPENDING ON TERMINATION-STATUS.

  UNSOL-MSG-ARRIVED.
    PERFORM SAVE-OUTCOME.
    RECEIVE UNSOLICITED
            YIELDS UNSOL-STATS-REQST.
          .   .   .
    REPLY TO UNSOLICITED latest-stats

* Save the outcome of the SEND MESSAGE I/O.

SAVE-OUTCOME.
  IF TERMINATION-SUBSTATUS = 187
     MOVE OP-QUIESCENT TO OP-OUTCOME
  ELSE
     IF TERMINATION-SUBSTATUS = 188
```

```
      MOVE OP-BAD-SESSION TO OP-OUTCOME
   ELSE
      IF TERMINATION-SUBSTATUS = 189
            MOVE OP-WONT-QUIESCENT TO OP-OUTCOME.
```

To detect the arrival of an unsolicited message, the program must process the condition code value in the TERMINATION-STATUS special register. In this example, TERMINATION-STATUS is set as follows:

Action	TERMINATION-STATUS
Received reply message with reply code of AA	1
Received reply message with reply code of XX	2
Received unsolicited message	3

The value of the TERMINATION-SUBSTATUS special register indicates what state the front-end process was in at the time the unsolicited message was received.

The TERMINATION-SUBSTATUS values for the CONTROL 26 process interface are shared between the two types of allowable escapes: UNSOLICITED MESSAGE or TIMEOUT. The SEND MESSAGE statement can be used to communicate with devices using a queued (CONTROL 26) protocol, for example, devices connected by a SNAX/HLS front-end process interface.

When a SCREEN COBOL program is interrupted on a CONTROL 26 terminal, the TCP issues one of the TERMINATION-SUBSTATUS values listed in Table 6-6. The value of the TERMINATION-SUBSTATUS register depends on the reason for the interruption.

Table 6-6. TERMINATION-SUBSTATUS Values for SEND MESSAGE Statement

TERMINATION-SUBSTATUS	Meaning	With UNSOLICITED MESSAGE Clause	With TIMEOUT Clause
187	No data	Front-end process in quiescent state.	Front-end process in quiescent state.
		The SCREEN COBOL program unit processes the message, can resume the session, or take other action depending on the message.	The SCREEN COBOL program unit performs TIMEOUT processing and resumes the session.
188	Data lost	Data or context lost.	Data or context lost.
		After processing the unsolicited message, the SCREEN COBOL program unit must terminate the session with the front-end process and then try to start a new one.	The SCREEN COBOL program unit must terminate the session with the front-end process and then try to start a new one.
189	Data forthcoming	Operation in progress. Front-end process cannot be put into quiescent state.	Front-end process cannot be put into quiescent state.

Note. On completion of a SEND MESSAGE statement for an ESCAPE ON UNSOLICITED MESSAGE or a TIMEOUT, the values of 187, 188, or 189 appear in TERMINATION-SUBSTATUS with one exception: for TIMEOUT, the "no data" code appears in TERMINATION-SUBSTATUS as 40, for compatibility purposes.

SET Statement

The SET statement stores the position of the indicated screen field into the special register NEW-CURSOR. This value, which could be further modified by the program, is used at the beginning of the next ACCEPT statement to establish the position of the cursor on the screen. This value also can be examined by the program for some specific purpose.

The SET statement cannot be used by programs communicating with intelligent devices.

```
SET NEW-CURSOR AT { screen-identifier } ,...

  [ DEPENDING [ ON ] identifier ]
  [ SHADOWED                    ]
```

screen-identifier

specifies the screen fields whose positions are to be stored; maximum of 127 per SET statement. Each *screen-identifier* can name an entire screen, a screen group, or

an elementary input item of any base or overlay screen that is currently displayed. If *screen-identifier* is a group, all subordinate elementary items that have a TO, FROM, or USING phrase in their definitions are included in the reference. *screen-identifier* cannot be a subscripted item.

The default cursor position is the first screen field defined with a TO or USING clause for the current ACCEPT statement.

DEPENDING ON *identifier*

selects zero or one *screen-identifier* from the list. The statement whose position in the *screen-identifier* list is the same as the value in *identifier* is selected. If the value in *identifier* is less than 1 or greater than the number of *screen identifiers*, no *screen-identifier* is selected.

SHADOWED

selects from the *screen-identifier* list only those fields that have SHADOWED items in which the SELECT bit is set; fields that do not have SHADOWED items are not selected.

Note. If neither the DEPENDING ON modifier nor the SHADOWED modifier is specified, all fields in the *screen-identifier* list are selected.

The SET statement selects fields in sequence from top to bottom and left to right as the fields are positioned on the screen. The field having the lowest row (line) number is selected before a field with a higher row number. For fields in the same row, the field having the lowest column number is selected before the field with a higher column number.

The SET statement places the row and column numbers of the leftmost character of the first selected field into the special register NEW-CURSOR. The implied structure of NEW-CURSOR is:

```
01  NEW-CURSOR.
    02 NEW-CURSOR-ROW  PIC 9999 COMP.
    02 NEW-CURSOR-COL  PIC 9999 COMP.
```

If the value specified in the special register NEW-CURSOR is not a valid screen position when an accept operation begins, the cursor is positioned to the first unprotected field of the ACCEPT statement for a 65xx terminal or to the first field of the ACCEPT statement for an IBM 3270 terminal. (IBM 3270 terminals do not prevent cursor positioning at a protected field.)

After execution of an ACCEPT statement, the special register is set to zero; this sets the cursor position for the next ACCEPT statement. For 65xx terminals, the cursor position is at the first unprotected field of the ACCEPT statement. For IBM 3270 terminals, the cursor position is at the first field of the ACCEPT statement defined with a TO or USING clause.

If you do not want the cursor positioned at a protected field on an IBM3270 terminal, you can use a SET statement to position the cursor at an unprotected field.

The DEPENDING ON clause for the SET statement is analogous to the DEPENDING ON clause for the PERFORM ONE statement. The following example illustrates this.

```
WORKING-STORAGE SECTION.
    :
77  ws-screen-status    PIC 9(4)    COMP VALUE 1.
01  ws-fld1             PIC x(10).
01  ws-fld2             PIC x(10).
01  ws-fld3             PIC x(10).

SCREEN SECTION.
    :
01  MENU1 SIZE 24, 80.
    05 screen-fld1    at 4, 20
                      PIC X(10)
                      from fld1.
    05 screen-fld2    at 5, 40
                      PIC X(10)
                      from fld2.
    05 screen-fld3    at 6, 60
                      PIC X(10)
                      from fld3.

PROCEDURE DIVISION.
    :
BODY-PARAGRAPH.
    :
  DISPLAY BASE MENU1.
  DISPLAY MENU1.
  SET NEW-CURSOR AT SCREEN-FLD1,
                    SCREEN-FLD2,
                    SCREEN-FLD3,
  DEPENDING ON WS-SCREEN-STATUS.
  ACCEPT MENU1
    :
```

If WS-SCREEN-STATUS equals 1, the cursor is positioned at SCREEN-FLD1 upon execution of the ACCEPT statement. If WS-SCREEN-STATUS equals 2, the cursor is positioned at SCREEN-FLD2 upon execution of the ACCEPT statement, and so on. It is not considered erroneous if WS-SCREEN-STATUS < 1 or WS-SCREEN-STATUS > 3. Control just falls through (execution continues with the next statement) and the cursor is left at its default position.

SET MINIMUM-ATTR Statement

The SET MINIMUM-ATTR statement establishes the minimum level of support for highlight and outline display attributes.

```
SET MINIMUM-ATTR USING data-name.
```

data-name

specifies the Working-Storage group item template for highlight and outline display attributes.

The Working-Storage group item template for highlight and outline display attributes must be defined in Working-Storage as follows:

```
01 WS-MINIMUM-ATTRIBUTE.
   02 FILLER                          PIC 9(04) COMP.
   02 IBM-BASE-ATTR                   PIC 9(04) COMP.
   02 IBM-FIELD-REVERSE               PIC 9(04) COMP.
   02 IBM-FIELD-BLINK                 PIC 9(04) COMP.
   02 IBM-FIELD-UNDERLINE             PIC 9(04) COMP.
   02 IBM-FIELD-OUTLINE               PIC 9(04) COMP.
```

- IBM-BASE-ATTR specifies the set of field attributes currently supported by the C00 and previous releases of Pathway for terminals in the IBM 3270 family. If you do not specify a SET MINIMUM-ATTR statement, the default value is IBM-BASE-ATTR.
- IBM-FIELD-REVERSE, IBM-FIELD-BLINK, and IBM-FIELD-UNDERLINE specify the set of highlight display attributes available on terminals in the IBM 3270 family.
- IBM-FIELD-OUTLINE specifies the set of outline display attributes for terminals in the IBM 3270 family, specifically TOPLINE, LEFTLINE, RIGHTLINE, BOTTOMLINE, and BOXFIELD.

A value of 1 is placed in the corresponding elementary item if that attribute is required. For example, if you want reverse video and outlining, you move the value 1 into the IBM-FIELD-REVERSE and IBM-FIELD-OUTLINE elementary fields. All other fields are set to 0.

The mimimum level of support for highlight or outline display attributes does not change until a subsequent SET MINIMUM-ATTR statement is executed, even if you use a CALL statement to move between program units.

SET MINIMUM-COLOR Statement

The SET MINIMUM-COLOR statement establishes the minimum level of support for color display attributes.

```
SET MINIMUM-COLOR USING data-name.
```

data-name

specifies the Working-Storage group item template for color display attributes, which must be defined as follows:

```
01 WS-MINIMUM-COLOR.
   02 FILLER                          PIC 9(04) COMP.
   02 IBM-PARTIAL-COLOR               PIC 9(04) COMP.
   02 IBM-FULL-COLOR                  PIC 9(04) COMP.
```

- IBM-PARTIAL-COLOR specifies terminals in the IBM 3270 family that can support three colors.
- IBM-FULL-COLOR specifies terminals in the IBM 3270 family that can support six colors.

A value of 1 is placed in the corresponding elementary item if that attribute is required. For example, if you want the program unit to work only on terminals with full color capability, you move the value 1 into IBM-FULL-COLOR. All other Working-Storage fields are set to 0.

The mimimum level of support for color display attributes does not change until a subsequent SET MINIMUM-COLOR statement is executed, even if you use a CALL statement to move between program units.

If you do not specify a SET MINIMUM-COLOR statement, the default setting for color display attributes is no color. All elementary fields in the Working-Storage group DEVICE-VIDEO-COLOR are set to 0.

STOP RUN Statement

The STOP RUN statement causes the executing program to stop immediately after this statement executes.

```
STOP RUN
```

If the executing program is a called program unit and a STOP RUN statement is executed, control does not return to the calling program.

SUBTRACT Statements

The SUBTRACT statements subtract elementary numeric data items and set the results in specific data items. The forms of the SUBTRACT statements are:

```
SUBTRACT
SUBTRACT GIVING
SUBTRACT CORRESPONDING
```

Each form is described in the following paragraphs.

SUBTRACT Statement

The SUBTRACT statement totals all data items before keyword FROM and then subtracts that sum from the current value of each data item after keyword FROM.

```
SUBTRACT { sub-1 } ,... FROM { sub-2 } ,...
```

sub-1

is either a numeric literal or the identifier of an elementary numeric data item.

sub-2

is the identifier of an elementary numeric data item.

SUBTRACT GIVING Statement

The SUBTRACT GIVING statement is the same as the SUBTRACT statement, except the result is stored in separate data items.

```
SUBTRACT { sub-1 } ,... FROM sub-2 GIVING { result } ,...
```

sub-1

is either a numeric literal or the identifier of an elementary numeric data item.

sub-2

is either a numeric literal or the identifier of an elementary numeric data item.

result

is the identifier of an elementary numeric data item.

SUBTRACT CORRESPONDING Statement

The SUBTRACT CORRESPONDING statement subtracts elementary items in one group from any corresponding items in another group. Items correspond when they have the same names and qualifiers up to but not including the group item name specified in the SUBTRACT CORRESPONDING statement.

```
SUBTRACT { CORR           } group-1 FROM group-2
         { CORRESPONDING }
```

group-1 and *group-2*

are the identifiers of group items in which some or all of the elementary items are numeric. The results are placed in the *group-2* items.

Refer to the ADD CORRESPONDING Statement for examples of corresponding items. The following conventions apply to data items used with the CORRESPONDING phrase:

- A REDEFINES or OCCURS clause can be specified in the data description entry of any data item.
- Data items can be subordinate to a data description entry with a REDEFINES or OCCURS clause.
- No data item can be defined with a level number 66, 77, or 88.

Subordinate data items in two different groups correspond to each other according to the following rules:

- Both data items must have the same data name.
- All possible qualifiers for the sending data item, up to but not including a group name, must be identical to all possible qualifiers for the receiving data item up to but not including the receiving group name.
- Only elementary numeric data items are considered.
- Any data item subordinate to a data item that is not eligible for correspondence is ignored.
- FILLER data items are ignored.

TERMINALINFO Statement

The TERMINALINFO statement returns information about terminals. From this information you can determine what action is to be taken if a terminal does not support the required extended field attributes.

```
TERMINALINFO USING data-name.
```

data-name

specifies the Working-Storage group item template for color, highlight, and outline display attributes supported by a terminal.

The information returned by the TERMINALINFO statement is placed in an elementary item defined in the Working-storage Section. The Working-Storage group item template for color, highlight, and outline display attributes must be defined in Working-Storage as follows:

```
01 WS-TERMINAL-INFO.
   02 DEVICE-VIDEO-STATUS-ATTR    PIC S9(04) COMP VALUE 0.
   02 DEVICE-VIDEO-STATUS-COLOR   PIC S9(04) COMP VALUE 0.
   02 DEVICE-VIDEO-COLOR.
      03 IBM-PARTIAL-COLOR        PIC 9(4) COMP.
      03 IBM-FULL-COLOR           PIC 9(4) COMP.
      03 FILLER                   PIC 9(4) COMP.
   02 DEVICE-VIDEO-ATTRIBUTES.
      03 IBM-BASE-ATTR            PIC 9(4) COMP.
      03 IBM-FIELD-REVERSE        PIC 9(4) COMP.
      03 IBM-FIELD-BLINK          PIC 9(4) COMP.
      03 IBM-FIELD-UNDERLINE      PIC 9(4) COMP.
      03 IBM-FIELD-OUTLINE        PIC 9(4) COMP.
```

- DEVICE-VIDEO-STATUS-ATTR indicates the level of support for the REVERSE, BLINK, and UNDERLINE highlight display attributes requested in a SET MINIMUM-ATTR statement.
  - 1 indicates the terminal has more capability than requested.
  - 0 indicates the terminal has the same capability as requested.
  - -1 indicates the terminal has less capability than requested.

  If no SET MINIMUM-ATTR statement was issued, the reply indicates the capability of the terminal to support the default (that is, no highlight, color, or outline display attributes).
- DEVICE-VIDEO-STATUS-COLOR indicates the level of support for color display attributes requested in a SET MINIMUM-COLOR statement.
  - 1 indicates the terminal has more capability than requested.
  - 0 indicates the terminal has the same capability as requested.
  - -1 indicates the terminal has less capability than requested.

- IBM-PARTIAL-COLOR indicates whether the terminal supports three colors.
  - 1 indicates that the terminal supports three colors.
  - 0 indicates that the terminal does not support three colors.
- IBM-FULL-COLOR indicates whether the terminal supports six colors.
  - 1 indicates that the terminal supports six colors.
  - 0 indicates that the terminal does not support six colors.
- For the display attributes IBM-BASE-ATTR, IBM-FIELD-REVERSE, IBM-FIELD-BLINK, IBM-FIELD-UNDERLINE, and IBM-FIELD-OUTLINE:
  - 1 indicates that the terminal supports the attribute(s).
  - 0 indicates that the terminal does not support the attribute(s).

  IBM-BASE-ATTR indicates whether the terminal supports attributes supported by the C00 and previous releases of Pathway for terminals in the IBM 3270 family.

  IBM-FIELD-REVERSE indicates whether the terminal supports reverse video.

  IBM-FIELD-BLINK indicates whether the terminal supports blinking video.

  IBM-FIELD-UNDERLINE indicates whether the terminal supports underlining.

  IBM-FIELD-OUTLINE indicates whether the terminal supports outlining.

If, for example, a SET MINIMUM-COLOR statement is executed, a value of 0 returned by the TERMINALINFO statement in DEVICE-VIDEO-STATUS-COLOR indicates that the terminal supports the required color display attributes. Similarly, if a SET MINIMUM-ATTR statement is executed, a value of 1 returned by the TERMINALINFO statement in DEVICE-VIDEO-STATUS-ATTR indicates that the terminal supports the required highlight display attributes and some that are not required.

Note the following considerations when you take action based on the TERMINALINFO statement:

- The default setting is IBM-BASE-ATTR with no color display attributes. This default is equivalent to executing a SET MINIMUM-ATTR statement with IBM-BASE-ATTR set to 1 and all other Working-Storage fields set to 0 and executing a SET MINIMUM-COLOR statement with all the Working-Storage fields set to 0.
- After issuing a SET MINIMUM-ATTR or SET MINIMUM-COLOR statement, you can issue a TERMINALINFO statement for a terminal to determine the level at which the terminal supports the required extended field attributes.
- You can change the required attributes between DISPLAY BASE operations; however, the TCP does not use attributes that the device does not support.

TRANSFORM Statement

The TRANSFORM statement replaces a set of Working-Storage Section data items with another set of Working-Storage Section data items.

```
TRANSFORM  trans-rec-out

[ SELECT CODE FIELD [ IS ] code-field ]

{YIELDS trans-rec-in                                          }
{                                                             }
{{CODE select-code [,select-code]...YIELDS trans-rec-in}...}

[ON ERROR imperative-statement ]

[USER [ CONVERSION ] numeric-literal ]
```

trans-rec-out

specifies the output record that TRANSFORM is to build and is made up of:

```
out-item-1 [, out-item-n ... ]
```

out-item-n

is a Working-Storage Section or Message Section 01 level item, group item, or elementary data item.

out-item-1 through *out-item-n* must all be of one type, either Working-Storage Section or Message Section items, not a combination.

YIELDS *trans-rec-in*

specifies the input record that is to be the result of the TRANSFORM operation and is made up of:

```
in-item-1 [, in-item-n ... ]
```

in-item-1

is a Working-Storage Section or MessageSection 01 level item, group item, or elementary data item. *in-item-1* through *in-item-n* must all be of one type, either Working-Storage Section or Message Section items, not a combination of types.

SELECT CODE FIELD [IS] *code-field*

defines the location, length, and data type of the *select-code* field in the input record.

The absence of this clause causes the default to be used:

- Offset—0 bytes offset from beginning of the record
- Length—2 bytes
- Data type—COMPUTATIONAL numeric data item

You need to specify this clause when the location, length, and data type of the *select-code* field is other than the default.

The *code-field* parameter in this clause specifies a field name in the Working-Storage Section.

The offset of the specified field from the beginning of its record dictates the offset of the *select-code* field from the beginning of the input record.

The length of the specified field dictates the length of the *select-code* field.

The data type of the specified field dictates the data type of the *select-code* field.

`CODE` *select-code*

identifies a literal or a data item that specifies which input record is expected.

The position of the *select-code* in the CODE *select-code* clause corresponds to a TERMINATION-STATUS value. One or more *select-code* values can be associated with each input record.

A nonnumeric literal must be enclosed within quotation marks.

`ON ERROR` *imperative-statement*

specifies action to be taken should an error occur on either the input or output phase of the TRANSFORM operation. If an error is detected, the *imperative-statement* is executed.

If ON ERROR is omitted and an error is detected, the TCP takes standard action for terminals.

`USER [ CONVERSION ]` *numeric-literal*

identifies a user conversion procedure associated with either the input record or the output record.

The operation of the TRANSFORM statement is similar to that of a SEND MESSAGE statement, but there is no physical I/O. With TRANSFORM, the output phase (SEND MESSAGE) is followed by an input phase (REPLY) without the data being output to or input from a device or process.

When a TRANSFORM statement is invoked, a reference is made to the Working-Storage Section either directly or through the Message Section. During this output phase, TRANSFORM prepares the data for output in an internal buffer.

On the input phase TRANSFORM processes the data from this internal buffer. The data is stored directly into the Working-Storage Section or is mapped through a Message Section template on its way to Working-Storage.

On either the output or input phase, if the data is mapped through the Message Section, the Message Section entry specifies which Working-Storage Section items to process. In addition, the Message Section entry defines the characteristics of the data, including order, format, and length.

If the data transformation capabilities of ordering, editing, and conversion are not used, TRANSFORM acts as a move statement. You could program a series of data moves from place to place by setting up a Message Section template and then invoking a block move using TRANSFORM.

The following rules apply:

- For either the output record or the input record, you cannot have different sections represented within one record. Each record must contain all Working-Storage Section items or all Message Section items. The compiler produces an error if either record is made up of items from more than one section.
- You can have different sections represented by the output record and the input record. For example, the output record can be made up of items from the Working-Storage Section and the input record made up of items from the Message Section.
- At compile time, only limited edit rule checking is imposed. Compatibility between output and input templates is not checked at compile time. For example, it is possible to code the output of a PIC A field to be input into a PIC 9 field. The compiler will not attempt to find this type of error.
- At run time, checks will be made on the input and output sides to ensure that data consistent with the specified data type is found. For example, a run-time error will occur if either too much or too little data is available to satisfy the input requirements. If an error is found, the ON ERROR clause will be executed and TERMINATION-STATUS, TERMINATION-SUBSTATUS will be set accordingly. Further, should a data editing error be detected during a TRANSFORM, the proper FIELD-STATUS items will be updated to show the fields involved.
- Refer to Appendix D, Errors for Message Section Statements, for an explanation of the error numbers that can be found in the TERMINATION-STATUS special register.

Consider the following SCREEN COBOL examples with multiple input records:

```
DATA DIVISION.

WORKING-STORAGE SECTION.

01  WS-SOURCE                  PIC X(102).

01  WS-CODE-MSG.
    05  WS-SEL-CODE-FIELD      PIC 9(4) COMP.

01  WS-DESTINATION-TEMP-1.
    05  WS-DEST-1-FLD1         PIC X(50).
    05  WS-DEST-1-FLD2         PIC X(50).

01  WS-DESTINATION-TEMP-2.
    05  WS-DEST-2-FLD1         PIC X(30).
```

```
    05  WS-DEST-2-FLD2        PIC X(30).
    05  WS-DEST-2-FLD3        PIC X(40).

MESSAGE SECTION.

01  MS-SOURCE                 PIC X(102) FROM WS-SOURCE.

01  DESTINATION-TEMP-1.
    05  FILLER                PIC 9(4) COMP.
    05  MS-DEST-1-FLD1        PIC X(50) TO WS-DEST-1-FLD1.
    05  MS-DEST-1-FLD2        PIC X(50) TO WS-DEST-1-FLD2.

01  DESTINATION-TEMP-2.
    05  FILLER                PIC 9(4) COMP.
    05  MS-DEST-2-FLD1        PIC X(30) TO WS-DEST-2-FLD1.
    05  MS-DEST-2-FLD2        PIC X(30) TO WS-DEST-2-FLD2.
    05  MS-DEST-2-FLD3        PIC X(40) TO WS-DEST-2-FLD3.

PROCEDURE DIVISON.
TRANSFORM  MS-SOURCE
    SELECT CODE FIELD IS WS-SEL-CODE-FIELD OF WS-CODE-MSG
       CODE 44, 54 YIELDS DESTINATION-TEMP-1
       CODE 64, 74 YIELDS DESTINATION-TEMP-2
    ON ERROR  PERFORM ERROR-HANDLER.

PERFORM ONE OF PROCESS-DEST-TEMP-1
               PROCESS-DEST-TEMP-2
DEPENDING ON TERMINATION-STATUS.
```

This TRANSFORM statement executes as follows:

1. TRANSFORM builds the output record from the data in WS-SOURCE. TRANSFORM sends this data through MS-SOURCE in the Message Section.

2. TRANSFORM determines the location, length, and type of the *code-field*, WS-SEL-CODE-FIELD OF WS-CODE-MSG. The location of this field relative to the beginning of the Working-Storage record defines the offset of the select code relative to the beginning of the input record. In this case, WS-SEL-CODE-FIELD is at the beginning of the Working-Storage record. Therefore, the select code is to be found at the beginning of the input record.

   The SELECT CODE FIELD IS clause could have been omitted in this example because it specifies the default.
   The SELECT CODE FIELD IS clause in the TRANSFORM statement must always have a Working-Storage Section data item specified as the *code-field*.

3. The FILLER fields in DESTINATION-TEMP-1 and DESTINATION-TEMP-2 are place holders for the fields in the input records that contain the select code. These fields are processed by the TCP and discarded so that they are not stored in Working-Storage.

4. Because there can be more than one input record, TRANSFORM compares the contents of the first two bytes of the input record to determine where to send the data. If the select code is 44 or 54, TRANSFORM sends the data through

DESTINATION-TEMP-1. If the select code is 64 or 74, TRANSFORM sends the data through DESTINATION-TEMP-2.

TRANSFORM moves a number from 1 to 4 into the TERMINATION-STATUS register depending on the position of the select code relative to the other possible select codes.

- If there are no errors, the TERMINATION-STATUS special register is set as follows:

Select Code	TERMINATION-STATUS
44	1
54	2
64	3
74	4

- If TRANSFORM detects an error, it sets TERMINATION-STATUS to a value that indicates the type of error and performs the procedure ERROR-HANDLER.

In the following example TRANSFORM looks for the select code 30 bytes from the beginning of the input record. The position of the *code-field*, WS-SEL-CODE-FIELD, in the Working-Storage record WS-CODE-MSG defines this offset. Also, notice that the select codes are specified in Working-Storage data items SC1 and SC2 rather than with numeric literals.

```
DATA DIVISION.

WORKING-STORAGE SECTION.

01  WS-SOURCE                  PIC X(102).
01  WS-CODE-MSG.
    05 PREFIX                  PIC X(30).
    05 WS-SEL-CODE-FIELD       PIC 9(4) COMP.
01  REPLY CODES.
    05  SC1                    PIC 9(4) COMP, VALUE IS 1.
    05  SC2                    PIC 9(4) COMP, VALUE IS 2.
01  WS-DESTINATION-TEMP-1.
    05  WS-DEST-1-FLD1         PIC X(30).
    05  WS-DEST-2-FLD2         PIC X(20).
    05  WS-DEST-1-FLD3         PIC X(50).
01  WS-DESTINATION-TEMP-2.
    05  WS-DEST-2-FLD1         PIC X(30).
    05  WS-DEST-2-FLD2         PIC X(30).
    05  WS-DEST-2-FLD3         PIC X(40).


MESSAGE SECTION.

01  MS-SOURCE                  PIC X(102) FROM WS-SOURCE.

01  DESTINATION-TEMP-1.
    05  MS-DEST-1-FLD1         PIC X(30) TO WS-DEST-1-FLD1.
    05  FILLER                 PIC 9(4) COMP.
```

```
    05  MS-DEST-1-FLD2        PIC X(20) TO WS-DEST-1-FLD2.
    05  MS-DEST-1-FLD3        PIC X(50) TO WS-DEST-1-FLD3.

01  DESTINATION-TEMP-2.
    05  MS-DEST-2-FLD1        PIC X(30) TO WS-DEST-2-FLD1.
    05  FILLER                PIC 9(4) COMP.
    05  MS-DEST-2-FLD2        PIC X(30) TO WS-DEST-2-FLD2.
    05  MS-DEST-2-FLD3        PIC X(40) TO WS-DEST-2-FLD3.


PROCEDURE DIVISON.
TRANSFORM  MS-SOURCE
    SELECT CODE FIELD IS WS-SEL-CODE-FIELD OF WS-CODE-MSG
       CODE SC1 YIELDS DESTINATION-TEMP-1
       CODE SC2 YIELDS DESTINATION-TEMP-2
    ON ERROR PERFORM ERROR-HANDLER.
```

TURN Statement

The TURN statement changes the display attributes of screen variable fields. The TURN statement cannot be used by programs communicating with intelligent devices.

```
TURN [ TEMP      ] { [ mnemonic-name  ]                      }
     [ TEMPORARY ] { [ DYNAMIC        ]                      }
                   { [ RECEIVE FROM { ALTERNAL }            }
                   {                {ALTERNATE OR TERMINAL} }
                   {                {TERMINAL              } }
                   {                {TERMINAL OR ALTERNATE} }

  IN

  { screen-identifier } [,]... [ DEPENDING [ON] identifier ]
                               [ SHADOWED                  ]
```

TEMP or TEMPORARY

indicates the fields are to be reset to their initial display attributes when the next RESET TEMP or ACCEPT statement is executed.

Note. During execution of a SCREEN COBOL program, the TCP controls the MDTs (modified data tag) in the same way it controls display attributes, with two important exceptions:

- When a TURN TEMP statement selects an input field for changing display attributes, the MDT bit is always set.
- When a RESET TEMP statement selects an input field for resetting of attributes, the MDT bit is set, regardless of the initial MDT attribute of the field.

These two exceptions apply only to the TURN and RESET statements that have the TEMP modifier. Note also that the field's MDT bit is not reset after the completion of the ACCEPT statement. Once the MDT bit is set, it stays set until the next DISPLAY BASE, TURN, RESET, or CLEAR INPUT operation.

---

mnemonic-name

specifies the display attributes to be used. The *mnemonic-name* must be associated with one or more display attributes through an IS phrase in the SPECIAL-NAMES paragraph in the Environment Division of the program.

DYNAMIC

indicates that the specified screen field can have its screen attributes constructed from a combination of the screen field's attributes as defined at compile time and the attribute settings as defined in the control structure.

RECEIVE FROM

specifies the type of device from which data can be accepted for a screen field. This clause is supported only for applications running on 6530 or 6540 terminals with 6AI (revision A00 or later) firmware.

screen-identifier

specifies the fields whose attributes can be changed. Each identifier can be the name of an entire screen, a screen group, or an elementary item of any base or overlay screen that is currently displayed. If *screen-identifier* is not an elementary item, *screen-identifier* refers to all subordinate elementary items that have a TO, FROM, or USING clause in their definitions. There may be at most 127 screen identifiers per TURN statement.

DEPENDING ON *identifier*

selects zero or one *screen-identifier* from the list. The statement whose position in the *screen-identifier* list is the same as the value in *identifier* is selected. If the value in *identifier* is less than 1 or greater than the number of screen identifiers, no *screen-identifier* is selected.

SHADOWED

selects from the *screen-identifier* list only those fields that have SHADOWED items with the SELECT bit set; fields that do not have SHADOWED items are not selected.

Note. If neither the DEPENDING ON modifier nor the SHADOWED modifier is specified, all fields in the *screen-identifier* list are selected.

The attributes of the selected fields are changed to those specified by *mnemonic-name*. The settings for attributes not specified by *mnemonic-name* are determined by the initial attributes of the field.

The DYNAMIC modifier provides the capability of changing the screen field attribute settings at run time using the contents of individual attribute elements in an associated control structure. It allows you to base the new display attributes specified by the TURN statement on the run-time attribute settings, not the compile-time settings. To return the attribute settings back to the compile-time values, use the RESET statement.

Table 6-7 shows the criteria for determining the selection of a screen field in a TURN operation.

Table 6-7. Screen Field Selection Criteria in TURN Operation

DYNAMIC SHADOWED	Selected	Not Selected	Selected
	Screen Fields With These Characteristics Are…		
TURN Operation Specifiers	**SHADOWED: YES, SELECT=1**	**SHADOWED: YES, SELECT=0**	**SHADOWED: NO**
mnemonic-name	Selected	Selected	Selected
mnemonic-name SHADOWED	Selected	Not selected	Selected
DYNAMIC	Selected	Selected	Selected

TURN does not cause a physical write to the screen but causes data to be written to the terminal buffer. A physical write to the screen occurs when one of the following events occurs:

- TERMBUF fills up
- One of the following statements is executed: ACCEPT; BEGIN-TRANSACTION, END-TRANSACTION, RESTART-TRANSACTION, or ABORT-TRANSACTION; CALL; CHECKPOINT; DELAY; EXIT PROGRAM; PRINT SCREEN; or SEND
- The DISPLAY statement is executed for a conversational terminal

The DEPENDING ON clause for the TURN statement is analogous to the DEPENDING ON clause for the PERFORM ONE statement. The following example illustrates this.

```
WORKING-STORAGE SECTION.
    :
77  ws-screen-status    PIC 9(4)    COMP VALUE 1.
01  ws-fld1             PIC x(10).
01  ws-fld2             PIC x(10).
01  ws-fld3             PIC x(10).

SCREEN SECTION.
    :
01  MENU1 SIZE 24, 80.
    05 screen-fld1    at 4, 20
                      PIC X(10)
                      from fld1.
    05 screen-fld2    at 5, 40
                      PIC X(10)
                      from fld2.
    05 screen-fld3    at 6, 60
                      PIC X(10)
                      from fld3.
```

```
PROCEDURE DIVISION.
    :
BODY-PARAGRAPH.
    :
  DISPLAY BASE MENU1.
  DISPLAY MENU1.
  TURN REVERSE IN SCREEN-FLD1,
                  SCREEN-FLD2,
                  SCREEN-FLD3,
  DEPENDING ON WS-SCREEN-STATUS.
```

If WS-SCREEN-STATUS equals 1, SCREEN-FLD1 is displayed in REVERSE. If WS-SCREEN-STATUS equals 2, SCREEN-FLD2 is displayed in REVERSE, and so on. It is not considered erroneous if WS-SCREEN-STATUS < 1 or WS-SCREEN-STATUS>3. Control just falls through (execution continues with the next statement) and no screen field is displayed in REVERSE.

Attribute Handling for IBM 3270 Terminals

Following are the conflicting scenarios you might encounter when you dynamically set the display attributes by using the TURN Statement.

- If you change a field's HIDDEN attribute to a BLINK, BRIGHT, REVERSE, or UNDERLINE attribute, there is no effect on the display.
- If you change a field with a BLINK attribute to HIDDEN, the field changes to HIDDEN.
- If you change a field with a BLINK attribute to REVERSE, the field goes reverse but it does not blink.
- If you change a field with a BLINK attribute to UNDERLINE, the field is underlined but it does not blink.
- If a field is originally REVERSE and you change it to BLINK, the field only blinks and is not reversed.
- If you change a REVERSE field to UNDERLINE, then the field is underlined but not reversed.
- If you change an underlined field to HIDDEN, neither the text nor the underline is displayed; they are hidden.
- If you change an underlined field to BLINK, the underline does not appear but the text blinks.
- If you cnage a field with an UNDERLINE attribute to REVERSE, the field is reversed but the underline does not appear
- If the field initially set to top, bottom, left, right, or in the boxfield was changed to BLINK, then only the text will blink. The lines will not blink.

Attribute Handling for 6500-Series Terminals

The following table shows how SCREEN COBOL handles multiple attributes for 6500-series terminals.

	HIDDEN	PROTECTED	BLINK	MDTON	DIM	REVERSE	UNDERLINE
HIDDEN	Hidden	Hidden & Protected	Hidden & Blink	Hidden and Mdton	Area is Dim. Text is Hidden.	Area is Reverse. Text is Hidden.	Underline and Hidden
PROTECTED	Protected & Hidden	Protected	Protected & Blink	Protected & Mdton	Protected & Dim	Protected & Reverse	Protected & Underline
BLINK	Blink & Hidden	Blink & Protected	Blink	Blink & Mdton	Blink & Dim	Blink & Reverse	Blink & Underline
MDTON	Mdton & Hidden	Mdton & Protected	Mdton and Blink	Mdton	Mdton & Dim	Mdton & Reverse	Mdton & Underline
DIM	Text is Hidden & area is Dim	Dim & Protected	Dim & Blink	Dim & Mdton	Dim	Dim & Reverse	Dim & Underline
REVERSE	Reverse & Hidden	Reverse & Protected	Reverse & Blink	Reverse & Mdton	Reverse & Dim	Reverse	Reverse & Underline
UNDERLINE	Underline & Hidden	Underline & Protected	Underline & Blink	Underline & Mdton	Underline & Dim	Underline & Reverse	Underline

USE FOR SCREEN RECOVERY Statement

The USE FOR SCREEN RECOVERY statement is entered in the Declaratives portion of the Procedure Division. It identifies a procedure to restore the terminal display should an error occur while the specified base screens are active. USE FOR SCREEN RECOVERY clause is invoked when there are terminal or communication errors, processor failures, or terminal suspensions. See the *Compaq NonStop™ Pathway/iTS TCP and Terminal Programming Guide* for a list of errors that invoke the USE FOR SCREEN RECOVERY clause. The Declaratives procedure is called automatically only after the terminal is suspended and restarted with the PATHCOM command RESUME TERM. Typically, the SCREEN COBOL program displays advisory text on the screen, and the user must then take corrective action. If no user error recovery is provided in the SCREEN COBOL program, then the TCP takes its own action on terminal errors.

The USE FOR TERMINAL-ERRORS clause is invoked when there is an irrecoverable error due to a terminal error or communications device error. The Declaratives procedure does not apply to errors detected by logic in the SCREEN COBOL program. Note that the DISPLAY RECOVERY statement cannot be used in a USE FOR SCREEN RECOVERY statement. The USE FOR SCREEN RECOVERY statement cannot be used by programs that communicate with intelligent devices.

```
USE [ FOR SCREEN ] RECOVERY

  [ ON { base-screen-name-n } ,... ] .
```

ON *base-screen-name-n*

specifies the base screens for which the declarative procedures are to be used. If this phrase is omitted, the declarative procedures are used for all screens not mentioned in another USE statement.

The recovery process performs the equivalent of a DISPLAY BASE for the current base screen followed by a DISPLAY OVERLAY for all currently active overlay screens. The applicable declarative procedure statements are then executed. When the declarative procedures complete execution, control returns to the statement that was being executed when the problem was detected; the statement is executed again. USE FOR SCREEN RECOVERY must immediately follow a section header in the Declaratives portion of the Procedure Division. The following example illustrates this USE statement:

```
PROCEDURE DIVISION.
DECLARATIVES.
S-R SECTION.
    USE FOR SCREEN RECOVERY.
RECOV-1.
    MOVE "SCREEN RECOVERY" TO error-msg,
    DISPLAY msg-1.
END DECLARATIVES.
MAIN SECTION.
```

USE FOR TERMINAL-ERRORS Statement

The USE FOR TERMINAL-ERRORS statement is entered in the Declaratives portion of the Procedure Division. It identifies a procedure to be executed only when an irrecoverable error occurs during terminal I/O. The error could occur because of terminal errors or communication line errors. See the *Compaq NonStop™ Pathway/iTS TCP and Terminal Programming Guide* for a list of errors that invoke the USE FOR TERMINAL -ERRORS clause.

The USE FOR TERMINAL-ERRORS statement cannot be used by programs that communicate with intelligent devices.

```
USE [ FOR ] TERMINAL-ERRORS.
```

If an irrecoverable terminal I/O error occurs, the TCP executes the declaratives procedure immediately following any USE FOR TERMINAL-ERRORS statement. The TCP executes the procedure as soon as it determines that a terminal error is

irrecoverable. It differs in this respect from the USE FOR SCREEN RECOVERY statement that executes its declaratives procedure only after the terminal is suspended and resumed.

If an irrecoverable terminal I/O error occurs and the program does not contain a USE FOR TERMINAL-ERRORS statement, the program suspends.

If a program contains a USE FOR TERMINAL-ERRORS statement and an irrecoverable terminal I/O error occurs, the TCP attempts screen recovery automatically at the next screen operation, unless a DISPLAY RECOVERY statement is executed successfully before the next screen operation. The next screen operation need not be in the Declaratives section for the TCP to perform the automatic screen recovery.

The following SCREEN COBOL statements add data to the terminal buffers and thus could cause an irrecoverable terminal I/O error:

```
ACCEPT                  DISPLAY
CLEAR INPUT             RECONNECT MODEM
DISPLAY BASE            RESET
DISPLAY OVERLAY         SCROLL
DISPLAY RECOVERY        TURN
```

For programs operating in conversational mode, an irrecoverable terminal I/O error can be associated directly with one of the statements listed. For programs operating in block mode, the error might not result directly from one of these statements.

In block mode, output data is buffered and an irrecoverable terminal I/O error occurs only when data is actually written to or read from the terminal. A number of input or output statements could be executed before the buffer fills or some other action forces an actual write to the terminal. In block mode, actual terminal writes occur in the following situations:

- When the terminal buffer is full
- Before executing an ACCEPT statement
- Before executing statements such as DELAY, SEND, or CALL

If an irrecoverable terminal I/O error occurs and you have included a USE FOR TERMINAL-ERRORS procedure in your program, the TCP performs the following actions:

1. Sets the TERMINATION-SUBSTATUS special register to the appropriate file-system error code
2. Sets the PW-TERMINAL-ERROR-OCCURRED special register to 1 (otherwise, it is set to zero)
3. Executes the declaratives procedure immediately following the USE FOR TERMINAL-ERRORS statement
4. Resumes execution at the statement immediately following the statement during which the I/O failure occurred

If an error occurs in the declaratives procedure itself, the TCP sets the special registers PW-TERMINAL-ERROR-OCCURRED to 1 and TERMINATION-SUBSTATUS to the

file-system error code just as in steps 1 and 2, but resumes execution at the next statement in the procedure itself.

The USE FOR TERMINAL-ERRORS declarative is both simple and powerful. It allows you maximum control when a terminal I/O error occurs, but it can lead to unexpected results. You should be careful both in how you use this procedure and for what purpose.

Consider the following:

- In block mode, you have no way of knowing which statement caused the error.
- Pathway logging is not performed on errors handled by the USE FOR TERMINAL-ERRORS procedure.
- If an irrecoverable terminal I/O error occurs during an ACCEPT statement, the TCP resets TERMINATION-STATUS to zero before returning to the statement following the ACCEPT.
- If you call another SCREEN COBOL program unit from within a USE FOR TERMINAL-ERRORS declaratives procedure:
  - The called program (and any program it calls) must not contain a USE FOR TERMINAL-ERRORS statement; if it does, a run-time error occurs and the program suspends.
  - An irrecoverable terminal I/O error in the called program will cause the calling program to suspend.

You should consider the nature of the error before attempting a retry within the Declaratives section. Some errors can be retried, others cannot. To determine the nature of the error, you can check the TERMINATION-SUBSTATUS special register for the file-system error code.

If the declaratives procedure does not contain any executable statement, execution resumes with the next statement following the statement that caused the error. Thus, irrecoverable terminal I/O errors can occur with no indication that they have occurred. One way to avoid this situation is to include at least one statement in the declaratives procedure, possibly an EXIT PROGRAM statement.

The Declaratives procedures can be used to save current data (context). Saving current data by using the declaratives procedures lets you resume the application with a more completely recovered screen than is possible with, for instance, EXIT PROGRAM WITH ERROR or CALL...ON ERROR when used outside the declaratives procedure. The following example saves the current and last record when a terminal I/O error occurs.

```
  . . .
DATA DIVISION.
WORKING-STORAGE SECTION.
01  CONTEXT-INFO.
    02 CURRENT-REC.
       04 NAME          PIC A(33).
       04 ACCT-NUM.
           06 ACCT-PRE  PIC AA.
```

```
            06 ACCT-SN   PIC 99999.
            06 ACCT-SUFF PIC X(4).
         04 ACT-BAL      PIC 999999V99.
     02 LAST-REC.
         04 NAME         PIC A(33).
         04 ACCT-NUM.
            06 ACCT-PRE  PIC AA.
            06 ACCT-SN   PIC 99999.
            06 ACCT-SUFF PIC X(4).
         04 ACT-BAL      PIC 999999V99.
01   REPLY-STRUCT.
     02 OUTCOME          PIC 999.
     02 EXTENSION        PIC AAA.
     . . .

PROCEDURE DIVISION.
DECLARATIVES.
TERM-ERROR SECTION.
   USE FOR TERMINAL-ERRORS.
   TERM-ERROR-PROC.
      SEND CONTEXT-INFO
           TO TROUBLE-SERVER
           REPLY OUTCOME OF REPLY-STRUCT.
   TERM-ERROR-END.
      EXIT PROGRAM.
 END DECLARATIVES.
 MAIN SECTION.
    . . .
```

The following example contains only an EXIT PROGRAM statement.

```
PROCEDURE DIVISION.
DECLARATIVES.
U-T-E SECTION.
    USE FOR TERMINAL-ERRORS.
    TERM-ERRORS.
        EXIT PROGRAM.
    END DECLARATIVES.
MAIN SECTION.
 . . .
DISPLAY BASE
 . . .
DISPLAY
 . . .
DISPLAY
```

If the declarative did not contain the EXIT PROGRAM statement, any of the DISPLAY statements could fail due to an irrecoverable terminal I/O error and not be detected.

7 Compilation

The SCOBOLX run command invokes the SCREEN COBOL compiler. If you choose to use all default parameters, you would enter the command as follows:

```
SCOBOLX
```

Running the SCREEN COBOL Compiler

The SCREEN COBOL compiler is usually run from the command interpreter. The syntax of the command is:

```
SCOBOLX [ / [ IN source-file ]
            [ , OUT [ list-file ] ]
            [ , run-option ] ... / ]
  [ tclprog-file ]
  [ , copy-library ]
  [ ; compiler-command ] ...
```

Note. You must compile a source program unit using a SCOBOLX run command on the same node where the source program unit resides. You cannot compile on one node a program unit residing on another node.

`IN` *`source-file`*

is a file containing SCREEN COBOL statements and compiler commands. The SCREEN COBOL compiler reads *`source-file`* as 132-byte records. If this parameter is omitted, input is taken from the current input file of the command interpreter; this is typically the home terminal.

`OUT` *`list-file`*

directs listing output to a named file. The file has the same form as *`source-file`*. If *`list-file`* is an unstructured disk file, each *`list-file`* record is 132 characters (partial lines are blank filled through column 132).

If *`list-file`* is omitted and OUT is present, no listing is produced. If the entire option is omitted, the listing output is directed to the command interpreter OUT file; this is typically the home terminal.

`run-option`

is one of the following operating system command options:

`NAME $`*`process-name`*

specifies the symbolic process name.

`CPU` *`cpu-number`*

specifies the processor module.

PRI *priority*

specifies the execution priority.

MEM *num-pages*

specifies the maximum number of data pages.

NOWAIT

specifies the command interpreter does not suspend while the program runs.

SWAP $*disk-volume*

specifies the disk volume where the SCOBOLX and SCOBOLX2 process swap files are created.

If you do not specify this option and you do not specify a default disk volume for process swap files (using the Compaq Tandem Advanced Command Language (TACL) SET SWAP command), the process swap files are created in the same subvolume where the SCOBOLX object program file resides. System performance might be adversely affected if the SCOBOLX object program file resides on $SYSTEM and the process swap files are created in the same subvolume as that of the SCOBOLX object program file.

For information about swap-file management, refer to the *Kernel-Managed Swap Facility (KMSF) Manual*.

tclprog-file

is the file specification from which the compiler generates names of disk files that contain the object program and other information. The *tclprog-file* has a disk file name of the following form:

```
[ \system . ] [ $volume . ] [ subvolume . ] file
```

file must not exceed 5 characters. If this parameter is omitted, POBJ is used. *file* specifies the root of the actual file names that the compiler generates by appending characters as follows:

COD

specifies the code file (contains the object program).

DIR

specifies the directory to the code file.

SYM

specifies the symbols table (generated if you specify the compiler option SYMBOLS).

The default file names are: POBJCOD, POBJDIR, and POBJSYM.

copy-library

is the name of an EDIT disk file. This file is used as the default library for source code when expanding a COPY statement that does not include a specific library name. *copy-library* has the form of a disk file name.

If this parameter is omitted, COPYLIB is used.

compiler-command

is any of the following compiler commands:

ANSI	OPTION
COMPILE	SHOWCOPY/NOSHOWCOPY
CROSSREF/NOCROSSREF	SMAP/NOSMAP
ERRORS	SYMBOLS/NOSYMBOLS
HEADING	SETTOG
LINES	SYNTAX
LIST/NOLIST	TANDEM
MAP/NOMAP	WARN/NOWARN

The following examples illustrate the command to run the SCREEN COBOL compiler:

```
SCOBOLX / IN MYSOURCE, OUT $S, NOWAIT / MPROG; MAP; &
CROSSREF ONLY LABELS; LIST

SCOBOLX / IN APROG /
```

Using Compiler-Generated Files

If the disk files do not exist, SCREEN COBOL creates them and stores object program information in them. If the COD, DIR, and SYM files exist, SCREEN COBOL adds the object program to those programs already present in the files. The addition is done in a way that does not disrupt concurrent use of the file, even if the program ID of the object program being added is the same as one already present. Thus, additions can be made to object files while they are in use by a TCP.

When you refer to the object files in PATHCOM commands and SCUP, use the file name root (without the COD, DIR, or SYM). When referring to the object files in commands, use the actual names (including the COD, DIR, or SYM). The actual files can be renamed or copied to another volume as long as the new file names are related in the same way; that is, they must have the same root and be the same except that one ends with COD, one ends with DIR, and one ends with SYM.

Using PARAM SAMECPU

The command interpreter SAMECPU parameter specifies running process SYMSERV in the same CPU in which SCOBOLX and SCOBOLX2 run. Usually SYMSERV runs in the CPU with the next higher number than that of the CPU in which SCOBOLX and SCOBOLX2 are running. If there is no processor with a higher number, SYMSERV runs in the processor with the lowest number.

The SCREEN COBOL compiler recognizes the SAMECPU parameter passed in the command interpreter PARAM command. The format of the command is:

```
PARAM SAMECPU number
```

number

is any nonzero number. A 0 disables SAMECPU.

number does not designate which CPU the processes run in; that is controlled by the CPU run option.

The command sequence to run processes in the same CPU is:

```
TACL 1> PARAM SAMECPU 1
TACL 2> SCOBOLX/IN ... , OUT ...
```

Using PARAM SWAPVOL

The command interpreter SWAPVOL parameter specifies a swap volume on which:

- SCOBOLX and SCOBOLX2 create their temporary work files
- SCOBOLX2 creates its process swap disk volume, which contains the run-time data stack swap file for the SCOBOLX2 process

The SCREEN COBOL compiler recognizes the SWAPVOL parameter passed in the command interpreter PARAM command. The format of the command is:

```
PARAM SWAPVOL $volume
```

$*volume*

specifies a disk volume other than the current one.

If you do not specify PARAM SWAPVOL, all temporary work files for both SCOBOLX and SCOBOLX2 are created in the same subvolume where the SCOBOLX object program file resides, unless the current swap volume has been specified in a TACL SET SWAP command. System performance might be adversely affected if the SCOBOLX object program file resides on $SYSTEM and the process swap files are created in the same subvolume as that of the SCOBOLX object program file.

Note. The SWAP option of the SCOBOLX run command does not affect where the temporary work files for SCOBOLX and SCOBOLX2 are created, but it does affect where the Kernel Managed Swap Facility (KMSF) creates the run-time data stack swap file to be used by the SCOBOLX process.

For example, the command sequence to specify $MINT as the disk volume on which temporary work files reside is:

```
TACL 3> PARAM SWAPVOL $MINT
```

Using Compiler Commands

Compiler commands provide information to the compiler and select compilation features. For example, compiler commands select the source format, the file to which the compiler writes object code, and listing options. Compiler commands also control selective compilation of portions of the source file.

Specifying Compiler Commands

Compiler commands, with one exception, can be specified in two places: in the compiler command field in the SCOBOLX run command and in the SCREEN COBOL source file. (See the SECTION Command on page 7-14 for the exception.) Specify the commands as follows:

- In the compiler-command field in the SCOBOLX run command—Precede each compiler command with a semicolon (;). Some compiler commands are also option commands. See the OPTION Command on page 7-13 for an alternate way to specify these commands.
- In the SCREEN COBOL source file—Specify each command alone in a source text line. A command can appear at any point in the source text, including those portions retrieved from a source library file with the COPY statement. Compiler command lines in the source text cannot be interspersed with multiline COPY statements.

The format of a compiler command in the source text is:

```
?compiler-command
```

`?`

appears in the indicator field (column 1 for Tandem standard reference format and either column 1 or 7 for ANSI standard reference format).

`compiler-command`

is any of the compiler commands described in this section.

The question mark is a source text format indicator and not part of the compiler command. The compiler command entered as part of the SCREEN COBOL run command is not preceded by a question mark.

When Compiler Commands Take Effect

The SCREEN COBOL compiler treats the commands specified in the SCOBOLX run command as if they were specified at the beginning of the source file. The compiler lists them at the beginning of the list file. The commands are in effect at the beginning of the compilation in the order in which they appear in the command. If conflicting commands are specified, the last command overrides the others.

Compiler commands specified in the source file take effect at the beginning of the next source text line. When conflicting commands are specified, the last command specified

overrides all others. Commands specified in the source text override commands specified in the SCOBOLX run command.

Compiler Command Summary

Compiler commands are divided into categories by function. The categories and descriptions are:

- Option commands—Specify the source text input format, the source listing options, the title field of the page header, and compilation options. For more information about option commands see the OPTION Command on page 7-13.
- Cross-reference commands—Control the generation of cross-reference information on program identifiers.
- Toggle commands—Selectively compile portions of the source text. Up to 15 flags, called toggles, can be turned on (set), turned off (reset), or tested.
- Section command—Identifies individual texts in a SCREEN COBOL source library accessed by a COPY statement.

Table 7-1 lists compiler option commands and their defaults. Table 7-2 lists compiler cross-reference commands and their defaults. Table 7-3 lists compiler toggle commands.

Table 7-1. Compiler Option Commands (page 1 of 2)

Command	Default
ANSI	TANDEM
COMPILE	
ERRORS	ERRORS 100
HEADING	
LINES	LINES 60
LIST	
MAP	NOMAP
NOLIST	LIST
NOMAP	
NOSHOWCOPY	SHOWCOPY
NOSMAP	
NOSYMBOLS	
NOWARN	WARN
OPTION	
SHOWCOPY	
SMAP	NOSMAP

Table 7-1. Compiler Option Commands (page 2 of 2)

Command	Default
SYMBOLS	NOSYMBOLS
SYNTAX	COMPILE
TANDEM	
WARN	

Table 7-2. Compiler Cross-Reference Commands

Command	Default
CROSSREF	NOCROSSREF
NOCROSSREF	

Table 7-3. Compiler Toggle Commands

Command	Default
ENDIF	
IF	
IFNOT	
RESETTOG	
SETTOG	

Compiler Command Descriptions

The commands are described in alphabetic order in the following paragraphs.

ANSI Command

The ANSI command specifies that the following source text is in ANSI standard reference format.

```
ANSI
```

Lines longer than 80 characters are truncated; shorter lines are padded with trailing blanks. The positions following Margin R (columns 73 through 80) form the identification field. This field, which can contain any ASCII characters, is treated as a comment and has no effect on the meaning of a program.

If this command is omitted, TANDEM is the source text format.

Note. For programs that need to be executed using the Compaq *Inspect* debugging tool, the SYMBOLS compiler command is required so that the compiler will pass the necessary information to SYMSERV which stores the data in a symbol table file. If the ANSI compiler command is also used, then the information for each line is identified by a set of line numbers generated by the compiler. This set of numbers starts at one and is incremented by one for each succeeding line. During an Inspect session, the line numbers from the source file are used to access lines of source for the program. An incompatibility arises if these line numbers are not the same as the line numbers used to identify the data via SYMSERV; wrong source lines will be displayed for the SOURCE command in Inspect. Recompilation of the program will avoid any problems in this area after the source file is resequenced by one.

COMPILE Command

The COMPILE command requests full compilation and production of an object file.

```
COMPILE
```

If this command and the SYNTAX command are omitted, COMPILE is assumed.

CROSSREF Command

The CROSSREF command causes a list of the SCREEN COBOL program identifiers to be added to the compiled program output. The list is a cross-reference showing where the identifiers are described, read, or written throughout the program. This command contains a list of classes into which program identifiers are classified. Selections from the class list determine the identifiers to be included in the cross-reference listing.

The NOCROSSREF command is the default and disables the CROSSREF command.

```
{ CROSSREF [ ONLY    ] [ class ] ... }
{          [ INCLUDE ]               }
{          [ EXCLUDE ]               }
{                                    }
{ NOCROSSREF                         }
```

`ONLY`

requests information for just the classes specified.

`INCLUDE`

adds a class of identifiers to an existing class list.

`EXCLUDE`

deletes a class of identifiers from an existing class list.

`class`

is one of the following SCREEN COBOL identifiers:

`CONDITIONS`

Items tested in the program that have condition names

`DATANAMES`
`VARIABLES`

Data items defined in the Working-Storage Section

`LABELS`
`PROCNAMES`

Paragraph names and section names

`LITERALS`

Numeric and nonnumeric literals

`MNEMONICS`

Mnemonic names associated with display attributes

`PROGRAMS`

Program unit names for called programs

`SCREEN`

Screen groups or fields described in the Screen Section

`UNREFS`

Items defined in the program, but never referred to

Specifying CROSSREF with no options or classes produces a list of the following program identifiers:

```
CONDITIONS
DATANAMES or VARIABLES
LABELS or PROCNAMES
MNEMONICS
PROGRAMS
SCREENS
```

If the NOLIST or SYNTAX commands are specified in the SCREEN COBOL source program or at compile time, no cross-reference listing is produced. For a complete description of CROSSREF, refer to the *CROSSREF Manual*.

ENDIF Command

The ENDIF command terminates the effect of a preceding IF or IFNOT command.

```
ENDIF toggle-number
```

toggle-number

is the *toggle-number* specified in the IF or IFNOT command.

ERRORS Command

The ERRORS command sets the maximum number of errors allowed during compilation.

```
ERRORS nnnnn
```

nnnnn

is an integer from 0 through 32,767.

If this command is omitted, the default limit is 100 errors.

If the limit set by the ERROR command is exceeded, the compilation terminates.

HEADING Command

The HEADING command replaces or sets to blanks the heading portion of the standard top-of-page line that appears on each page of the compilation listing.

```
HEADING  [ "character-string" ]
```

"*character-string*"

is a string of any ASCII characters enclosed in quotation marks. At least one character must appear. If a quotation mark is part of the string, it must be represented as two contiguous quotation marks. The character string is used in all subsequent top-of-page lines.

If the *character-string* option is omitted, the heading portion of these lines is set to all blanks.

IF Command

The IF command causes the compiler to ignore subsequent source text unless the specified toggle is turned on with a SETTOG command.

```
IF toggle-number
```

toggle-number

is an integer from 1 through 15.

COPY statements are not affected by this command; these statements are still processed and expanded.

The following example illustrates the IF command:

```
?RESETTOG 1, 2
     .
     .
     .
?IF 2
     .
     .
     .
   text
     .
     .
     .
?ENDIF 2
```

The source text bounded by the IF 2 and ENDIF 2 commands is ignored.

IFNOT Command

The IFNOT command causes the compiler to ignore subsequent source text unless the specified toggle is turned off either with the RESETTOG command or by default (never set).

```
IFNOT toggle-number
```

toggle-number

is an integer from 1 through 15.

COPY statements are not affected by this command; these statements are still processed and expanded.

The following example illustrates the IFNOT command:

```
?RESETTOG 1, 2
     .
     .
     .
?IFNOT 1
```

```
      .
      .
      .
    text
      .
      .
      .
?ENDIF 1
```

The source text bounded by the IFNOT 1 and ENDIF 1 commands will be compiled.

LINES Command

The LINES command sets the number of lines listed on each page. Whenever the next line to be listed would overflow the line count, a page is ejected and the standard page heading and two blank lines are listed at the top of the next page, followed by the pending line.

```
LINES nnnnn
```

nnnnn

is an integer from 10 through 32,767.

The line limit is ignored if paging does not apply to the compilation list device.

If this command is omitted, LINES 60 is assumed.

LIST Command

The LIST command transmits each source image to the compilation list device. The NOLIST command disables the LIST option.

```
{ LIST   }
{ NOLIST }
```

A MAP command is not effective unless LIST is enabled.

If LIST and NOLIST are omitted, LIST is assumed.

MAP Command

The MAP command lists a table of user-defined symbols following the listing of the program or subprogram source text. The NOMAP command disables the MAP option.

```
{ MAP   }
{ NOMAP }
```

The MAP command is not effective unless LIST is enabled. The MAP command does not produce a statement offset list if the SYNTAX option is specified.

OPTION Command

The OPTION command controls the source text input format, the source listing options, the title field of the page header, and compilation options.

```
[ OPTION ] command-option [ , command-option ] ...
```

command-option

is any of the following commands:

```
ANSI                NOSMAP
COMPILE             NOSYMBOLS
ERRORS              NOWARN
HEADING             OPTION
LINES               SHOWCOPY
LIST                SMAP
MAP                 SYMBOLS
NOLIST              SYNTAX
NOMAP               TANDEM
NOSHOWCOPY          WARN
```

A single OPTION command can contain any combination of the available options, in any order. An option takes effect at the beginning of the next source text line. If a command contains two or more conflicting options, the last option specified overrides all the others. For example, the following commands are equivalent:

```
OPTION LIST, ERRORS 20, LIST, NOLIST
OPTION ERRORS 20, NOLIST
```

The OPTION command can cause confusion over the uses of commas and semicolons as separators in the SCOBOLX run command. The same compiler commands can appear as follows:

- Listed as a command option in the OPTION command with the comma as the separator
- Specified as a compiler command with the semicolon as the separator

For example, the following commands are equivalent:

```
SCOBOLX / in infile / mprog; OPTION ERROR 20, NOLIST;
SCOBOLX / IN INFILE / mprog; ERROR 20; NOLIST
```

The keyword OPTION is not required. The following examples show the items ERROR 20 and NOLIST listed in an OPTION command where OPTION is assumed. The examples show a run command and a source text entry.

```
SCOBOLX / IN INFILE / ; ERROR 20, NOLIST; CROSSREF

?ERROR 20, NOLIST
?CROSSREF
```

RESETTOG Command

The RESETTOG command turns off all specified toggles.

```
RESETTOG [ toggle-number [ , toggle-number ] ... ]
```

toggle-number

is an integer from 1 through 15.

If the toggle-number option is omitted, all toggles are turned off.

SECTION Command

The SECTION command is used to identify individual texts in a SCREEN COBOL source library accessed by a COPY statement. The command is ignored if it appears in the text of the compilation source file.

```
SECTION text-name [ , library-text-format ]
```

text-name

is a SCREEN COBOL word (1 to 30 letters, digits, and hyphens, but not all digits).

The compiler assumes that the format of the library text is the same as the current source text format. Although this default format can be overridden by entering a compiler command as the first line following the SECTION command, the ANSI or TANDEM command is usually more convenient for this purpose.

The following reserved words cannot be used in a SECTION command:

```
ANSI                LABELS
CHECK               LITERALS
COMPILE             MESSAGES
CONDITIONS          ONLY
DEBUG1              OPTION
DEBUG2              PROGRAMS
ENDIF               SCREENS
ERRORS              SYNTAX
EXCLUDE             TRACE
INCLUDE             VARIABLES
```

SETTOG Command

The SETTOG command turns on all specified toggles.

```
SETTOG [ toggle-number [ , toggle-number ] ... ]
```

toggle-number

is an integer from 1 through 15.

If the *toggle-number* option is omitted, all toggles are turned on.

SHOWCOPY Command

The SHOWCOPY command specifies listing the COPY statement as a comment line in the list file before the copied statements. The NOSHOWCOPY disables SHOWCOPY.

```
{ SHOWCOPY   }
{ NOSHOWCOPY }
```

If this command is omitted, SHOWCOPY is assumed.

SMAP Command

The SMAP command creates a descriptor map following the listing of the program or subprogram source text. The map produced by the SMAP command is more compact and contains more information than that produced by the MAP command. If both MAP and SMAP are specified, only SMAP is effective.

```
{ SMAP   }
{ NOSMAP }
```

If this command is omitted, NOSMAP is assumed.

The SMAP command is not effective unless LIST is enabled. The SMAP command does not produce a statement offset list if the SYNTAX option is specified.

SYMBOLS Command

The SYMBOLS command causes a symbol table file to be built for the SCREEN COBOL program. This file is used by INSPECT to examine and debug programs. The NOSYMBOLS command disables the SYMBOLS command so that the compiler does not build a symbol table file.

```
{ SYMBOLS   }
{ NOSYMBOLS }
```

If this command is omitted, NOSYMBOLS is assumed.

If SYMBOLS is specified, data items (either elementary or group items) cannot exceed 12,288 characters.

The file for the symbol table is given the name used in the SCOBOLX run command with SYM appended. In the following example, the SCOBOLX compiler compiles the program in TESTFILE and adds the symbol table to a file named MANUFSYM.

```
SCOBOLX / IN TESTFILE / MANUF; MAP; SYMBOLS
```

A program compiled with NOSYMBOLS produces, in most cases, a significantly smaller run unit than a program compiled with SYMBOLS. For NOSYMBOLS, the compiler discards all data descriptors for nonreferenced data declarations. For SYMBOLS, all data descriptors are retained in the run unit.

SYNTAX Command

The SYNTAX command requests a syntax check of the source text only. No object file is produced.

```
SYNTAX
```

If this command and the COMPILE command are omitted, COMPILE is assumed.

If this command is specified with the CROSSREF command, no cross-reference listing is produced.

If the SYNTAX option is specified, the MAP and SMAP commands do not produce statement offset lists.

TANDEM Command

The TANDEM command specifies that the following source text is in Tandem standard reference format.

```
TANDEM
```

If this command and the ANSI command is omitted, TANDEM is assumed.

Lines in Tandem standard reference format can have up to 132 characters (longer lines are truncated). The source text does not include either the initial six-character sequence number area or the final six-character identification field of the ANSI standard reference format.

WARN Command

The WARN command allows minor error conditions to be reported in the source text. The NOWARN command disables the WARN option.

```
{ WARN   }
{ NOWARN }
```

If this command is omitted, WARN is assumed.

If LIST is not enabled, the last line of source text scanned by the compiler is also listed to provide a point of reference.

Compilation Statistics

Statistics are printed at the end of every compilation. For example:

```
OBJECT FILE NAME IS $MART.BEN.POBJ
PROGRAM NAME IS EXAMPLE
PROGRAM VERSION IS 1
NO. ERRORS=0;     NO. WARNINGS=0
CODE SIZE=245
RUN UNIT SIZE=748
DATA SIZE=328
NUMBER OF SOURCE LINES READ=147
MAXIMUM SYMBOL TABLE SIZE=4920  WORDS
ELAPSED TIME -   0:00:32
```

`OBJECT FILE NAME IS`

the short form of the object file name. In the example, the object code for the program is placed in the files POBJCOD, POBJDIR, and POBJSYM on the $MART.BEN subvolume.

`PROGRAM NAME IS`

the name of the program (this line is printed only if no errors occurred).

`PROGRAM VERSION IS`

the version number of the program (this line is printed only if no errors occurred).

`NO. ERRORS =`

the total number of error messages issued.

`NO. WARNING =`

the total number of warning messages issued.

`CODE SIZE =`

the total number of bytes used for all Procedure Division code in the object file.

`RUN UNIT SIZE =`

the total number of bytes in the POBJCOD file taken up by the program unit.

`DATA SIZE =`

the total number of bytes of user-allocated working storage, plus compiler-allocated working storage.

`NUMBER OF SOURCE LINES READ`

the total number of source lines read by the SCREEN COBOL compiler, including any COPY lines.

`MAXIMUM SYMBOL TABLE SIZE =`

the number of words that the compiler needed for its symbol table (this is a snapshot view and should be considered only a rough estimate).

`ELAPSED TIME -`

wall-clock elapsed time for the compilation.

Stopping the Compiler

A compilation is performed by three compiler processes. To stop a compilation before normal completion, do the following:

1. Press the BREAK key.
2. Type STOP to terminate SCOBOLX, the first compiler process.
3. Type either STOP and the SCOBOLX2 PID number or PAUSE to terminate the second and third compiler processes, SCOBOLX2 and SYMSERV respectively. If you type PAUSE, SCOBOLX2 issues an error message (** FAILURE 10 ** COMPILER COMMUNICATION LOST : 00) and terminates.
4. Press the BREAK key to return to the command interpreter.
5. Type a STATUS command to check that the unwanted processes have terminated.

Conserving Disk Space

You can prevent unnecessary use of disk space by monitoring the number of object files allowed to accumulate during program development. Each time a SCREEN COBOL source program is compiled, a new version of the object program is added to the code file and an entry is made in the directory file.

For information on how to manipulate and maintain multiple versions of SCREEN COBOL object files, refer to the *Compaq NonStop™ Pathway/iTS SCUP Reference Manual*.

SCREEN COBOL Limits

The Compaq *NonStop™ Himalaya* system architecture and the method of implementing the SCREEN COBOL compiler impose certain limits on programs, such as:

- A nonnumeric literal cannot exceed 120 characters.
- A numeric literal cannot exceed 18 digits.
- The value of numeric data cannot exceed 18 digits.
- The maximum total size of a SCREEN COBOL program unit is 65,408 bytes. Of that, the pseudocode is restricted to a maximum of 32,767 bytes. Data size, however, is not restricted. It is recommended that an elementary data item or referenced group item not exceed 32,000 bytes in length. (These items are defined in the Working-Storage Section or Linkage Section.) However, it is possible for data to exceed 32,000 bytes by using space not needed for pseudocode.
- The maximum message or reply length in a SEND statement cannot exceed the data space allocated to the TCP by any of the following three parameters of the PATHCOM command SET TCP:
  - SERVERPOOL (limit for server I/O messages)
  - MAXTERMDATA (limit for terminal context data)
  - MAXREPLY (limit for a reply)

See the Pathway system administrator at your site for values for these limits. In addition, a message and reply are included in Working Storage and are affected by the limit of 32,000 bytes for Working Storage.

- The number of paragraphs in a PERFORM ONE statement cannot exceed 255.
- A screen item (defined in the Screen Section) cannot exceed 255 characters in length.
- Length specified in the LENGTH MUST BE clause (screen item) cannot exceed 255 characters.
- A screen item can have only one subscript. A data item can have a maximum of three subscripts.
- Representation of a PICTURE character string cannot exceed 30 characters. Data items with more character positions must be described with parenthesized repetition counts.
- An overlay screen cannot exceed the size of the overlay area.
- The maximum number of diagnostic messages that the compiler writes during compilation is 32,767.
- The maximum number of screen field identifiers in a DEPENDING clause is 127 per RESET, SET, DISPLAY, or TURN statement.

8 Pathway Application Example

This section provides a sample Pathway application, including the commands that create the PATHMON and PATHCOM processes, sample commands that configure Pathway and define the components to be used, two SCREEN COBOL programs to illustrate general programming concepts for terminals operating in block mode and for terminals operating in conversational mode, and an associated server program written in COBOL.

The example can be duplicated exactly as shown by taking the following steps in the order indicated:

1. Code the sample COBOL server program.
2. Compile and run the sample COBOL server program. Rename the default object file RUNUNIT to EXSERV to correspond to the Pathway configuration.
3. Code the sample SCREEN COBOL application program.
4. Compile the sample SCREEN COBOL application program for terminals operating in block mode. The object files default to POBJCOD and POBJDIR.

   Compile and run the sample SCREEN COBOL application program for terminals operating in conversational mode. (This program is just for illustration and does not communicate with the server program included in this section.)
5. Code the Pathway configuration file, named PWCONFIG. Change the process names $term01 and $term02 in the two SET TERM commands to legal terminal names in your installation. For convenience, the second set of SET TERM commands specifying $term02 can be eliminated without interfering with the operation.
6. Set up an obey file that contains the PATHMON and PATHCOM run commands. The PATHMON name $PM can be changed to any appropriate five-letter name.
7. Issue an OBEY command.
8. Issue the PATHCOM RUN command for the SCREEN COBOL application program.

The following rules should be noted before attempting to establish this application:

- The Pathway configuration is established through a command terminal. The command terminal cannot be included in the configuration.
- A Pathway system should allow more than one PATHCOM to communicate with PATHMON at a time; therefore, the SET PATHWAY MAXPATHCOMS command should never be set to 1. (The default is 5.)
- The individual at the command terminal can issue appropriate commands and alter the Pathway configuration online. This is accomplished by typing PATHCOM and responding to the PATHCOM = prompt. If the PATHMON name is not $PM, this PATHCOM command must include the $process-name parameter.

- If a configured terminal has a command interpreter, the terminal operator must type PAUSE to activate the SCREEN COBOL application on the terminal.

The sample SCREEN COBOL requester program for terminals operating in block mode describes a base screen only that appears as shown in the following example. The program accepts operator input that consists of a name and address until an appropriate function key is pressed.

```
DEPARTMENT : MKT          PASSWORD :

NAME : _____________________________
ADDR : ____________________

MONTH : FEBRUARY      DAY : 15   YEAR : 92

REPLY -

F1 - ENTER PASSWORD         F5  - BLINK REPLY
F2 - ENTER DATA             F6  - RESET ATTR REPLY
F3 - CLEAR INPUT            F7  - RESET DATA REPLY
F4 - RESET DATA SCREEN      F16 - EXIT PROGRAM
```

- If the operator types the name SMITH and presses the F2 key to enter data, the server returns a reply code of 999 and an error code of 1; this causes the message SMITH IS ALREADY ON FILE to be displayed in the advisory field of the terminal.
- If the operator types the name JONES and presses the F2 key to enter data, the server returns a reply code of 999 and an error code of 2; this causes the message JONES IS ALREADY ON FILE to be displayed in the advisory field of the terminal.
- If the operator types any other name and presses the F2 key to enter data, the server returns a reply code of 0 and writes the record; this causes the entered record to be displayed on the terminal screen.

The sample SCREEN COBOL requester program for terminals operating in conversational mode describes a base screen only and illustrates the SCREEN COBOL characteristics for conversational mode programs. The program displays a screen header, prompts the operator, accepts operator input that consists of a name and an address. The program also contains a function selection and responds to the input control characters named in the program, but no server response is provided.

PATHMON and PATHCOM Process Creation

To execute the program use the PATHCOM RUN command.

```
PATHMON / NAME $PM, CPU 0, NOWAIT /
PATHCOM / IN PWCONFIG / $PM
```

PWCONFIG contains the following commands:

```
SET PATHMON   BACKUPCPU
SET PATHWAY   MAXTCPS 1
SET PATHWAY   MAXTERMS 5
SET PATHWAY   MAXSERVERCLASSES 5
```

```
SET PATHWAY   MAXSERVERPROCESSES 5
SET PATHWAY   MAXSTARTUPS 5
SET PATHWAY   MAXASSIGNS 5
SET PATHWAY   MAXPARAMS 5
SET PATHWAY   MAXPATHCOMS 3
SET PATHWAY   MAXPROGRAMS 1
START PATHWAY COLD !

RESET TCP
SET TCP       CPUS 1:2
SET TCP       PROGRAM $SYSTEM.SYSTEM.PATHTCP2
SET TCP       PRI 141
SET TCP       PROCESS $XTCP
SET TCP       TCLPROG pobj
SET TCP       MAXTERMS 5
ADD TCP       ex-tcp

RESET TERM
SET TERM      FILE $term01
SET TERM      TCP ex-tcp
SET TERM      INITIAL example
SET TERM      TMF OFF
ADD TERM      t1
RESET TERM
SET TERM      LIKE t1
SET TERM      FILE $term02
ADD TERM      t2

RESET PROGRAM
SET PROGRAM   TCP ex-tcp
SET PROGRAM   TYPE T16-6520 INITIAL example
SET PROGRAM   ERROR-ABORT ON
SET PROGRAM   TMF OFF
ADD PROGRAM   exprog

RESET SERVER
SET SERVER    PROGRAM exserv
SET SERVER    CPUS 0:1
SET SERVER    NUMSTATIC 1
SET SERVER    MAXSERVERS 3
ADD SERVER    example-server

START TCP     ex-tcp
START TERM *
```

SCREEN COBOL Program for Block Mode

The SCREEN COBOL programs and an associated COBOL server appear on the following pages:

```
IDENTIFICATION DIVISION.                    (1)
PROGRAM-ID. EXAMPLE.                        (1)

ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  SOURCE-COMPUTER. T16.
  OBJECT-COMPUTER. T16,
    TERMINAL IS T16-6520.
  SPECIAL-NAMES.
    F1-KEY IS F1, F2-KEY IS F2, F3-KEY IS F3,
    F4-KEY IS F4, F5-KEY IS F5, F6-KEY IS F6,
```

```
    F7-KEY IS F7,  F16-KEY IS F16
    ATTENTION IS BLINK, HIDDEN IS HIDDEN.

DATA DIVISION.
  WORKING-STORAGE SECTION.

  01 WS.
      02 ERROR-MSG          PIC X(77).
      02 PASSWORD           PIC X(3).
      02 DEPT-HEADER        PIC X(3).

  01 EXIT-FLAG              PIC S9        VALUE 0.
      88 EXIT-PROGRAM       VALUE 1.

  01 ENTRY-MSG.
      02 PW-HEADER.                                  (2)
        04 REPLY-CODE       PIC S9(4) COMP.   (2)   (4)
        04 APPLICATION-CODE PIC XX.           (2)   (3)
        04 FUNCTION-CODE    PIC XX.           (2)   (3)
        04 TRANS-CODE       PIC 99.           (2)   (3)
        04 TERM-ID          PIC X(15).        (2)   (3)
        04 LOG-REQUEST      PIC X.            (2)   (3)
      02 ENTRY-GROUP.
        04 NAME-IN          PIC A(30).
        04 ADDR-IN          PIC X(20).
        04 DATE-GRP.
          06 MONTH-IN       PIC A(10).
          06 DAY-IN         PIC 99.
          06 YEAR-IN        PIC 99.

  01 ENTRY-REPLY.                                   (5)
      02 PW-HEADER.                                 (5)
        04 REPLY-CODE       PIC S9(4) COMP.   (4)   (5)
        04 FILLER           PIC X(22).        (5)
      02 SERVER-RECORD      PIC X(64).        (5)

  01 ERROR-REPLY.
      02 REPLY-CODE         PIC S9(4) COMP.   (5)
      02 FILLER             PIC X(22).        (4)   (5)
      02 ERROR-CODE         PIC S999  COMP.   (5)
```

(1) These lines give the program name that is specified in the SET TERM INITIAL command. This program is used when a terminal is first started.

(2) These lines illustrate a sample header for the transaction messages.

(3) These lines are not required.

(4) These lines show the reply code that is required by Pathway. The item must be defined as COMPUTATIONAL.

(5) These lines show that two reply messages are used to limit the amount of data sent between the server and the SCREEN COBOL program. When only an error code is returned from the server, *ERROR-REPLY* is used. When data is returned, *ENTRY-REPLY* is used.

```
SCREEN SECTION.
01 EXAMPLE-SCREEN BASE SIZE 24, 80.
  03 FILLER        AT 1, 20  VALUE "EXAMPLE SCREEN COBOL PROGRAM".(1)
  03 FILLER        AT 3, 1   VALUE "DEPARTMENT :".                (2)
  03 DEPT-HEADER   AT 3, 14  PIC X(3) FROM DEPT-HEADER OF WS.     (3)
  03 FILLER        AT 3, * + 10 VALUE "PASSWORD :".               (4)
  03 PASSWORD      AT 3, * + 2  PIC X(3) LENGTH 1 THRU 3, HIDDEN, (5)
         UPSHIFT INPUT, MUST BE "AAA", "X", TO PASSWORD OF WS.   (5)
```

```
03 DATA-IN.
   05 FILLER      AT 5, 1  VALUE "NAME :".
   05 NAME-IN     AT 5, 8  PIC A(30) LENGTH 1 THRU 30          (6)
                           TO NAME-IN OF ENTRY-MSG, FILL "_".  (6)
   05 FILLER      AT 6, 1  VALUE "ADDR :".
   05 ADDR-IN     AT 6, 8  PIC X(20) LENGTH 1 THRU 20
                           TO ADDR-IN OF ENTRY-MSG, FILL "_".
   05 DATE-GRP    AT 8, 1.
      07 FILLER   AT @, 1  VALUE "MONTH :".                   (7)
      07 MONTH-IN AT @, * + 2  PIC A(10) LENGTH 1 THRU 10
            MUST BE "JANUARY", "FEBRUARY" USING MONTH-IN OF
            ENTRY-MSG, UPSHIFT INPUT, VALUE "FEBRUARY".
      07 FILLER   AT @, * + 4  VALUE "DAY :".
      07 DAY-IN   AT @, * + 2  PIC Z9 LENGTH 1 THRU 2, VALUE "15"
            MUST BE 1 THRU 31, USING DAY-IN OF ENTRY-MSG.
      07 FILLER   AT @, * + 4  VALUE "YEAR :".
      07 YEAR-IN  AT @, * + 2  PIC Z9 MUST BE 79, 82, 85 THRU 88
            USING YEAR-IN OF ENTRY-MSG, VALUE "85".

03 FILLER         AT 10, 1 VALUE "REPLY -".
03 SERVER-RECORD  AT 10, * + 2  PIC X(64)
                           FROM SERVER-RECORD OF ENTRY-REPLY.
03 FILLER         AT 18, 1 VALUE
                  "F1 - ENTER PASSWORD    F5 - BLINK REPLY".
03 FILLER         AT 19, 1 VALUE
                  "F2 - ENTER DATA        F6 - RESET ATTR REPLY".
03 FILLER         AT 20, 1 VALUE
                  "F3 - CLEAR INPUT       F7 - RESET DATA REPLY".
03 FILLER         AT 21, 1 VALUE
                  "F4 - RESET DATA SCREEN F16 - EXIT PROGRAM".
03 ERROR-MSG      AT 24, 2 PIC X(76)  ADVISORY                 (8)
                           FROM ERROR-MSG OF WS.               (8)
```

(1) The literal is displayed on the screen starting at line 1, column 20.

(2) The literal is displayed on the screen starting at line 3, column 1.

(3) If a data name is used in a screen section, a PIC clause must be associated with that data name. The FROM (data association clause) specifies an output association. Data is moved from DEPT-HEADER OF WS to this position on the screen.

(4) The asterisk means relative to the current position; therefore, the literal PASSWORD is displayed on the screen at line 3, column 26 (16 + 10).

(5) The data entered for PASSWORD is hidden from the operator as it is entered. The password is upshifted and tested for the correct value. If the password is correct, the password is moved to the data name PASSWORD of working storage.

(6) The operator must type in from 1 to 30 alphabetic characters that are moved to ENTRY-MSG. A fill character of underscore is present on the screen in these 30 positions.

(7) The at sign (@) indicates the position is relative to the home position of the group. This literal is displayed on line 8, column 1.

(8) These lines identify the field to be used for information and error messages generated by the TCP. The programmer also can use this field.

```
  PROCEDURE DIVISION.
A-MAIN.
    DISPLAY BASE EXAMPLE-SCREEN                                    (1)
    MOVE "MKT" TO DEPT-HEADER OF WS.
    DISPLAY DEPT-HEADER OF EXAMPLE-SCREEN                          (2)
    ACCEPT PASSWORD OF EXAMPLE-SCREEN UNTIL F1-KEY                 (3)
    PERFORM CASE-MANAGER UNTIL EXIT-PROGRAM.
A-EXIT.
    EXIT PROGRAM.

CASE-MANAGER.
    ACCEPT DATA-IN OF EXAMPLE-SCREEN UNTIL F2-KEY                  (4)
      ESCAPE ON F3-KEY F4-KEY F5-KEY F6-KEY F7-KEY F16-KEY         (4)
    PERFORM ONE OF
      DATA-ENTERED, CLEAR-INPUT, RESET-DATA, BLINK-REPLY
      RESET-ATTR-REPLY, RESET-DATA-REPLY, SET-EXIT
    DEPENDING ON TERMINATION-STATUS                                (5)

DATA-ENTERED.
    MOVE SPACES TO PW-HEADER OF ENTRY-MSG.
    PERFORM SEND-DATA.
CLEAR-INPUT.
    CLEAR INPUT                                                    (6)
RESET-DATA.
    RESET DATA EXAMPLE-SCREEN                                      (7)
BLINK-REPLY.
    TURN ATTENTION IN SERVER-RECORD OF EXAMPLE-SCREEN              (8)
RESET-ATTR-REPLY.
    RESET ATTR SERVER-RECORD OF EXAMPLE-SCREEN                     (9)
RESET-DATA-REPLY.
    RESET DATA SERVER-RECORD OF EXAMPLE-SCREEN                     (10)
SET-EXIT.
    MOVE 1 TO EXIT-FLAG.

SEND-DATA.
    SEND ENTRY-MSG TO "EXAMPLE-SERVER"                             (11)
      REPLY CODE 0    YIELDS ENTRY-REPLY
            CODE 999  YIELDS ERROR-REPLY.
    IF TERMINATION-STATUS = 2 AND ERROR-CODE = 1
       MOVE "SMITH IS ALREADY ON FILE" TO ERROR-MSG OF WS
       PERFORM 901-DISPLAY-ADVISORY
    ELSE IF TERMINATION-STATUS = 2 AND ERROR-CODE = 2
       MOVE "JONES IS ALREADY ON FILE" TO ERROR-MSG OF WS
       PERFORM 901-DISPLAY-ADVISORY
    ELSE
       DISPLAY SERVER-RECORD OF EXAMPLE-SCREEN                     (12)

901-DISPLAY-ADVISORY.
    DISPLAY TEMP ERROR-MSG OF EXAMPLE-SCREEN                       (13)
    TURN TEMP ATTENTION IN ERROR-MSG OF EXAMPLE-SCREEN.            (14)
```

(1) This line displays the screen and the initial values, FILL characters, and default values.

(2) The value of DEPT-HEADER is moved to the screen at line 3, column 14.

(3) When the F1 key is pressed, the field is tested for validity. Data can be typed into any other field on the screen, but only the PASSWORD field is used.

(4) The UNTIL F2-KEY expects data to be entered before the F2 key is pressed and validity checks are performed. The ESCAPE series of function keys causes the statement to terminate without data being entered.

(5) The key that was pressed to terminate the ACCEPT statement has a positional value associated with it from the ACCEPT statement; the key is put into TERMINATION-STATUS.

(6) All unprotected fields are cleared.

(7) This line resets the fields to the initial values and FILL characters declared.

(8) This line causes SERVER-RECORD to blink by setting the BLINK attribute.

(9) This line stops the blinking of SERVER-RECORD by resetting the attribute to normal.

(10) This line resets the portion of SERVER-RECORD to its original value (blank line).

(11) This line specifies the server class to be used. This can be a data name in working storage.

(12) The fields that comprise SERVER-RECORD are displayed.

(13) This line displays ERROR-MSG on the screen as temporary data.

(14) This line sets the BLINK attribute as a temporary attribute and makes the value of ERROR-MSG blink.

SCREEN COBOL Program for Conversational Mode

The following is a SCREEN COBOL program for conversational terminals:

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. dconv-exp                                                    (1)

ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER.  T16.
 OBJECT-COMPUTER.  T16, TERMINAL IS CONVERSATIONAL                        (2)
 SPECIAL-NAMES.
              BELL IS BELL,
              NOBELL IS NOBELL.

DATA DIVISION.
WORKING-STORAGE SECTION.

01 EMPLOYEE-REC.
  05  EMP-LAST-NAME                 PIC X(10) VALUE SPACES.
  05  EMP-FIRST-NAME                PIC X(10) VALUE SPACES.
  05  EMP-MIDDLE-INIT               PIC X(02) VALUE SPACES.
  05  EMP-ADDR                      PIC X(30) VALUE SPACES.
  05  EMP-CITY                      PIC X(10) VALUE SPACES.
  05  EMP-STATE                     PIC X(02) VALUE SPACES.
  05  EMP-ZIP                       PIC 9(05) VALUE ZEROS.

01 WS-ADVISORY                      PIC X(70) VALUE SPACES.

01 WS-FUNC                          PIC X(06) VALUE SPACES.
  88  WS-SEARCH-REQUEST                       VALUE "SEARCH".
  88  WS-ADD-REQUEST                          VALUE "ADD".
  88  WS-DELETE-REQUEST                       VALUE "DELETE".
```

```
  88  WS-SHOW-REQUEST                         VALUE "SHOW".
  88  WS-EXIT-REQUEST                         VALUE "EXIT".

01 EXIT-FLAG                          PIC 9(01) COMP VALUE ZERO.
  88 EXIT-PROGRAM                              VALUE 1.
  88 INVALID-RESPONSE                          VALUE 2.

01 MESSAGE-ID                         PIC 9(04) COMP VALUE ZERO.

01 R-CODE                             PIC 9(04) COMP VALUE ZERO.
  88 SEND-ERROR                               VALUE 999.
```

(1) These lines give the program name that you use in the SET TERM INITIAL command.

(2) This line specifies a conversational mode terminal type and identifies the terminal type that you use in the SET PROGRAM TYPE and SET TERM TYPE commands.

```
SCREEN SECTION.
01 EMPLOYEE-REC-SCREEN    BASE  SIZE 24, 80
*                                  FIELD-SEPARATOR ","              (1)
*                                  GROUP-SEPARATOR ";"              (1)
                                   ABORT-INPUT     "AI"             (2)
                                   END-OF-INPUT    47               (2)
*   The keyboard character for END-OF-INPUT is "/"                  (1)
                                   RESTART-INPUT   58, 58.
*   The keyboard characters for RESTART-INPUT is "::"               (1)

   05 TITLE              AT 1, 3   VALUE "PERSONNEL SYSTEM EXAMPLE".
   05 NAME-PROMPT        AT 2, 1   VALUE "LAST NAME: ".
   05 LAST-NAME-FLD      AT 3, 1   PIC X(10)
                                   USING EMP-LAST-NAME
                                   LENGTH 1 THRU 10
                                   PROMPT NAME-PROMPT.              (3)
   05 FIRST-NAME-PROMPT  AT 2, 12  VALUE "FIRST NAME: ".
   05 FIRST-NAME-FLD     AT 3, 12  PIC X(10)
                                   USING EMP-FIRST-NAME
                                   LENGTH 1 THRU 10
                                   PROMPT FIRST-NAME-PROMPT.
   05 MI-PROMPT          AT 2, 24  VALUE "MI: ".
   05 MIDDLE-INIT-FLD    AT 3, 24  PIC X(2)
                                   USING EMP-MIDDLE-INIT
                                   PROMPT MI-PROMPT.
   05 ADDR-PROMPT        AT 4, 1   VALUE "ADDRESS: ".
   05 ADDR-FLD           AT 4, 11  PIC X(30)
                                   USING EMP-ADDR
                                   PROMPT ADDR-PROMPT.
   05 CITY-PROMPT        AT 5, 1   VALUE "CITY: ".
   05 CITY-FLD           AT 5, 11  PIC X(10)
                                   USING EMP-CITY
                                   PROMPT CITY-PROMPT.
   05 STATE-PROMPT       AT 5, 22  VALUE "STATE: ".
   05 STATE-FLD          AT 5, 30  PIC X(10)
                                   USING EMP-STATE
                                   PROMPT STATE-PROMPT.
   05 ZIP-PROMPT         AT 5, 45  VALUE "ZIP: ".
   05 ZIP-FLD            AT 5, 51  PIC Z(5)
                                   USING EMP-ZIP
                                   PROMPT ZIP-PROMPT.
   05 TYPEAHEAD-MSG      AT 10, 1  VALUE "TO GET TYPEAHEAD, ENTER    (4)
-                                    " LAST NAME, FIRST NAME, MIDDLE INITIAL." (4)
   05 PROMPT-AREA  AREA  AT 23, 1  SIZE 1, 80.
   05 ADVISORY-FLD     AT 24, 1  PIC X(70)
                                   ADVISORY FROM WS-ADVISORY.
```

```
01 EMPLOYEE-REC-PROMPT  OVERLAY   SIZE 1, 80.
   05 FUNC-PROMPT      AT  1, 1 VALUE "(FUNCTION) SEARCH, ADD,
-                        "DELETE, SHOW, EXIT: ".
   05 FUNC-INPUT       AT  1, 45 PIC X(06)
                                 TO WS-FUNC
                                 UPSHIFT INPUT
                                 LENGTH MUST BE 3 THRU 6
                                 PROMPT FUNC-PROMPT.
```

(1) These lines are instructive comments about the input control characters. They are not required by SCREEN COBOL.

(2) These lines redefine the terminal input characters you use for control during an ACCEPT statement.

(3) This is the first PROMPT clause for the screen. The value of this clause will be displayed indicating the terminal is ready to accept data for this field.

(4) These lines identify the typeahead message that is included in the heading displayed at the beginning of the screen.

```
PROCEDURE DIVISION.
BEGIN-PROGRAM.
    DISPLAY BASE EMPLOYEE-REC-SCREEN.
    DISPLAY TITLE, TYPEAHEAD-MSG.
    PERFORM LOOP UNTIL EXIT-PROGRAM.

EXIT-PROG.
    EXIT PROGRAM.

LOOP.
   ACCEPT EMPLOYEE-REC-SCREEN
      UNTIL INPUT
      ESCAPE ON ABORT.
   IF TERMINATION-STATUS = 1
      PERFORM FUNCTION-DISPLAY
      PERFORM INIT-EMPLOYEE-REC
   ELSE
      PERFORM EXIT-IT.

FUNCTION-DISPLAY.
   DISPLAY OVERLAY EMPLOYEE-REC-PROMPT AT PROMPT-AREA.
   MOVE 2 TO EXIT-FLAG.
   PERFORM OPERATION UNTIL NOT INVALID-RESPONSE.
OPERATION.
   ACCEPT EMPLOYEE-REC-PROMPT
      UNTIL INPUT
      ESCAPE ON ABORT.
   IF TERMINATION-STATUS = 1
      PERFORM FUNCTION-SELECTION
   ELSE
      PERFORM EXIT-IT.

FUNCTION-SELECTION.
   MOVE ZERO TO EXIT-FLAG.
   IF WS-SEARCH-REQUEST
      PERFORM SEARCH-IT
   ELSE
   IF WS-ADD-REQUEST
      PERFORM ADD-IT
   ELSE
   IF WS-DELETE-REQUEST
      PERFORM DELETE-IT
```

```
      ELSE
      IF WS-SHOW-REQUEST
         PERFORM SHOW-IT
      ELSE
      IF WS-EXIT-REQUEST
         PERFORM EXIT-IT
      ELSE
         PERFORM INVALID-FUNCTION.

   SEARCH-IT.
      MOVE 1 TO MESSAGE-ID.
      SEND MESSAGE-ID, EMPLOYEE-REC TO "USER-SERVER"
        REPLY CODE 1 YIELDS R-CODE, EMPLOYEE-REC
              CODE 2 YIELDS R-CODE
        ON ERROR MOVE 999 TO R-CODE.
      IF NOT SEND-ERROR
       PERFORM ONE OF DISPLAY-EMPLOYEE-REC, EMPLOYEE-NOT-FOUND
       DEPENDING ON R-CODE
      ELSE
       PERFORM SEND-ERROR-NOTICE.
   ADD-IT.
      MOVE 2 TO MESSAGE-ID.
      SEND MESSAGE-ID, EMPLOYEE-REC TO "USER-SERVER"
        REPLY CODE 1, 3 YIELDS R-CODE
        ON ERROR MOVE 999 TO R-CODE.
      IF NOT SEND-ERROR
        PERFORM ONE OF EMPLOYEE-ADDED, EMPLOYEE-ALREADY-EXISTS
        DEPENDING ON R-CODE
      ELSE
         PERFORM SEND-ERROR-NOTICE.

   DELETE-IT.
      MOVE 3 TO MESSAGE-ID.
      SEND MESSAGE-ID, EMPLOYEE-REC TO "USER-SERVER"
        REPLY CODE 1, 2 YIELDS R-CODE
        ON ERROR MOVE 999 TO R-CODE.
      IF NOT SEND-ERROR
       PERFORM ONE OF EMPLOYEE-DELETED, EMPLOYEE-NOT-FOUND
         DEPENDING ON R-CODE
      ELSE
         PERFORM SEND-ERROR-NOTICE.

   SHOW-IT.
      MOVE 4 TO MESSAGE-ID.
      SEND MESSAGE-ID, EMPLOYEE-REC TO "USER-SERVER"
        REPLY CODE 1, 2 YIELDS R-CODE
        ON ERROR MOVE 999 TO R-CODE.
      IF NOT SEND-ERROR
       PERFORM ONE OF DISPLAY-EMPLOYEE-REC, EMPLOYEE-NOT-FOUND
       DEPENDING ON R-CODE
      ELSE
         PERFORM SEND-ERROR-NOTICE.

   EXIT-IT.
     MOVE 1 TO EXIT-FLAG.

   DISPLAY-EMPLOYEE-REC.
     DISPLAY EMPLOYEE-REC-SCREEN.

   EMPLOYEE-NOT-FOUND.
     MOVE "EMPLOYEE DOES NOT EXIST" TO WS-ADVISORY.
     DISPLAY ADVISORY-FLD.

   EMPLOYEE-ADDED.
     MOVE "EMPLOYEE ADDED" TO WS-ADVISORY.
     DISPLAY ADVISORY-FLD.
```

```
EMPLOYEE-ALREADY-EXISTS.
  MOVE "EMPLOYEE ALREADY EXISTS" TO WS-ADVISORY.
  DISPLAY ADVISORY-FLD.

EMPLOYEE-DELETED.
  MOVE "EMPLOYEE DELETED" TO WS-ADVISORY.
  DISPLAY ADVISORY-FLD.

INIT-EMPLOYEE-REC.
  MOVE SPACES TO EMPLOYEE-REC.
  MOVE ZEROES TO EMP-ZIP.

INVALID-FUNCTION.
  MOVE 2 TO EXIT-FLAG.
  MOVE "INVALID FUNCTION REQUESTED" TO WS-ADVISORY.
  DISPLAY ADVISORY-FLD.

SEND-ERROR-NOTICE.
  MOVE "ERROR ACCESSING PERSONNEL SYSTEM" TO WS-ADVISORY.
  DISPLAY ADVISORY-FLD.
```

Server Program in COBOL

The following is a sample server program in COBOL:

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. EXAMPLE-SERVER.

ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  SOURCE-COMPUTER. TANDEM/16.
  OBJECT-COMPUTER. TANDEM/16.
  INPUT-OUTPUT SECTION.
    FILE-CONTROL.
     SELECT MESSAGE-IN, ASSIGN TO $RECEIVE
       FILE STATUS IS RECEIVE-FILE-STATUS.
     SELECT MESSAGE-OUT, ASSIGN TO $RECEIVE
       FILE STATUS IS RECEIVE-FILE-STATUS.

DATA DIVISION.
FILE SECTION.
FD  MESSAGE-IN
    LABEL RECORDS ARE OMITTED.
  01 ENTRY-MSG.
     02 PW-HEADER.
       04 REPLY-CODE       PIC S9(4) COMP.
       04 APPLICATION-CODE PIC XX.
       04 FUNCTION-CODE    PIC XX.
       04 TRANS-CODE       PIC 99.
       04 TERM-ID          PIC X(15).
       04 LOG-REQUEST      PIC X.
     02 ENTRY-GROUP.
       04 NAME-IN          PIC A(30).
       04 ADDR-IN          PIC X(20).
       04 DATE-GRP.
         06 MONTH-IN       PIC A(10).
         06 DAY-IN         PIC 99.
         06 YEAR-IN        PIC 99.

FD  MESSAGE-OUT
    LABEL RECORDS ARE OMITTED
    RECORD CONTAINS 1 TO 88 CHARACTERS.
  01 ENTRY-REPLY.
     02 PW-HEADER.
       04 REPLY-CODE       PIC S9(4) COMP.
       04 FILLER           PIC X(22).
     02 SERVER-RECORD      PIC X(64).
```

```
  01 ERROR-REPLY.
     02 REPLY-CODE         PIC S9(4) COMP.
     02 FILLER             PIC X(22).
     02 ERROR-CODE         PIC S999  COMP.
WORKING-STORAGE SECTION.
  01 RECEIVE-FILE-STATUS.
     02 STAT-1             PIC 9.
        88 CLOSE-FROM-REQUESTOR VALUE 1.
     02 STAT-2             PIC 9.

PROCEDURE DIVISION.
  BEGIN-COBOL-SERVER.
    OPEN INPUT MESSAGE-IN.
    OPEN OUTPUT MESSAGE-OUT SYNCDEPTH 1.
    PERFORM B-TRANS UNTIL CLOSE-FROM-REQUESTOR.
    STOP RUN.

  B-TRANS.
    MOVE SPACES TO ENTRY-REPLY, ENTRY-MSG.
    READ MESSAGE-IN, AT END STOP RUN.
    MOVE PW-HEADER OF MESSAGE-IN TO PW-HEADER OF MESSAGE-OUT.
    IF NAME-IN = "SMITH"
      MOVE 999 TO REPLY-CODE OF ERROR-REPLY
      MOVE 1 TO ERROR-CODE
      WRITE ERROR-REPLY
    ELSE IF NAME-IN = "JONES"
      MOVE 999 TO REPLY-CODE OF ERROR-REPLY
      MOVE 2 TO ERROR-CODE
      WRITE ERROR-REPLY
    ELSE
      MOVE 0 TO REPLY-CODE OF ENTRY-REPLY
      MOVE ENTRY-GROUP TO SERVER-RECORD
      WRITE ENTRY-REPLY.
```

A Advisory Messages

The Pathway terminal control process (TCP) displays messages in the advisory field. The advisory field is an alphanumeric output field defined in the ADVISORY field-characteristic clause of the Screen Section. Messages in this field primarily describe errors detected during input checking. The text of each standard message is a brief indication of the condition that invoked the message.

Messages and Descriptions

The text of the message and the number used internally in the TCP to refer to the message are listed in numeric order. The list of messages provides a reference for those installations that develop their own user conversion procedures.

1

```
REQUIRED FIELD MISSING
```

Cause. The field does not allow zero length input.

2

```
PREVIOUS FIELD MISSING
```

Cause. For a required occurring field with a DEPENDING clause, an occurrence is present, but a previous occurrence was absent.

3

```
EARLIER FIELD MISSING
```

Cause. For an occurring field with a DEPENDING clause, a different occurring field depending on the same item was required but absent for this occurrence number, and this field's occurrence is present.

4

```
FIELD TOO SHORT
```

Cause. The length of the field data, after stripping of fill characters and spaces, is shorter than allowed.

5

```
FIELD NOT CORRECT LENGTH
```

Cause. The input does not have an allowed length.

6

```
FIELD TOO LONG
```

Cause. The input is too long. Generally, this occurs only when the terminal's formatting has been corrupted.

7

```
WRONG FORMAT
```

Cause. Input to an alphanumeric item does not obey the PICTURE clause.

8

```
WRONG FORMAT: DIGIT EXPECTED
```

Cause. Input to an alphanumeric item does not have a digit where a 9 symbol appeared in the PICTURE clause.

9

```
WRONG FORMAT: LETTER EXPECTED
```

Cause. Input to an alphanumeric item does not have a letter or space where an A appeared in the PICTURE clause.

10

```
INVALID NUMBER FORMAT
```

Cause. Input to a numeric item does not obey the PICTURE clause.

11

```
VALUE WRONG
```

Cause. The numeric value input is larger than allowed by the constraints imposed by the field and the receiving data item.

12

```
VALUE INCORRECT
```

Cause. The input value is not allowed by the MUST BE constraints.

13

```
MESSAGE:
```

Cause. This text is used to prefix a tell message.

14

```
DEP OCCUR FLD ERR-INPUT RESTARTED
```

Cause. For multiple screen identifiers in which the OCCURS DEPENDING ON clauses refers to the same dependent data item, one of the following is detected:

- The dependent data item value is greater than the maximum number of elements allowed for one of the screen identifiers.
- A required screen-identifier field occurs fewer times than the value of the associated dependent data item.

This advisory message is displayed only for terminals operating in conversational mode.

15

```
ABORT NOT ALLOWED
```

Cause. The ESCAPE ON ABORT phrase is not present; therefore, the abort-input control character is not effective. ACCEPT processing continues from where the control character is entered.

This advisory message is displayed only for terminals operating in conversational mode.

17

```
WRONG FORMAT: DBCS EXPECTED
```

Cause. The operator entered single-byte (alphanumeric) data into a screen field that requires double-byte data.

18

```
WRONG FORMAT: KATAKANA NOT ALLOWED
```

Cause. Katakana data was entered on a device that has not been configured for Katakana support.

19

```
WRONG FORMAT: INVALID ASIAN CHARACTERS
```

Cause. Translation routines were unable to process the input character stream.

Modifying or Replacing the Advisory Message Routine

You can modify the Pathway advisory message routine or replace it with a routine of your own.

If you want to change the text of the messages or add error messages for use in association with your own user conversion procedures, you can modify the existing advisory-message routine, which is provided in TLIB. This routine is written in the Portable Transaction Application Language (pTAL). A sample source listing (the actual source in TLIB may differ somewhat) is given in Example A-1.

To replace the routine, you must write a replacement procedure in pTAL. Whether you modify or replace the routine, after compilation you must use the `nld` utility to link your procedures in the native TCP user library object file, PATHTCPL.

Note. In releases prior to D40, advisory message routines were written in TAL. In D40 and later releases (including all G-series releases), advisory message routines must be written in pTAL. pTAL is based on TAL. The pTAL language excludes architecture-specific TAL constructs and includes new constructs that replace the architecture-specific constructs. You can write advisory message routines that can be compiled by both the TAL and the pTAL compilers, thus enabling you to use the same source code for different releases of Pathway/iTS.

If you are converting existing alternate advisory message routines to pTAL for use with a D40 or later version of Pathway/iTS, refer to the *pTAL Conversion Guide* and the *pTAL Reference Manual* for further information. Many advisory message routines are simple enough that no changes will be needed.

The declaration for the advisory message procedure is as follows:

```
PROC ADVISORY^MESSAGE( MSGNUM, BUF, MESSLEN );
INT     MSGNUM;   ! THE ERROR NUMBER
STRING .BUF;      ! PLACE MESSAGE HERE
INT    .MESSLEN; ! RETURN MESSAGE LENGTH HERE (MAX 255)
```

The MSGNUM parameter is the internal message number as given in the list of advisory messages in this appendix, or as returned by the user conversion procedure. If new

error-message numbers are to be used (through the use of user conversion procedures), the numbers should be larger than 100 to avoid conflict with future Pathway error numbers.

The BUF parameter is a string buffer where the text associated with MSGNUM should be placed. The text cannot be longer than 255 characters.

The MESSLEN parameter should be set to the length of the text returned.

After revising the standard advisory message procedure or coding a new one, compiling the procedure with the pTAL compiler, and removing all compilation errors, build the user library by using the following command:

```
NLD advisory-msg-object $volume.ZPATHWAY.TCPLIB
-UL-O native-user-library
```

advisory-msg-object

is the pTAL object file.

$*volume*

is the volume where the installation subvolume ZPATHWAY resides.

native-user-library

is the native user library object file used by the TCP.

Example A-1 provides a sample of the source listing for the standard ADVISORY^MESSAGE procedures. This listing can be modified and used for foreign-language versions.

Example A-1. ADVISORY^MESSAGE Source Listing (page 1 of 2)

```
PROC ADVISORY^MESSAGE( ERRNUM, BUF, MESSLEN );
INT     ERRNUM;  ! THE ERROR NUMBER
STRING .BUF;     ! PLACE MESSAGE HERE
INT    .MESSLEN; ! RETURN MESSAGE LENGTH HERE (MAX 255)
BEGIN
   ! RETURN THE MESSAGE (AND ITS LENGTH) FOR THE GIVEN ERROR NUMBER.
   ! THE MESSAGES SHOULD PROBABLY BE LIMITED TO 38 CHARACTERS TO ALLOW
   ! THE FULL MESSAGE TO FIT ON MOST SCREENS.
   !
   INT LEN;
   INT OFFSET;

   STRING MSGTEXT = 'P' := [
                        !....+....1....+....2....+....3....+....4
   !REQMISS       =  1! "REQUIRED FIELD MISSING",
   !PREVMISS      =  2! "PREVIOUS FIELD MISSING",
   !OTHERMISS     =  3! "EARLIER FIELD MISSING",
   !SHORTLEN      =  4! "FIELD TOO SHORT",
   !LENWRONG      =  5! "FIELD NOT CORRECT LENGTH",
   !LONGLEN       =  6! "FIELD TOO LONG",
   !UNEXPECTCHAR  =  7! "WRONG FORMAT",
   !NOTDIGIT      =  8! "WRONG FORMAT: DIGIT EXPECTED",
   !NOTALPHA      =  9! "WRONG FORMAT: LETTER EXPECTED",
                        !....+....1....+....2....+....3....+....4
   !INVALIDNUMFORM= 10! "INVALID NUMBER FORMAT",
   !VALOVFL       = 11! "VALUE WRONG",
   !ILLVAL        = 12! "VALUE INCORRECT",
   !TELLHEAD      = 13! "MESSAGE: ",
   !DEPOCCUR      = 14! "DEP OCCUR FLD ERR-INPUT RESTARTED",
   !CANTABORT     = 15! "ABORT NOT ALLOWED",
   !FIELDABSENT   = 16! "FIELD IS ABSENT",                          !15!
   !NOTALLDBCS    = 17! "WRONG FORMAT: DBCS EXPECTED",              !27!
   !NOKATAKANA    = 18! "WRONG FORMAT: KATAKANA NOT ALLOWED",       !34!
   !INVALIDASIAN  = 19! "WRONG FORMAT: INVALID ASIAN CHARACTERS", !38!
                        " " ];

 LITERAL                      ! OFFSETS INTO MSGTEXT
      OFF^01  = 0,
      OFF^02  = OFF^01 + 22,
      OFF^03  = OFF^02 + 22,
      OFF^04  = OFF^03 + 21,
      OFF^05  = OFF^04 + 15,
      OFF^06  = OFF^05 + 24,
      OFF^07  = OFF^06 + 14,
      OFF^08  = OFF^07 + 12,
      OFF^09  = OFF^08 + 28,
      OFF^10  = OFF^09 + 29,
      OFF^11  = OFF^10 + 21,
      OFF^12  = OFF^11 + 11,
      OFF^13  = OFF^12 + 15,
      OFF^14  = OFF^13 +  9,
      OFF^15  = OFF^14 + 33,
      OFF^16  = OFF^15 + 17,
      OFF^17  = OFF^16 + 15,
      OFF^18  = OFF^17 + 27,
      OFF^19  = OFF^18 + 34,
      OFF^LAST= OFF^19 + 38,
```

Example A-1. ADVISORY^MESSAGE Source Listing (page 2 of 2)

```
        LAST^ERR=     19;          ! LAST ERR # IN TABLE

   INT MSGOFFSET = 'P' := [
               0, OFF^01, OFF^02, OFF^03, OFF^04,
          OFF^05, OFF^06, OFF^07, OFF^08, OFF^09,
          OFF^10, OFF^11, OFF^12, OFF^13, OFF^14,
          OFF^15, OFF^16, OFF^17, OFF^18, OFF^19, OFF^LAST  ];

   IF ERRNUM <= LAST^ERR THEN
      BEGIN
      OFFSET:= MSGOFFSET[ ERRNUM ];
      LEN   := MSGOFFSET[ ERRNUM+1 ] - OFFSET;
      END
   ELSE
      LEN:= 0;
   IF LEN THEN
      BUF ':=' MSGTEXT[ OFFSET ] FOR LEN
   ELSE
      BEGIN
      BUF ':=' "-ERROR ######-";
      CALL NUMOUT( BUF[7], ERRNUM, 10, 6 );
      LEN:= 14
      END;
   MESSLEN:= LEN;
END; !ADVISORY^MESSAGE!
```

B Diagnostic Screens

Diagnostic screens are displayed to inform the terminal operator if an error condition or termination occurs. Diagnostic screens are displayed unless the PATHCOM SET TERM command DIAGNOSTIC parameter is set off. When the parameter is set on (the default setting), the special register DIAGNOSTIC-ALLOWED is initialized to YES.

Screen recovery is invoked following display of a diagnostic screen. This is especially important if the diagnostic screen is displayed because of an error during a PRINT SCREEN sequence. The default diagnostic screen has the following form:

```
PATHWAY ERROR REPORT: timestamp

TERMINAL: termname

diagnostic-message
  [ device-name ]
  [ retry-info  ]
```

- The default value for *device-name* is:

```
PRINTER: filename
```

- The default value for *retry-info* is:

```
PRESS f1 TO RETRY,
      f2 TO ABORT
```

  `f` = F1 for T16-6510, T16-6520, T16-6530, and T16-6540 terminals
  `1` = PA1 for IBM-3270 terminals

  `f` = F2 for T16-6510, T16-6520, T16-6530, and T16-6540 terminals
  `2` = PA2 for IBM-3270 terminals

The standard diagnostic messages and their meanings are as follows.

```
PRINTER BUSY
```

Cause. The print device that is the target of a PRINT SCREEN statement is currently in use.

```
PRINTER REQUIRES ATTENTION
```

Cause. The printer device that is the target of a PRINT SCREEN statement needs to be placed in the READY state.

```
TERMINAL STOPPED BY PROGRAM
```

Cause. The terminal stopped because the highest level program unit was exited.

```
TERMINAL STOPPED BY SYSTEM OPERATOR
```

Cause. The terminal was stopped or aborted by command from the system operator.

```
TERMINAL SUSPENDED BY SYSTEM OPERATOR
```

Cause. The terminal was suspended by command from the system operator.

```
TERMINAL SUSPENDED FOR SYSTEM ERROR
```

Cause. The terminal was suspended because an error occurred during program execution.

```
TERMINAL STOPPED FOR SYSTEM ERROR
```

Cause. The terminal was suspended without possibility of restart because an error occurred during program execution.

Using Alternate Diagnostic Message Routines

You can replace the Pathway generated diagnostic messages with, for example, messages displayed in another language. To change the messages you must replace the PATHTCP routine DIAGNOSTIC^MESSAGE with a user-written routine having the same name. This is handled in the same manner as the ADVISORY^MESSAGE procedure previously described in this appendix.

The declaration for the DIAGNOSTIC^MESSAGE procedure is as follows:

```
PROC DIAGNOSTIC^MESSAGE( DIAG^FORMAT,MESSAGE,MSGLEN,CONTEXT );
INT    .EXT DIAG^FORMAT( DIAG^FORMAT^DEF );
                ! Byte addressable diagnostic info struct.
STRING .MESSAGE; ! Returned --Message to display (byte addr).
INT    .MSGLEN;  ! Returned --Length in bytes of message.
INT    .EXT CONTEXT;
```

The procedure is called repeatedly to initialize the screen, one call for each row of the screen. The parameter DIAG^FORMAT is described in Example B-1 and defines the error condition and the sequencing to build the screen. The parameter CONTEXT provides one word of storage that is not altered between successive calls to initialize a given screen; the parameter is set to zero before the first call in the initialization sequence.

Example B-1. DIAG^FORMAT Parameter for Diagnostic Message Generation

```
STRUCT DIAG^FORMAT^DEF( * );
BEGIN                                   ! - ALL STRING ARRAYS ARE BLANK PADDED.
   STRING CLASS;                        ! CLASS 0 = IBM3270, 1 = T16-6510,
                                        !       2 = T16-6520.
   STRING SUBCLASS;                     ! SUBCLASS FOR IBM-3270 (SCREEN SIZE).
                                        !       0 = 24 X 80 (NOT IBM^3270),
                                        !       1 = 12 X 40,
                                        !       2 = 24 X 80,
                                        !       3 = 24 X 80 - ALT 32 X 80,
                                        !       4 = 24 X 80 - ALT 43 X 80,
                                        !       4 = 24 X 80 - ALT 12 X 80,
   INT    ROW;                          ! ROW OF SCREEN FORMAT [1:NROWS].
                                        !   SEE DIAG^ROW^??? BELOW.
   INT    ERRTYPE;                      ! ERROR TYPE [1:4].
                                        !   SEE DIAG^ERRTYPE^??? BELOW.
   STRING LOG^TERM^NAME[ 0:14 ];        ! TERMINAL PATHWAY NAME.
   STRING TERM^PRINTER[ 0:35 ];         ! PRINTER NAME, EXTERNAL FORM.
   INT    ERRNUM;                       ! ERROR NUMBER OF SUSPENSION CAUSE.
   INT    ERRINFO;                      ! ADDITIONAL ERROR INFO.
   STRING PUNAME[ 0:30 ];               ! PROGRAM-UNIT NAME.
   INT    PUVERSION;                    ! VERSION OF PROGRAM UNIT.
   INT    INSTR^ADDR;                   ! ADDRESS OF INSTRUCTION AT SUSP.
   STRING INSTR^CODE[ 0:19 ];           ! INSTRUCTION AT SUSPENSION.
   INT    CONTEXT;                      ! ONE WORD OF USER CONTEXT.
 END;

LITERAL ! PATHWAY DEFINED DIAGNOSTIC DISPLAY ROWS.
                          !                  1         2         3       3
   DIAG^ROW^NULL      = 0, !           1........0.........0.........0.......8
   DIAG^ROW^HEADER    = 1, ! HEADER - "PATHWAY(TM) ERROR REPORT ddMONyy,hh:mm"
   DIAG^ROW^TERMNAME  = 3, ! TERM   - "TERM: term-name"
   DIAG^ROW^ERRTYPE   = 5, ! ERROR  - "TERMINAL STOPPED BECAUSE OF ERROR".
   DIAG^ROW^DEVNAME   = 6, ! DEVNAME- "  PRINTER: $LP".
   DIAG^ROW^RETRYINFO = 7; ! RETRY  - "  PRESS F1 TO RETRY, F2 TO ABORT"

LITERAL ! DIAGNOSTIC DISPLAY ERROR TYPES.
   DIAG^ERRTYPE^STOP^BY^PROG    = 1, ! TERM STOPPED BY PROGRAM.
   DIAG^ERRTYPE^STOP^BY^OP      = 2, ! TERM STOPPED BY OPERATOR.
   DIAG^ERRTYPE^ABRT^BY^OP      = 3, ! TERM ABORTED BY OPERATOR.
   DIAG^ERRTYPE^SUSP^BY^ERR     = 4, ! TERM SUSPENDED BECAUSE OF ERROR.
   DIAG^ERRTYPE^SUSP^BY^ERR^NRS = 5, ! TERM SUSPENDED BECAUSE OF ERROR, NOT
                                     !   RESUMABLE.
   DIAG^ERRTYPE^SUSP^BY^OP      = 6, ! TERM SUSPENDED BY OPERATOR.
   DIAG^ERRTYPE^NOT^READY       = 7, ! PRINTER NOT READY.
   DIAG^ERRTYPE^BUSY            = 8; ! PRINTER BUSY.
```

Modifying Diagnostic Messages

You can modify the source listing of the standard DIAGNOSTIC^MESSAGE procedures for foreign language versions. An example of this source listing is shown in Example B-2.

Example B-2. DIAGNOSTIC^MESSAGE Source Listing (page 1 of 3)

```
PROC DIAGNOSTIC^MESSAGE( DIAG^FORMAT, MESSAGE, MSGLEN, CONTEXT );
INT    .EXT DIAG^FORMAT ( DIAG^FORMAT^DEF ); ! DIAGNOSTIC !
STRING .MESSAGE; ! RETURNED - MESSAGE TO BE DISPLAYED (IN LOWER 32K).
INT    .MSGLEN;  ! RETURNED - LENGTH IN BYTES OF MESSAGE.
INT    .EXT CONTEXT; ! ONE WORD OF "OWN" STORAGE.  SET TO ZERO ON FIRST
                     ! CALL OF DIAGNOSTIC DISPLAY SEQUENCE. !
BEGIN
   !---------------------------------------------------------------------------
   !
   ! This procedure is used to format a row of diagnostic display text.
   ! It is called once for each row of the display.
   !
   !---------------------------------------------------------------------------
   STRING .S;
   INT(32) S32;  ! receives MOVE address !
   INT TS[0:2];

INT SUBPROC ASCII^TIMESTAMP ( TS , ARRAY ) ;
INT    .TS;    ! TIMESTAMP TO BE CONVERTED.
STRING .ARRAY; ! TARGET OF CONVERSION.
BEGIN
   !---------------------------------------------------------------------------
   !
   ! THIS PROCEDURE CONVERTS THE TIMESTAMP FROM INTERNAL TO THE FOLLOWING
   ! EXTERNAL FORM:  "ddMMMyy,hh:mm:ss"
   !
   ! RETURNS - LENGTH OF TIMESTAMP STRING.
   !
   !---------------------------------------------------------------------------
   INT T[0:6]; ! RETURN ARRAY FOR INTERNAL TIME.

   STRING  MONTH[3:38] = 'P' := "JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC";

   ARRAY := " ";
   ARRAY[1] ':=' ARRAY FOR 15;
   CALL CONTIME( T , TS[0] , TS[1] , TS[2] );
   CALL NUMOUT(ARRAY,T[2],10,2);       ! DAY.
   ARRAY[2] ':=' MONTH[T[1]*3] FOR 3;  ! MONTH.
   CALL NUMOUT(ARRAY[5],T,10,4);       ! YEAR.
   ARRAY[5] ':=' ARRAY[7] FOR 2;
   ARRAY[7] := ",";                    ! ,
   CALL NUMOUT(ARRAY[8],T[3],10,2);    ! HOUR.
   ARRAY[10] := ":";                   ! :
   CALL NUMOUT(ARRAY[11],T[4],10,2);   ! MIN.
   ARRAY[13] := ":";                   ! :
   CALL NUMOUT(ARRAY[14],T[5],10,2);   ! SEC.
   RETURN 16;
END;! ASCII^TIMESTAMP!
```

Example B-2. DIAGNOSTIC^MESSAGE Source Listing (page 2 of 3)

```
MSGLEN:= 0;
 IF DIAG^FORMAT.ROW = DIAG^ROW^HEADER THEN
    BEGIN
    MESSAGE ':=' "PATHWAY ERROR REPORT: " -> @S;
    CALL TIMESTAMP( TS );
    @S:= @S[ ASCII^TIMESTAMP( TS, S ) ];
    MSGLEN:= @S '-' @MESSAGE;
    END
 ELSE
 IF DIAG^FORMAT.ROW = DIAG^ROW^TERMNAME THEN
    BEGIN
    MESSAGE ':=' "TERMINAL: " -> @S;
    S ':=' DIAG^FORMAT.LOG^TERM^NAME FOR 15 -> S32;
    ! Compiler required INT(32) to hold address here. !
    @S := $INT ( S32 );
    RSCAN S[-1] WHILE " " -> @S;
    MSGLEN:= @S[ 1 ] '-' @MESSAGE;
    END
 ELSE
 IF DIAG^FORMAT.ROW = DIAG^ROW^ERRTYPE THEN
    BEGIN           !         1         2         3       3
                    !1........0.........0.........0.......8
    IF DIAG^FORMAT.ERRTYPE = DIAG^ERRTYPE^STOP^BY^PROG THEN
       MESSAGE ':=' "TERMINAL STOPPED BY PROGRAM" -> @S
    ELSE
    IF DIAG^FORMAT.ERRTYPE = DIAG^ERRTYPE^STOP^BY^OP OR
       DIAG^FORMAT.ERRTYPE = DIAG^ERRTYPE^ABRT^BY^OP THEN
       MESSAGE ':=' "TERMINAL STOPPED BY SYSTEM OPERATOR" -> @S
    ELSE
    IF DIAG^FORMAT.ERRTYPE = DIAG^ERRTYPE^SUSP^BY^ERR THEN
       MESSAGE ':=' "TERMINAL SUSPENDED FOR SYSTEM ERROR" -> @S
    ELSE
    IF DIAG^FORMAT.ERRTYPE = DIAG^ERRTYPE^SUSP^BY^ERR^NRS THEN
       MESSAGE ':=' "TERMINAL STOPPED FOR SYSTEM ERROR" -> @S
    ELSE
    IF DIAG^FORMAT.ERRTYPE = DIAG^ERRTYPE^SUSP^BY^OP THEN
       MESSAGE ':=' "TERMINAL SUSPENDED BY SYSTEM OPERATOR" -> @S
    ELSE
    IF DIAG^FORMAT.ERRTYPE = DIAG^ERRTYPE^NOT^READY THEN
       MESSAGE ':=' "PRINTER REQUIRES ATTENTION" -> @S
    ELSE
    IF DIAG^FORMAT.ERRTYPE = DIAG^ERRTYPE^BUSY THEN
       MESSAGE ':=' "PRINTER BUSY" -> @S;
    MSGLEN:= @S '-' @MESSAGE;
    END
```

Example B-2. DIAGNOSTIC^MESSAGE Source Listing (page 3 of 3)

```
      ELSE
        IF DIAG^FORMAT.ROW = DIAG^ROW^DEVNAME AND
           ( DIAG^FORMAT.ERRTYPE = DIAG^ERRTYPE^NOT^READY OR
             DIAG^FORMAT.ERRTYPE = DIAG^ERRTYPE^BUSY ) THEN
           BEGIN
           IF DIAG^FORMAT.TERM^PRINTER <> " " THEN
              BEGIN
              MESSAGE ':=' "  PRINTER: " -> @S;
              S ':=' DIAG^FORMAT.TERM^PRINTER FOR 36 -> S32;
              ! Compiler required INT(32) to hold address here. !
              @S := $INT ( S32 );
              RSCAN S[-1] WHILE " " -> @S;
              MSGLEN:= @S[1] '-' @MESSAGE;
              END;
           END
        ELSE
        IF DIAG^FORMAT.ROW = DIAG^ROW^RETRYINFO AND
           ( DIAG^FORMAT.ERRTYPE = DIAG^ERRTYPE^NOT^READY OR
             DIAG^FORMAT.ERRTYPE = DIAG^ERRTYPE^BUSY ) THEN
           BEGIN
           MESSAGE ':=' "  PRESS " -> @S;
           IF DIAG^FORMAT.CLASS = 0 THEN
              S ':=' "PA1" -> @S
           ELSE
              S ':=' "F1" -> @S;
           S ':=' " TO RETRY, " -> @S;
           IF DIAG^FORMAT.CLASS = 0 THEN
              S ':=' "PA2" -> @S
           ELSE
              S ':=' "F2" -> @S;
           S ':=' " TO ABORT" -> @S;
           MSGLEN:= @S '-' @MESSAGE;
           END;

      END;!DIAGNOSTIC^MESSAGE!
```

C

SCREEN COBOL Compiler Diagnostic Messages

The SCREEN COBOL compiler produces three types of diagnostic messages to report the severity of problems in the source text or compilation process:

- Warnings—A warning message reports a questionable condition but does not inhibit code generation. Some warnings merely report a minor deviation from the conventions of the SCREEN COBOL language. Other warnings indicate more important violations that could result in a different interpretation of the program than is intended. The explanation of a warning includes a brief description and any actions taken or assumptions made by the compiler. Warning messages can be suppressed with the NOWARN compiler command, as described in Section 7, Compilation.
- Errors—An error message reports a serious violation of SCREEN COBOL syntax or semantics. The compiler stops generating code and deletes any previously generated code, but compilation continues for syntax checking purposes. Because information at this point would be incomplete or incorrect, correct syntax might be reported as an error.
- Failures—A failure message reports a condition so severe that the compiler cannot continue. Any previously generated code is deleted.

Most warnings or errors pertain to a specific portion of the source text or a specific user-defined item. The compiler assists in locating the error as follows:

- When the problem is local, the line preceding the message contains a caret (^). The language element in error is in the last source line either at the position indicated or somewhere to the left of that position. Occasionally, the language element to the left is actually on a source line preceding the last one listed.
- Some problems are not found until the entire program is examined. When the line preceding the message contains the phrase PROBLEM AT OR NEAR LINE nnnnn, it refers to a preceding portion of the program by line number. The cause of the problem, or one of several interrelated causes, is found in the vicinity of the specified line.
- When a user-defined name appears at the end of a message, the message concerns the item specified.

SCREEN COBOL compiler error messages are written to the file or printer device specified by the OUT parameter of the SCOBOLX run command. In addition, a failure message is sent to the home terminal. The explanations describe the problem in further detail or describe the language rule violated. When the same message can sto different problems, the discussion includes several independent explanations.

All messages carry a number, are preceded by the word FAILURE, WARNING, or ERROR, and are surrounded by asterisks. For example:

```
** FAILURE nnn **
** WARNING nnn **
** ERROR nnn **
```

0

```
TOO MANY ERRORS
```

Type. Failure

Cause. The number of error diagnostics exceeds the limit specified. The default limit is 100.

1

```
UNABLE TO INVOKE COMPILER PROCESS
```

Type. Failure

Cause. The compiler is unable to invoke one of its processes. The error code returned by the NEWPROCESS procedure (bits 0 through 7) is appended to the message.

2

```
UNABLE TO OPEN $RECEIVE
```

Type. Failure

Cause. The compiler is unable to open the job communication file. The error code returned by the operating system is appended to the message.

3

```
UNABLE TO OPEN COMMUNICATION FILE
```

Type. Failure

Cause. The compiler is unable to open the interprocess communication file. The error code returned by the operating system is appended to the message.

4

```
UNABLE TO OPEN (SOURCE/LIST) FILE
```

Type. Failure

Cause. The compiler is unable to open the specified file. The error code returned by the operating system is appended to the message.

5

```
UNABLE TO USE (SOURCE/LIST) FILE
```

Type. Failure

Cause. The message applies to one of the following conditions:

- The source file does not have read capability, or the list file does not have write capability.
- Access to the source file (an EDIT file) failed. The error code returned from the attempt to access the file is appended to the message.
- The record length of the list file is less than 40 bytes, or the list device is a printer or process and the initial control operation failed.

6

```
UNABLE TO CREATE WORK FILE
```

Type. Failure

Cause. The compiler is unable to create one of its work files. The error code returned by the operating system is appended to the message.

7

```
UNABLE TO OPEN WORK FILE
```

Type. Failure

Cause. The compiler is unable to open one of its work files. The error code returned by the operating system is appended to the message.

8

```
UNABLE TO OPEN COPY FILE
```

Type. Failure

Cause. The compiler is unable to open a COPY library file. The error code returned by the operating system is appended to the message.

9

```
UNABLE TO USE COPY FILE
```

Type. Failure

Cause. The message applies to one of the following conditions:

- The default COPY library file name is not a legal file name.
- The COPY library file is not an EDIT file or has been modified since the start of this compilation.
- An attempt to access the COPY library failed. The error code returned from the attempt to access the file is appended to the message.

10

```
COMPILER COMMUNICATION LOST
```

Type. Failure

Cause. Communication between compiler processes failed. The error code returned by the operating system is appended to the message. If the code is 0, one of the compiler processes has abended.

11

```
(SOURCE/LIST) FILE (READ/WRITE) FAILURE
```

Type. Failure

Cause. The compiler is unable to access the specified file. The error code returned by the operating system is appended to the message.

12

```
SOURCE FILE EDITREAD FAILURE
```

Type. Failure

Cause. A read issued to the source file failed. The error code returned from the attempt to access the file is appended to the message.

13

```
COPY FILE EDITREAD FAILURE
```

Type. Failure

Cause. A read issued to the COPY library file failed. The error code returned from the attempt to access the file is appended to the message.

14

```
UNABLE TO CREATE RUN UNIT FILE
```

Type. Failure

Cause. The compiler is unable to create the object file. The error code returned by the operating system is appended to the message.

15

```
UNABLE TO OPEN RUN UNIT FILE
```

Type. Failure

Cause. The compiler is unable to open the object file. The error code returned by the operating system is appended to the message.

16

```
FAILURE IN USING OR ALLOCATING EXTENDED SEGMENT
```

Type. Failure

Cause. The compiler cannot allocate an extended segment because of insufficient memory or disk space.

17

```
COMPILER LOGIC ERROR
```

Type. Failure

Cause. Internal consistency checking has discovered an error in the compiler logic. Report this failure to your service provider.

18

```
DICTIONARY OVERFLOW
```

Type. Failure

Cause. Compiler dictionary space is insufficient for the number of items defined in the current program unit. The deficiency might be corrected by invoking the SCREEN COBOL compiler with a larger value for the MEM parameter. If the failure persists when MEM 64 is used, the program must be subdivided into smaller program units.

19

```
FILE ERROR ON WORK FILE
```

Type. Failure

Cause. An operation on a compiler work file failed. The error code returned by the operating system is appended to the message.

20

```
RUN UNIT OVERFLOW
```

Type. Failure

Cause. The union of the code space and data descriptor area has overflowed. The user data space which is in Working-Storage is not monitored by the compiler.

21

```
CONTROL DATA SPACE OVERFLOW
```

Type. Failure

Cause. For each program unit, the SCREEN COBOL compiler allocates an auxiliary data space used for control purposes. The cumulative requirements for these control data spaces exceed the maximum available to SCREEN COBOL.

22

```
PROGRAM CODE SPACE OVERFLOW
```

Type. Failure

Cause. Either the code requirements for a single program unit or the cumulative requirements for the entire object file exceed the maximum code space available to SCREEN COBOL.

23

```
FILE ERROR ON RUN UNIT FILE
```

Type. Failure

Cause. An operation on the object file failed. The error code returned by the operating system is appended to the message.

25

```
INTERPROCESS MESSAGE OVERFLOW
```

Type. Failure

Cause. The length of the source line exceeds the input buffer size.

26

```
MISSING QUOTE CHARACTER
```

Type. Error

Cause. The terminating quotation mark is missing from a nonnumeric literal.

27

```
NULL LITERAL
```

Type. Error

Cause. A nonnumeric literal contains no characters (that is, has no value).

28

```
LITERAL EXCEEDS 120 CHARACTERS
```

Type. Error

Cause. A nonnumeric literal contains more than 120 characters.

29

```
LITERAL EXCEEDS 18 DIGITS
```

Type. Error

Cause. A numeric literal contains more than 18 digits.

30

```
WORD EXCEEDS 30 CHARACTERS
```

Type. Error

Cause. A SCREEN COBOL word contains more than 30 characters.

31

```
NOT SUPPORTED
```

Type. Warning

Cause. SCREEN COBOL does not support some of the optional elements of the ANSI COBOL language. The message probably refers to one of the following language elements, which are not normally critical to correct program execution:

- The Rerun facility
- File labels—SCREEN COBOL does not have file-handling capability
- More than one system name appears in an ASSIGN clause (NO ASSIGN)

31

```
NOT SUPPORTED
```

Type. Error

Cause. A SCREEN COBOL does not support some of the optional elements of the ANSI COBOL language.

32

```
ILLEGAL CONTEXT FOR RESERVED WORD
```

Type. Warning

Cause. A SCREEN COBOL reserved word is used as the text name or library name in a COPY statement.

32

```
ILLEGAL CONTEXT FOR RESERVED WORD
```

Type. Error

Cause. The indicated SCREEN COBOL reserved word cannot appear in this context. The cause for this message might be an attempt to define one of the reserved words as a user-defined name.

33

```
ILLEGAL CHARACTER
```

Type. Error

Cause. The character indicated is not permitted in this context. Since the character might be unprintable, the internal value of the character is listed with the message.

34

```
TOKEN EXCEEDS 120 CHARACTERS
```

Type. Error

Cause. An entry considered to be a character string contains more than 120 characters. If the character string is actually several adjacent language elements, you can correct the problem by inserting blanks to separate the elements.

35

```
BLANK CONTINUATION LINE
```

Type. Warning

Cause. A source line marked as a continuation line contains only blanks.

36

```
ILLEGAL INDICATOR CHARACTER
```

Type. Warning

Cause. The message applies to one of the following conditions:

- The character in the indicator field of a source line is not - * / ? (hyphen, asterisk, slash, or question mark) or a blank space.
- A continuation line appears as part of a comment entry in a paragraph of the Identification Division.

37

```
MISSING SEPARATOR
```

Type. Warning

Cause. The message applies to one of the following conditions:

- A character string is not followed by a separator.
- A comma, semicolon, or period separator is not followed by a blank space.

38

```
UNEXPECTED TEXT
```

Type. Error

Cause. The message applies to one of the following conditions:

- A section header or division header is followed by other text on the same source line.
- The program name in the PROGRAM-ID paragraph of the Identification Division is followed by other text on the same source line.
- The Identification Division header or the PROGRAM-ID paragraph must be followed by an Identification Division paragraph header or the Environment Division header, and it must begin in Area A of the source line.

39

```
UNEXPECTED END OF TEXT
```

Type. Error

Cause. The source text ended before the appearance of all four required divisions.

40

```
INCORRECT NUMBER OF PARAMETERS
```

Type. Error

Cause. The number of operands in the USING clause of a CALL statement differs from the number of names in the USING Division header for the SCREEN COBOL subprogram it invokes.

41

```
NAME CONFLICT
```

Type. Error

Cause. The message applies to one of the following conditions:

- The definition of a user-defined name in one class conflicts with its previous definition in another class.
- The name of a new data item cannot be distinguished from the name of a previous data item, even with full qualification.

42

```
AMBIGUOUS REFERENCE
```

Type. Error

Cause. A reference has insufficient qualification to identify a unique object within the program unit.

43

```
NON-STANDARD ALPHABET NOT SUPPORTED
```

Type. Warning

Cause. A clause specifies EBCDIC, which is not supported.

43

```
NON-STANDARD ALPHABET NOT SUPPORTED
```

Type. Error

Cause. The message applies to one of the following conditions:

- The program's collating sequence is not standard.
- An integer value is out of the range of valid values; the range depends on the specific context.

44

```
SYNTAX ERROR DETECTED AT TOKEN: xxxxx
```

Type. Error

Cause. A syntax error occurred at the token pointed to by the indicator line. The compiler attempted to correct token *xxxxx* but could not. The compiler examined and discarded subsequent source code until reaching the token indicated by error 48.

45

```
SYNTAX ERROR REPLACING UNEXPECTED TOKEN BY xxxxx
```

Type. Error

Cause. A syntax error occurred at the token pointed to by the indicator line. The compiler tried to correct the error by inserting token *xxxxx*. Verify the compiler's correction for accuracy in your program.

46

```
SYNTAX ERROR - INSERTING MISSING TOKEN: xxxxx
```

Type. Error

Cause. A syntax error occurred at the token pointed to by the indicator line. The compiler corrected the error by inserting token *xxxxx*. Usually, the correction is one of several choices. Verify the compiler's correction for accuracy in your program.

47

```
SYNTAX ERROR - DELETING UNEXPECTED TOKEN: xxxxx
```

Type. Error

Cause. A syntax error occurred at the token pointed to by the indicator line. The compiler attempted to correct the error by deleting token *xxxxx*. Verify the compiler's correction for accuracy in your program.

48

```
PARSING RESUMED AT TOKEN: xxxxx
```

Type. Warning

Cause. This message follows errors 44, 45, 46, and 47. It indicates that in attempting recovery, the compiler ignored all tokens between the original error token and token xxxxx (but not including token *xxxxx*). When this message occurs, error correction by the compiler is usually not the preferred correction. Examine the program from the original error to determine the appropriate correction.

49

```
END-OF-FILE ENCOUNTERED DURING RECOVERY
```

Type. Failure

Cause. A syntax error has probably occurred near the end of the source code. Error 44 indicates where the error occurred. There is insufficient source code following the error for the compiler to make a correction.

50

```
EXPECTED 'IDENTIFICATION'
```

Type. Error

Cause. A SCREEN COBOL program unit must begin with an Identification Division header. The reserved word IDENTIFICATION must start in Area A of the source line.

51

```
EXPECTED UNSIGNED INTEGER
```

Type. Error

Cause. The message applies to one of the following conditions:

- A numeric literal in this context must be an unsigned integer.
- Only an unsigned integer numeric literal is permitted in this context.

52

```
0 NOT PERMITTED IN THIS CONTEXT
```

Type. Error

Cause. The indicated integer numeric literal cannot be zero in this context.

53

```
INTEGER NOT IN EXPECTED RANGE
```

Type. Error

Cause. The message applies to one of the following conditions:

- The value of the integer numeric literal is too small for this context.
- The value of the integer numeric literal is too large for this context.

54

```
ILLEGAL RANGE
```

Type. Error

Cause. The message applies to one of the following conditions:

- The first value in a numeric range exceeds the last value.
- The first value in a nonnumeric range is greater than the last value.

55

```
OUT OF ORDER
```

Type. Warning

Cause. The position of a phrase, clause, or paragraph does not conform to SCREEN COBOL language requirements.

55

```
OUT OF ORDER
```

Type. Error

Cause. The message applies to one of the following conditions:

- The REDEFINES clause must be the first clause in a data description entry.
- A section of the Data Division occurs out of order.

56

```
DUPLICATE PHRASE
```

Type. Error

Cause. The indicated phrase duplicates the function of a preceding one.

57

```
DUPLICATE CLAUSE
```

Type. Error

Cause. The indicated clause duplicates the function of a preceding one.

58

```
DUPLICATE PARAGRAPH
```

Type. Error

Cause. The indicated paragraph header duplicates the function of a preceding one.

59

```
DUPLICATE SECTION
```

Type. Error

Cause. The indicated section header duplicates the function of a preceding one.

61

```
EXPECTED COMMAND WORD
```

Type. Error

Cause. A compiler command line must begin with the keyword of a command or with one of the command options defined for the OPTION command.

62

```
EXPECTED QUOTED STRING
```

Type. Error

Cause. The heading value in a HEADING command option must be a quoted string (that is, a nonnumeric literal).

63

```
EXPECTED COMMA
```

Type. Error

Cause. The message applies to one of the following conditions:

- Compiler command options must be separated by commas.
- Toggle numbers in a SETTOG or RESETTOG command must be separated by commas.

64

```
MISSING TEXT NAME
```

Type. Error

Cause. The message applies to one of the following conditions:

- The text name is missing from a SECTION command.
- The text name in a COPY statement cannot be found in the copy library.

65

```
COMMAND NOT PERMITTED AFTER OPTION
```

Type. Error

Cause. A command keyword follows one or more command options. Only a single command is permitted on each command line.

66

```
TEXT NOT PERMITTED AFTER COMMAND
```

Type. Error

Cause. Additional text follows a complete command. Only a single command is permitted on each command line.

68

```
COMMAND NOT COMPATIBLE WITH PREVIOUS COMMANDS
```

Type. Warning

Cause. The SYNTAX and SYMBOLS commands are both used.

70

```
MISSING PROGRAM ID
```

Type. Error

Cause. The required PROGRAM-ID paragraph of the Identification Division is missing.

71

```
MISSING CONFIGURATION SECTION
```

Type. Error

Cause. The required Configuration Section of the Environment Division is missing.

72

```
MISSING SOURCE COMPUTER PARAGRAPH
```

Type. Warning

Cause. The Configuration Section should contain the SOURCE-COMPUTER paragraph. The default is SOURCE-COMPUTER. TANDEM/16.

73

```
MISSING OBJECT COMPUTER PARAGRAPH
```

Type. Warning

Cause. The Configuration Section should contain the OBJECT-COMPUTER paragraph.

75

```
ILLEGAL EXTERNAL FILE NAME FORM
```

Type. Error

Cause. An attempt to convert an external file name to an internal file name has failed.

76

```
MISSING CONDITION NAME
```

Type. Warning

Cause. A switch has been encountered without a condition name.

77

```
ILLEGAL CURRENCY SYMBOL
```

Type. Error

Cause. Either the alternative currency symbol specified is not a single character, or the specified character is not among the set of characters permitted for this purpose.

94

```
ALPHABET NAME NOT FOUND
```

Type. Error

Cause. The indicated name is either not defined or not an alphabet name.

111

```
CLAUSE NOT PERMITTED FOR THIS ENTRY
```

Type. Error

Cause. The indicated clause appears in an entry whose level number prohibits it.

112

```
NOT PERMITTED IN THIS SECTION
```

Type. Error

Cause. A VALUE clause defining an initial value appears in the Linkage Section or the Data Division.

113

```
ILLEGAL LEVEL NUMBER
```

Type. Error

Cause. A level number is not 66, 77, 88, or in the range 01 through 49. The compiler converts the illegal number to 50.

114

```
INCONSISTENT LEVEL NUMBER
```

Type. Error

Cause. A level number is neither greater than the level number of the preceding data description entry nor equal to that of some preceding data description entry in the same data structure.

115

```
MISSING 01 LEVEL ENTRY
```

Type. Error

Cause. A level number in the range 02 through 49 is not subordinate to a data description entry with level number 01; that is, it is not within a data structure.

116

```
PRECEDED BY VARIABLE OCCURRENCE TABLE
```

Type. Error

Cause. Within a data structure, the data description entry for a variable occurrence table (one containing an OCCURS clause with a range) cannot be followed by a data description entry with a lower level number.

117

```
NOT PRECEDED BY CONDITIONAL VARIABLE
```

Type. Error

Cause. The definition of a condition-name (that is, a name whose data description entry has level number 88) must be preceded by the entry of the data item whose value it tests. Any intervening data description entries must also have level number 88.

118

```
NOT PRECEDED BY RECORD
```

Type. Error

Cause. A data description entry with level number 66 must be preceded by a data structure. Any intervening data description entries must also have level number 66.

119

```
FILLER PERMITTED ONLY FOR ELEMENTARY RECORD ITEM
```

Type. Error

Cause. The data description entry of a FILLER data item must have a level number in the range 02-49 and cannot be followed by descriptions of subordinate data items; that is, it must be an elementary data item defined within a data structure.

120

```
DO NOT QUOTE PICTURE STRING
```

Type. Warning

Cause. A PICTURE character string should not be written as a nonnumeric literal. The SCREEN COBOL compiler accepts the contents of the nonnumeric literal as the PICTURE character string.

121

```
PICTURE STRING EXCEEDS 30 CHARACTERS
```

Type. Error

Cause. The SCREEN COBOL language limits the representation of a PICTURE character string to 30 characters. Data items with more than 30 character positions must be described with parenthesized repetition counts.

122

```
TOO MANY DIGIT POSITIONS
```

Type. Error

Cause. The SCREEN COBOL language supports a maximum of 18 digits in a numeric or numeric edited data item.

123

```
TOO MANY CHARACTER POSITIONS
```

Type. Error

Cause. SCREEN COBOL supports a maximum of 32,000 character positions for an elementary data item.

124

```
ILLEGAL PICTURE STRING
```

Type. Error

Cause. The PICTURE character string does not conform to SCREEN COBOL syntax. Some of the causes for this message are illegal characters, unmatched parentheses, improper combinations of otherwise legal characters, and pictures with no positions for data characters.

125

```
LAST SYMBOL IS ',' OR '.'
```

Type. Warning

Cause. After removing the terminating comma, semicolon, period, or blank, the last character in the PICTURE character in the PICTURE character string is a comma or decimal point. The compiler accepts the picture and interprets the character in conformance with the presence or absence of the DECIMAL-POINT IS COMMA clause in the SPECIAL-NAMES paragraph.

126

```
PICTURE NOT PERMITTED FOR INDEX ITEM
```

Type. Error

Cause. An indexed item may not have a PICTURE clause.

127

```
SUBORDINATE USAGE CONFLICTS WITH GROUP USAGE
```

Type. Error

Cause. The data description entry for a containing group item has a USAGE clause. The description of the subordinate data item cannot specify a different usage.

128

```
DISPLAY USAGE REQUIRED IN GROUP WITHOUT VALUE OR CONDITION
NAME
```

Type. Error

Cause. The data description entry of a containing group item has a VALUE clause specifying an initial value or is followed by entries defining condition-names for the group item. The subordinate data item must have DISPLAY usage.

129

```
COMPUTATIONAL USAGE REQUIRES NUMERIC
```

Type. Error

Cause. The category of a data item must be numeric when its usage is COMPUTATIONAL.

130

```
SUBORDINATE SIGN CONFLICTS WITH GROUP SIGN
```

Type. Error

Cause. The data description for a containing group item has a SIGN clause. The description of the subordinate data item cannot specify different sign characteristics.

131

```
SIGN CLAUSE REQUIRES DISPLAY USAGE
```

Type. Error

Cause. The message applies to one of the following conditions:

- The data description entry for a containing group item has a SIGN clause. The subordinate numeric data item is signed (that is, has an S in its PICTURE); therefore, the item must have DISPLAY usage.
- The data description entry for the current data item has a SIGN clause; therefore, the item must have DISPLAY usage.

132

```
SIGN CLAUSE REQUIRES SIGNED NUMERIC
```

Type. Error

Cause. The data description entry for the current data item has a SIGN clause; therefore, the item PICTURE must specify category numeric and contain an S.

133

```
JUSTIFIED REQUIRES DISPLAY USAGE
```

Type. Error

Cause. A data item described with the JUSTIFIED clause must have DISPLAY usage.

134

```
JUSTIFIED NOT PERMITTED FOR NUMERIC OR EDITED
```

Type. Error

Cause. The JUSTIFIED clause cannot appear for a data item described as numeric.

135

```
JUSTIFIED NOT PERMITTED IN GROUP WITH VALUE OR CONDITION NAME
```

Type. Error

Cause. The data description entry of a containing group item has a VALUE clause specifying an initial value or is followed by entries defining condition names for the group item. The subordinate data item cannot be described with the JUSTIFIED clause.

136

```
SYNCHRONIZED NOT PERMITTED FOR INDEX ITEM
```

Type. Warning

Cause. An indexed item may not have a SYNCHRONIZED clause.

137

```
SYNCHRONIZED NOT PERMITTED IN GROUP WITH VALUE OR CONDITION
NAME
```

Type. Error

Cause. The data description entry of a containing group item has a VALUE clause specifying an initial value or is followed by entries defining condition names for the group item. The subordinate data item cannot be described with the SYNCHRONIZED clause.

138

```
BLANK WHEN ZERO REQUIRES DISPLAY USAGE
```

Type. Error

Cause. A data item described with the BLANK WHEN ZERO clause must have DISPLAY usage. BLANK WHEN ZERO syntax is enforced when used, but data items using this syntax cannot be accessed by SCREEN COBOL programs.

139

```
BLANK WHEN ZERO REQUIRES NUMERIC EDITED
```

Type. Error

Cause. Only a numeric data item can be described with the BLANK WHEN ZERO clause. BLANK WHEN ZERO syntax is enforced when used, but data items using this syntax cannot be accessed by SCREEN COBOL programs.

140

```
BLANK WHEN ZERO NOT COMPATIBLE WITH '*'
```

Type. Error

Cause. A data item cannot be described with both the BLANK WHEN ZERO clause and a picture containing the asterisk (*). BLANK WHEN ZERO syntax is enforced when used, but data items using this syntax cannot be accessed by SCREEN COBOL programs.

141

```
TOO MANY NESTED TABLES
```

Type. Error

Cause. The SCREEN COBOL language supports access to a data item with at most three subscripts. The OCCURS clause is subordinate to three or more other OCCURS clauses and would require four or more subscripts to access the data item it describes.

142

```
VARIABLE OCCURRENCE NOT PERMITTED FOR SUBORDINATE TABLE
```

Type. Error

Cause. The SCREEN COBOL language does not permit a variable occurrence table to be subordinate to a group table item.

143

```
VARIABLE OCCURRENCE NOT PERMITTED IN REDEFINITION
```

Type. Error

Cause. A data item described in a redefinition cannot be a variable occurrence table.

144

```
VARIABLE OCCURRENCE NOT COMPATIBLE WITH GROUP INITIAL VALUE
```

Type. Error

Cause. The data description entry of a containing group item has an initial value. The subordinate data item cannot be a variable occurrence table.

145

```
TOO MANY KEYS
```

Type. Error

Cause. There are more than 30 keys.

146

```
SUBORDINATE VALUE NOT PERMITTED WITH GROUP VALUE
```

Type. Error

Cause. The data description entry of a containing group item specifies an initial value for the group item. The subordinate data item cannot also specify an initial value.

147

```
ONLY ONE INITIAL VALUE PERMITTED
```

Type. Error

Cause. A data item cannot be initialized with more than one value.

148

```
RANGE NOT PERMITTED FOR INITIAL VALUE
```

Type. Error

Cause. A data item cannot be initialized with a range of values.

149

```
INITIAL VALUE OR CONDITION NAME NOT PERMITTED FOR INDEX ITEM
```

Type. Error

Cause. An indexed item may not have a VALUE clause.

150

```
INITIAL VALUE NOT PERMITTED FOR TABLE ITEM
```

Type. Error

Cause. A data item that is described with an OCCURS clause or is subordinate to a group table item cannot be initialized.

151

```
INITIAL VALUE NOT PERMITTED FOR REDEFINITION
```

Type. Error

Cause. A data item described in a redefinition cannot be initialized.

152

```
SIGNIFICANCE RANGE OF LITERALS EXCEEDS 18 DIGITS
```

Type. Error

Cause. The number of significant digits of a numeric literal exceeds 18.

153

```
NUMERIC LITERAL NOT COMPATIBLE WITH NONNUMERIC FIGURATIVE OR
LITERAL
```

Type. Error

Cause. When a VALUE clause contains a numeric literal, all other values must also be numeric literals or one of the figurative constants ZERO, ZEROS, or ZEROES.

154

```
RENAME OBJECT NOT DATA ITEM
```

Type. Error

Cause. The RENAMES clause can only rename data items.

155

```
RENAME OBJECT IS 66 LEVEL ITEM
```

Type. Error

Cause. A RENAMES clause cannot rename a level 66 item.

156

```
RENAME OBJECT NOT SUBORDINATE TO PRECEDING RECORD
```

Type. Error

Cause. A data item referenced in the RENAMES clause must be defined within the preceding data description.

157

```
RENAME OBJECT IN TABLE OR HAS VARIABLE SIZE
```

Type. Error

Cause. The RENAMES clause cannot reference a table item; in addition, it cannot reference a group data item which contains a table whose occurence is variable.

158

```
ILLEGAL RENAMES OBJECT RANGE
```

Type. Error

Cause. The second data item in the range of a RENAMES clause must include some character positions that are not part of the first data item. However, the initial character position of the second data item cannot precede the initial character position of the first data item within their data structure.

159

```
REDEFINITION OBJECT NOT FOUND
```

Type. Error

Cause. Either the name in the REDEFINES clause cannot be found or it is not the name of a data item. Note that when a REDEFINES clause appears in a data structure, only that data structure is searched for the data item to be redefined.

160

```
REDEFINITION OBJECT HAS CONFLICTING LEVEL NUMBER
```

Type. Error

Cause. The data item to be redefined must have the same level number as the redefining data description entry.

161

```
REDEFINITION OBJECT IS REDEFINITION
```

Type. Error

Cause. A data item described with a REDEFINES clause cannot itself be redefined. This restriction does not apply to a subordinate of a redefinition item unless its data description entry also contains a REDEFINES clause.

162

```
REDEFINITION OBJECT AND REDEFINITION NOT SUBORDINATE TO SAME
LEVELS
```

Type. Error

Cause. When the redefined data item is subordinate to a set of group items, the redefinition item must also be subordinate to them.

163

```
REDEFINITION OBJECT NOT PRECEDING ITEM AT THIS LEVEL
```

Type. Error

Cause. The data description entry of a redefinition must not be separated from that of the redefined item by any other data description entry with the same level number, unless the intervening entry redefines the same data item.

164

```
REDEFINITION OBJECT IS TABLE OR HAS VARIABLE SIZE
```

Type. Error

Cause. A table item or a group item that has a variable size (that is, a subordinate variable occurrence table) cannot be redefined.

165

```
MISSING VALUE CLAUSE
```

Type. Error

Cause. The required VALUE clause is missing from a data description entry with level number 88.

166

```
MISSING RENAMES CLAUSE
```

Type. Error

Cause. The required RENAMES clause is missing from a data description entry with level number 66.

167

```
GROUP ITEM HAS ELEMENTARY ITEM CLAUSE
```

Type. Error

Cause. The data description entry of a group item has a BLANK WHEN ZERO, JUSTIFIED, SYNCHRONIZED, or PICTURE clause. These clauses can only describe an elementary data item. BLANK WHEN ZERO syntax is enforced when used, but data items using this syntax cannot be accessed by SCREEN COBOL programs.

168

```
GROUP WITH SIGN CLAUSE HAS NO SIGNED NUMERIC SUBORDINATE
```

Type. Warning

Cause. The SCREEN COBOL language required a group data item described with a SIGN clause to have at least one signed numeric subordinate data item. SCREEN COBOL reports nonconformance for informational purposes only.

169

```
ELEMENTARY ITEM HAS NO PICTURE
```

Type. Error

Cause. An elementary data item must be described with a PICTURE clause.

174

```
FIRST ELEMENTARY ITEM NOT DISPLAY AND NOT ALIGNED
```

Type. Error

Cause. The indicated data item cannot be aligned to the first character position of the area it redefines. SCREEN COBOL does not permit a redefinition that requires allocation of implicit FILLER character positions to align the first elementary item.

175

```
REDEFINITION HAS INCORRECT SIZE
```

Type. Error

Cause. The number of character positions occupied by a redefinition must equal the number of character positions occupied by the redefined data item(s), unless the redefinition begins at the 01 level.

176

```
NONNUMERIC FIGURATIVE OR LITERAL NOT PERMITTED FOR NUMERIC
ITEM
```

Type. Error

Cause. The initial value for a numeric data item must be a numeric literal or one of the figurative constants ZERO, ZEROS, or ZEROES.

177

```
SIGNED LITERAL NOT PERMITTED FOR UNSIGNED NUMERIC ITEM
```

Type. Error

Cause. The initial value for an unsigned numeric data item must be an unsigned numeric literal or one of the figurative constants ZERO, ZEROS, or ZEROES.

178

```
TOO MANY FRACTION DIGITS IN NUMERIC LITERAL
```

Type. Error

Cause. Assignment of the initial value to the numeric data item would require truncation of nonzero digits to the right of the decimal point.

179

```
NUMERIC LITERAL VALUE TOO LARGE FOR ITEM
```

Type. Error

Cause. Assignment of the initial value to the numeric data item would require truncation of nonzero digits to the left of the decimal point.

180

```
NUMERIC LITERAL NOT PERMITTED FOR NONNUMERIC OR GROUP ITEM
```

Type. Error

Cause. A numeric literal can only be used as the initial value for an elementary numeric data item.

181

```
NONNUMERIC LITERAL EXCEEDS ITEM SIZE
```

Type. Error

Cause. Assignment of the initial value to the indicated data item would require truncation of one or more characters.

182

```
01 OR 77 LEVEL ITEM TOO LARGE
```

Type. Error

Cause. SCREEN COBOL supports a maximum of 32,000 character positions for a level 01 or level 77 data item defined in the Working-Storage Section or Linkage Section.

183

```
DATA ITEM DESCRIPTION NOT COMPATIBLE WITH FILE CODE SET
```

Type. Error

Cause. A file with CODE SET may not have a data item with this description.

190

```
DEPENDING ITEM NOT FOUND
```

Type. Error

Cause. A name referenced in the DEPENDING phrase of an OCCURS clause is not defined.

191

```
DEPENDING ITEM NOT SIMPLE UNSIGNED INTEGER DATA ITEM
```

Type. Error

Cause. Either the indicated name (referenced in the DEPENDING phrase of an OCCURS clause) does not identify an elementary unsigned integer data item, or access to the item requires subscripting.

192

```
DEPENDING ITEM IN TABLE
```

Type. Error

Cause. The indicated data item is allocated within the table it controls. The allocation is a result of an explicit or implicit redefinition.

203

```
MESSAGE FORMAT MUST BE THE SAME IN ALL REPLIES
```

Type. Error

Cause. All reply messages in any one SEND MESSAGE statement must have the same format (either FIXED, VARYING1, or VARYING2). A reply from working storage is FIXED.

205

```
USING OPERAND NOT FOUND IN LINKAGE SECTION
```

Type. Error

Cause. A name in the USING phrase of the Procedure Division header is not defined in the Linkage Section of the Data Division.

206

```
USING OPERAND NOT DATA ITEM
```

Type. Error

Cause. The indicated name does not identify a data item. Only data item names can be specified in the USING phrase of the Procedure Division header.

207

```
USING OPERAND IS REDEFINITION OR NOT LEVEL 01 OR LEVEL 77
DATA ITEM
```

Type. Error

Cause. SCREEN COBOL requires that a data item in the USING phrase be a level 01 or level 77 item. SCREEN COBOL does not permit a redefinition, including one of a level 01 or level 77 item, to appear in the USING phrase.

208

```
DATA ITEM PERMITTED ONLY ONCE AS USING OPERAND
```

Type. Error

Cause. The same name cannot appear more than once in the USING phrase of the Procedure Division header.

209

```
TOO MANY USING OPERANDS
```

Type. Error

Cause. SCREEN COBOL supports a maximum of 29 names in the USING phrase of the Procedure Division header.

210

```
LINKAGE DATA ITEM MUST BE USING OPERAND
```

Type. Error

Cause. The indicated data item is defined in the Linkage Section but cannot be addressed. Addressable items are those specified in the USING phrase of the Procedure Division header; their subordinate items; and the redefinition, renaming, and condition-names of the subordinate items.

211

```
TOO MANY RECORDS
```

Type. Error

Cause. The program defines more data structures than the compiler can address.

212

```
TOO MANY ELEMENTARY ITEMS
```

Type. Error

Cause. The program defines more level 77 data items than the SCREEN COBOL compiler can address.

213

```
FEATURE NOT SUPPORTED BY SCREEN COBOL
```

Type. Error

Cause. There are many features not supported by SCREEN COBOL, of which this is one.

214

```
MULTIPLE UNIT COMPILES NOT SUPPORTED BY SCREEN COBOL
```

Type. Error

Cause. Multiple IDENTIFICATION divisions encountered during compilation.

221

```
INSUFFICIENT SPECIFICATION TO DETERMINE TYPE OF SCREEN ITEM
```

Type. Error

Cause. Any screen item (other than the 01 level screen name) must be either a group, an overlay area, a literal field, an input field, an output field, or an input-output field. This item cannot be classified because it does not have the minimum requirements for definition.

222

```
THIS SCREEN ITEM MUST BE NAMED
```

Type. Error

Cause. A name is required for 01 levels (screen names) and overlay areas.

223

```
THIS SCREEN ITEM MUST HAVE BASE SPECIFICATION
```

Type. Error

Cause. The screen item must have a BASE clause specified.

224

```
THIS SCREEN ITEM MUST HAVE 'OVERLAY' SPECIFICATION
```

Type. Error

Cause. The screen item must have an OVERLAY clause specified.

225

```
THIS SCREEN ITEM MUST HAVE SIZE SPECIFIED
```

Type. Error

Cause. Overlay areas must have a SIZE clause.

226

```
THIS SCREEN ITEM MUST HAVE 'AREA' SPECIFICATION
```

Type. Error

Cause. Overlay areas must have an AREA clause.

227

```
THIS SCREEN ITEM MUST HAVE LOCATION ('AT') SPECIFIED
```

Type. Error

Cause. All screen items must have allocation specified. Screen fields can use either the AT clause or the REDEFINES clause or both.

228

```
THIS SCREEN ITEM MUST HAVE LOCATION ('REDEFINES') SPECIFIED
```

Type. Error

Cause. All screen items must have a location specified. Screen fields can use either the AT clause or the REDEFINES clause or both.

229

```
THIS SCREEN ITEM MUST HAVE FROM (OR USING) DATA ITEM
```

Type. Error

Cause. The screen item must have a FROM or USING clause specified with an associated data item.

230

```
THIS SCREEN ITEM MUST HAVE TO (OR USING) DATA ITEM
```

Type. Error

Cause. The screen item must have a TO or USING clause specified with an associated data item.

231

```
THIS SCREEN ITEM MUST HAVE SHADOW DATA ITEM SPECIFIED
```

Type. Error

Cause. The screen item must have a SHADOWED clause specified with an associated data item.

232

```
THIS SCREEN ITEM MUST HAVE PICTURE SPECIFICATION
```

Type. Error

Cause. Input fields, output fields, and input-output fields must have a PICTURE clause.

233

```
THIS SCREEN ITEM MUST HAVE INITIAL VALUE
```

Type. Error

Cause. Literal fields must have an initial value.

234

```
THIS SCREEN ITEM MUST HAVE FILL CHARACTER SPECIFIED
```

Type. Error

Cause. The screen item must have a FILL clause specified with a fill character.

235

```
THIS SCREEN ITEM MUST HAVE OCCURS SPECIFICATION
```

Type. Error

Cause. The screen item must have an OCCURS clause specified.

236

```
THIS SCREEN ITEM MUST HAVE ACCEPTABLE VALUE(S) ('MUST')
SPECIFIED
```

Type. Error

Cause. The MUST BE clause for the screen item must specify a value that is compatible with the screen PICTURE clause.

237

```
THIS SCREEN ITEM MUST HAVE ACCEPTABLE LENGTH(S) SPECIFIED
```

Type. Error

Cause. The LENGTH clause for the screen item must specify a length that is compatible with the screen PICTURE clause.

238

```
THIS SCREEN ITEM MUST HAVE UPSHIFT SPECIFICATION
```

Type. Error

Cause. The screen item must have an UPSHIFT clause specified with a valid input or output specification.

239

```
THIS SCREEN ITEM MUST HAVE FULL ACTION ('WHEN FULL')
SPECIFIED
```

Type. Error

Cause. The screen item must have a WHEN FULL clause specified.

240

```
THIS SCREEN ITEM MUST HAVE USER CONVERSION NUMBER SPECIFIED
```

Type. Error

Cause. The screen item must have a USER CONVERSION clause specified.

243

```
THIS SCREEN ITEM MUST HAVE PROMPT FIELD SPECIFIED
```

Type. Error

Cause. The screen item must have a PROMPT clause specified in the Screen Section.

254

```
THIS SCREEN ITEM MUST NOT BE NAMED
```

Type. Error

Cause. Literal fields must not be named.

255

```
THIS SCREEN ITEM MUST NOT HAVE BASE SPECIFICATION
```

Type. Error

Cause. The BASE clause is allowed only at the 01 level.

256

```
THIS SCREEN ITEM MUST NO HAVE 'OVERLAY' SPECIFICATION
```

Type. Error

Cause. The OVERLAY clause is allowed only at the 01 level.

257

```
THIS SCREEN ITEM MUST NOT HAVE SIZE SPECIFIED
```

Type. Error

Cause. The SIZE clause is allowed only at the 01 level or for overlay area items.

258

```
THIS SCREEN ITEM MUST NOT HAVE 'AREA' SPECIFICATION
```

Type. Error

Cause. AREA can be specified only for overlay areas. This item either has conflicting clauses or subordinate items (is a group).

259

```
THIS SCREEN ITEM MUST NOT HAVE LOCATION ('AT') SPECIFIED
```

Type. Error

Cause. The screen item must not have an associated screen location. The AT clause is allowed only for screen groups and fields.

260

```
THIS SCREEN ITEM MUST NOT HAVE LOCATION ('REDEFINES')
SPECIFIED
```

Type. Error

Cause. The screen item must not redefine another screen item. The REDEFINES clause is allowed only for elementary screen fields.

261

```
THIS SCREEN ITEM MUST NOT HAVE FROM (OR USING) DATA ITEM
```

Type. Error

Cause. Only output fields and input-output fields can have FROM or USING clauses.

262

```
THIS SCREEN ITEM MUST NOT HAVE TO (OR USING) DATA ITEM
```

Type. Error

Cause. Only input fields and input-output fields can have TO or USING clauses.

263

```
THIS SCREEN ITEM MUST NOT HAVE SHADOW DATA ITEM SPECIFIED
```

Type. Error

Cause. SHADOWED clauses are allowed only for input, output, or input-output fields.

264

```
THIS SCREEN ITEM MUST NOT HAVE PICTURE SPECIFICATION
```

Type. Error

Cause. PICTURE clauses are allowed only for input, output, or input-output fields.

265

```
THIS SCREEN ITEM MUST NOT HAVE INITIAL VALUE
```

Type. Error

Cause. VALUE clauses are allowed only for input, output, input-output, or literal fields.

266

```
THIS SCREEN ITEM MUST NOT HAVE FILL CHARACTER SPECIFIED
```

Type. Error

Cause. FILL clauses are allowed only for input, output, or input-output fields.

267

```
THIS SCREEN ITEM MUST NOT HAVE OCCURS SPECIFICATION
```

Type. Error

Cause. OCCURS clauses are allowed only for input, output, or input-output fields.

268

```
THIS SCREEN ITEM MUST NOT HAVE ACCEPTABLE VALUE(S) ('MUST')
SPECIFIED
```

Type. Error

Cause. MUST clauses are allowed only for input or input-output fields.

269

```
THIS SCREEN ITEM MUST NOT HAVE ACCEPTABLE LENGTH(S) SPECIFIED
```

Type. Error

Cause. LENGTH clauses are allowed only for input or input-output fields.

270

```
THIS SCREEN ITEM MUST NOT HAVE UPSHIFT SPECIFICATION
```

Type. Error

Cause. UPSHIFT clauses are allowed only for input, output, or input-output fields.

271

```
THIS SCREEN ITEM MUST NOT HAVE FULL ACTION ('WHEN FULL')
SPECIFIED
```

Type. Error

Cause. WHEN FULL clauses are allowed only for input or input-output fields.

272

```
THIS SCREEN ITEM MUST NOT HAVE USER CONVERSION NUMBER
SPECIFIED
```

Type. Error

Cause. USER CONVERSION clauses are allowed only for input, output, or input-output fields.

275

```
THIS SCREEN ITEM MUST NOT HAVE PROMPT FIELD SPECIFIED
```

Type. Error

Cause. The screen item must not have a PROMPT clause specified.

276

```
THIS SCREEN ITEM MUST NOT HAVE FIELD-SEPARATOR CHARACTER
SPECIFIED
```

Type. Error

Cause. The screen item must not have a FIELD-SEPARATOR clause specified. This clause can be specified only for an 01 screen level item.

277

```
THIS SCREEN ITEM MUST NOT HAVE GROUP-SEPARATOR CHARACTER
SPECIFIED
```

Type. Error

Cause. The screen item must not have a GROUP-SEPARATOR clause specified. This clause can be specified only for an 01 screen level item.

278

```
THIS SCREEN ITEM MUST NOT HAVE ABORT-INPUT CHARACTERS
SPECIFIED
```

Type. Error

Cause. The screen item must not have an ABORT-INPUT clause specified. This clause can be specified only for an 01 screen level item.

279

```
THIS SCREEN ITEM MUST NOT HAVE END-OF-INPUT CHARACTERS
SPECIFIED
```

Type. Error

Cause. The screen item must not have an END-OF-INPUT clause specified. This clause can be specified only for an 01 screen level item.

280

```
THIS SCREEN ITEM MUST NOT HAVE RESTART-INPUT CHARACTERS
SPECIFIED
```

Type. Error

Cause. The screen item must not have a RESTART-INPUT clause specified. This clause can be specified only for an 01 screen level item.

281

```
THIS SCREEN ITEM MUST NOT HAVE FIXED-LENGTH SPECIFICATION
```

Type. Error

Cause. A fixed-length specification is not allowed for this item.

282

```
THIS SCREEN ITEM MUST NOT HAVE TRANSPARENT SPECIFICATION
```

Type. Error

Cause. A transparent specification is not allowed for this item.

285

```
THIS SCREEN ITEM MUST NOT HAVE ADVISORY SPECIFICATION
```

Type. Error

Cause. ADVISORY clauses are allowed only for output, or input-output fields.

286

```
THIS SCREEN ITEM HAS DUPLICATE NAMING
```

Type. Error

Cause. Duplicate clauses are not allowed.

287

```
THIS SCREEN ITEM HAS DUPLICATE BASE SPECIFICATION
```

Type. Error

Cause. Duplicate clauses are not allowed.

288

```
THIS SCREEN ITEM HAS DUPLICATE 'OVERLAY' SPECIFICATION
```

Type. Error

Cause. Duplicate clauses are not allowed.

289

```
THIS SCREEN ITEM HAS DUPLICATE SIZE SPECIFIED
```

Type. Error

Cause. Duplicate clauses are not allowed.

290

```
THIS SCREEN ITEM HAS DUPLICATE 'AREA' SPECIFICATION
```

Type. Error

Cause. Duplicate clauses are not allowed.

291

```
THIS SCREEN ITEM HAS DUPLICATE LOCATION ('AT') SPECIFIED
```

Type. Error

Cause. Duplicate clauses are not allowed.

292

```
THIS SCREEN ITEM HAS DUPLICATE LOCATION ('REDEFINES')
SPECIFIED
```

Type. Error

Cause. Duplicate clauses are not allowed.

293

```
THIS SCREEN ITEM HAS DUPLICATE FROM (OR USING) DATA ITEM
```

Type. Error

Cause. Duplicate clauses are not allowed.

294

```
THIS SCREEN ITEM HAS DUPLICATE TO (OR USING) DATA ITEM
```

Type. Error

Cause. Duplicate clauses are not allowed.

295

```
THIS SCREEN ITEM HAS DUPLICATE SHADOW DATA ITEM SPECIFIED
```

Type. Error

Cause. Duplicate clauses are not allowed.

296

```
THIS SCREEN ITEM HAS DUPLICATE PICTURE SPECIFICATION
```

Type. Error

Cause. Duplicate clauses are not allowed.

297

```
THIS SCREEN ITEM HAS DUPLICATE INITIAL VALUE
```

Type. Error

Cause. Duplicate clauses are not allowed.

298

```
THIS SCREEN ITEM HAS DUPLICATE FILL CHARACTER SPECIFIED
```

Type. Error

Cause. Duplicate clauses are not allowed.

299

```
THIS SCREEN ITEM HAS DUPLICATE OCCURS SPECIFICATION
```

Type. Error

Cause. Duplicate clauses are not allowed.

300

```
THIS SCREEN ITEM HAS DUPLICATE ACCEPTABLE VALUE(S) ('MUST')
SPECIFIED
```

Type. Error

Cause. Duplicate clauses are not allowed.

301

```
THIS SCREEN ITEM HAS DUPLICATE ACCEPTABLE LENGTH(S) SPECIFIED
```

Type. Error

Cause. Duplicate clauses are not allowed.

302

```
THIS SCREEN ITEM HAS DUPLICATE UPSHIFT SPECIFICATION
```

Type. Error

Cause. Duplicate clauses are not allowed.

303

```
THIS SCREEN ITEM HAS DUPLICATE FULL ACTION ('WHEN FULL')
SPECIFIED
```

Type. Error

Cause. Duplicate clauses are not allowed.

304

```
THIS SCREEN ITEM HAS DUPLICATE USER CONVERSION NUMBER
SPECIFIED
```

Type. Error

Cause. Duplicate clauses are not allowed.

307

```
THIS SCREEN ITEM HAS DUPLICATE PROMPT FIELD SPECIFIED
```

Type. Error

Cause. Duplicate clauses are not allowed.

308

```
THIS SCREEN ITEM HAS DUPLICATE FIELD-SEPARATOR CHARACTER
SPECIFIED
```

Type. Error

Cause. Duplicate characters are not allowed in multiple input character clauses.

309

```
THIS SCREEN ITEM HAS DUPLICATE GROUP-SEPARATOR CHARACTERS
SPECIFIED
```

Type. Error

Cause. Duplicate characters are not allowed in multiple input character clauses.

310

```
THIS SCREEN ITEM HAS DUPLICATE ABORT-INPUT CHARACTERS
SPECIFIED
```

Type. Error

Cause. Duplicate characters are not allowed in multiple input character clauses.

311

```
THIS SCREEN ITEM HAS DUPLICATE END-OF-INPUT CHARACTERS
SPECIFIED
```

Type. Error

Cause. Duplicate characters are not allowed in multiple input character clauses.

312

```
THIS SCREEN ITEM HAS DUPLICATE RESTART-INPUT CHARACTERS
SPECIFIED
```

Type. Error

Cause. Duplicate characters are not allowed in multiple input character clauses.

313

```
THIS SCREEN ITEM HAS DUPLICATE FIXED-LENGTH SPECIFICATION
```

Type. Error

Cause. Duplicate fixed-length specification is not allowed.

314

```
THIS SCREEN ITEM HAS DUPLICATE TRANSPARENT SPECIFICATION
```

Type. Error

Cause. Duplicate transparent specification is not allowed.

317

```
THIS SCREEN ITEM HAS DUPLICATE ADVISORY SPECIFICATION
```

Type. Error

Cause. Duplicate clauses are not allowed.

318

```
INPUT SCREEN ITEMS (TO OR USING) MAY NOT BE PROTECTED
```

Type. Error

Cause. Input and input-output fields must not be protected; if they were, data entry would be impossible.

319

```
REDEFINED SCREEN ITEM HAS DIFFERENT LOCATION
```

Type. Error

Cause. A redefined field must have the same location as the field it redefines.

320

```
REDEFINED SCREEN ITEM HAS DIFFERENT LENGTH
```

Type. Error

Cause. A redefined field must have the same length as the field it redefines.

321

```
REDEFINED SCREEN ITEM HAS DIFFERENT DISPLAY ATTRIBUTE
```

Type. Error

Cause. A redefined field must have the same display attribute as the field it redefines.

322

```
REDEFINED SCREEN ITEM HAS DIFFERENT FULL ACTION
```

Type. Error

Cause. A redefined field must have the same full action (WHEN FULL) as the field it redefines.

323

```
REDEFINED SCREEN ITEM HAS DIFFERENT OCCURS SPECIFICATION
```

Type. Error

Cause. A redefined field must have the same occurs specification as the field it redefines.

324

```
DUPLICATE SPECIFICATION FOR DISPLAY ATTRIBUTE
```

Type. Error

Cause. A given type of display attribute has been declared more than once; the attribute can only be declared once.

325

```
INITIAL VALUE MUST BE QUOTED STRING
```

Type. Error

Cause. Only string literals are allowed for initial values of screen items.

326

```
TOO MANY SEPARATORS OR OFFSETS IN COLUMN SPACING LIST
```

Type. Error

Cause. The column spacing list must contain fewer entries than there are column occurrences.

327

```
UNKNOWN TERMINAL TYPE
```

Type. Error

Cause. Terminal type must be IBM-3270, T16-6510, T16-6520, T16-6530, or T16-6540.

328

```
NO TERMINAL TYPE SPECIFIED
```

Type. Error

Cause. A terminal type clause is required.

329

```
FUNCTION KEY NOT ALLOWED FOR THIS TERMINAL TYPE
```

Type. Error

Cause. The function key mentioned is not available for this terminal type.

330

```
DISPLAY ATTRIBUTE NOT ALLOWED FOR THIS TERMINAL TYPE
```

Type. Error

Cause. The display attribute mentioned is not available for this terminal type.

331

```
FROM (USING) DATA ITEM HAS DIFFERENT TYPE (NUMBER VS STRING)
```

Type. Error

Cause. Numeric screen items must be associated with numeric data items; nonnumeric screen items must be associated with nonnumeric data items.

332

```
FROM (USING) DATA ITEM HAS INSUFFICIENT NUMBER OF OCCURRENCES
```

Type. Error

Cause. The screen item has more occurrences than the data item.

333

```
FROM (USING) DATA ITEM HAS INCOMPATIBLE SCALE
```

Type. Error

Cause. The scale specified for the TO or USING data item is not compatible with that specified by the screen item PICTURE. The scale should be adjusted for compatible editing of data.

334

```
TO (USING) DATA ITEM HAS DIFFERENT TYPE (NUMBER VS. STRING)
```

Type. Error

Cause. Numeric screen items must be associated with numeric data items; nonnumeric screen items must be associated with nonnumeric data items.

335

```
TO (USING) DATA ITEM HAS INSUFFICIENT NUMBER OF OCCURRENCES
```

Type. Error

Cause. The screen item has more occurrences than the data item.

336

```
TO (USING) DATA ITEM HAS INCOMPATIBLE SCALE
```

Type. Error

Cause. The scale specified for the FROM or USING data item is not compatible with that specified by the screen item PICTURE. The scale should be adjusted for compatible editing of data.

337

```
VALUE STRING LONGER THAN PICTURE
```

Type. Error

Cause. The value string must not be longer than the screen item.

338

```
OVERLAY AREA TOO LARGE
```

Type. Error

Cause. The overlay area is larger than the base screen.

339

```
OVERLAY SCREENS MAY NOT CONTAIN OVERLAY AREAS
```

Type. Error

Cause. An overlay screen cannot have an overlay area; only a base screen can contain an overlay area.

340

```
OVERLAY AREAS MUST BE FULL WIDTH FOR THIS TERMINAL TYPE
```

Type. Error

Cause. Overlay areas must be as wide as the base screen on T16-6510 terminals.

341

```
SCREEN TOO LARGE FOR TERMINAL TYPE
```

Type. Error

Cause. Screen size exceeds the largest supported size for this terminal type.

342

```
ERROR ENHANCEMENT MAY NOT SPECIFY PROTECTION ATTRIBUTE
```

Type. Error

Cause. An ERROR-ENHANCEMENT clause must not specify PROTECTED attribute. If a field in error is protected, correction of the error would not be possible.

343

```
SHADOWED DATA ITEM HAS INSUFFICIENT NUMBER OF OCCURRENCES
```

Type. Error

Cause. If a shadowed field contains an OCCURS clause, the shadowed data item must have the same number of occurrences as the field.

344

```
SCREEN ITEM TOO LONG
```

Type. Error

Cause. The maximum field length of 255 characters has been exceeded.

345

```
UNKNOWN CHARACTER SET TYPE
```

Type. Error

Cause. Invalid character-set type was specified in the OBJECT-COMPUTER paragraph of the Environment Division.

346

```
CHARACTER SET NOT VALID FOR THIS TERMINAL TYPE
```

Type. Warning

Cause. The character set specified is not valid for the terminal. The character set specification is ignored.

347

```
PROMPT SCREEN ITEM MUST BE A FIELD IN SAME SCREEN
```

Type. Error

Cause. The message applies to one of the following conditions:

- The screen item named in the PROMPT clause and the definition of the screen field must be in the same screen.
- The screen item must be a field.
- The screen item cannot be an overlay, a group, or a filler item.

348

```
PROMPT SCREEN ITEM MUST NOT BE THE CURRENT ITEM
```

Type. Error

Cause. The screen item named in the PROMPT clause cannot refer to the screen item containing the PROMPT clause. A screen field cannot be prompted by itself.

349

```
PROMPT SCREEN ITEM MUST HAVE A FROM (OR USING) DATA ITEM
```

Type. Error

Cause. If the screen item named in the PROMPT clause has an associated Working-Storage data item, the screen item must have a FROM or USING clause. A TO clause generates this error.

350

```
DUPLICATE INPUT EDIT CONTROL CHARACTERS DEFINED
```

Type. Error

Cause. The same character cannot be defined for more than one input-control character within each screen.

351

```
OPERAND MUST BE A SCREEN ITEM
```

Type. Error

Cause. The operand must be an item defined in the Screen Section.

352

```
OPERAND MUST BE A DATA ITEM
```

Type. Error

Cause. The operand must be an item defined in the Working-Storage Section.

353

```
OPERAND MUST BE A MNEMONIC-NAME
```

Type. Error

Cause. The operand must be a mnemonic name specified in the SPECIAL-NAMES paragraph.

354

```
INVALID CONVERSATIONAL SEPARATOR
```

Type. Error

Cause. A field or group separator is defined incorrectly. A nonnumeric literal must be one alphanumeric character enclosed in quotation marks. A numeric literal must be in the range 0 through 255.

355

```
TOO MANY COLUMN OCCURRENCES SPECIFIED FOR SCREEN FIELD
```

Type. Error

Cause. An OCCURS clause includes a column number greater than the number of columns in the size of the screen. For example, if a screen is defined as SIZE 20, 80, an OCCURS IN 82 COLUMNS generates this message.

356

```
TOO MANY LINE OCCURRENCES SPECIFIED FOR SCREEN FIELD
```

Type. Error

Cause. An OCCURS clause includes a line number greater than the number of lines in the size of the screen. For example, if a screen is defined as SIZE 20, 80, an OCCURS IN 24 LINES generates this message.

357

```
LENGTH CLAUSE NOT VALID FOR SCREEN FIELD PICTURE
```

Type. Error

Cause. The length of the data item specified in the LENGTH clause is too short or long for the length indicated by the PICTURE clause. Correct either the LENGTH or PICTURE clause so that they specify compatible lengths.

358

```
THIS MESSAGE ITEM MUST HAVE TO, FROM, OR USING DATA ITEM
```

Type. Error

Cause. A field in the Message Section must be described with a TO, FROM,or USING clause that names a data item in Working-Storage.

359

```
THIS MESSAGE ITEM MUST HAVE PICTURE SPECIFICATION
```

Type. Error

Cause. A field in the Message Section must be described with a PICTURE clause.

360

```
MESSAGE ITEM TOO LONG
```

Type. Error

Cause. A message field in the Message Section is defined with a size greater than 32,000 bytes.

361

```
THIS MESSAGE ITEM HAS DUPLICATE FROM (OR USING) DATA ITEM
```

Type. Error

Cause. Duplicate clauses are not allowed.

362

```
THIS MESSAGE HAS DUPLICATE TO (OR USING) DATA ITEM
```

Type. Error

Cause. Duplicate clauses are not allowed.

363

```
THIS MESSAGE ITEM HAS DUPLICATE PICTURE SPECIFICATION
```

Type. Error

Cause. Duplicate clauses are not allowed.

364

```
THIS MESSAGE ITEM HAS DUPLICATE USER CONVERSION NUMBER
SPECIFIED
```

Type. Error

Cause. Duplicate clauses are not allowed.

365

```
THIS MESSAGE ITEM HAS DUPLICATE FORMAT SPECIFICATION
```

Type. Error

Cause. Duplicate clauses are not allowed.

366

```
THIS MESSAGE ITEM HAS DUPLICATE NAMING
```

Type. Error

Cause. A field within the Message Section has more than one name clause specified; only one name clause is allowed.

367

```
MESSAGE ITEM HAS DUPLICATE OCCURS SPECIFICATION
```

Type. Error

Cause. Duplicate OCCURS specifications are not allowed.

368

```
THIS MESSAGE ITEM HAS DUPLICATE FIELD STATUS
```

Type. Error

Cause. A field within the Message Section has more than one FIELD STATUS clause specified; duplicate clauses are not allowed.

369

```
ITEM SPECIFIED BY FIELD STATUS IS TOO SHORT
```

Type. Error

Cause. A Message Section item referenced a Working-Storage Section item that was not large enough to contain the field status information. The recommended size and format for the Working-Storage item is two 9(4) COMP fields within one group item.

370

```
VARYING FORMAT NOT SUPPORTED FOR MULTI-FIELD MESSAGES
```

Type. Error

Cause. Only messages of FIXED, DELIMITED, or FIXED DELIMITED format can have group and field items within the 01 level.

371

```
OPERAND MUST BE AN ELEMENTARY MESSAGE ITEM
```

Type. Error

Cause. The operand must be a Message Section item and cannot be an 01 level item, a group item, or an OCCURS clause item.

372

```
RESULTING COUNT ALLOWED ON 01 LEVEL OR ELEMENTARY ITEMS ONLY
```

Type. Error

Cause. The RESULTING COUNT clause is not allowed as part of a group item specification.

373

```
PRESENT IF MUST REFERENCE ELEMENTARY ITEM IN CURRENT MESSAGE
```

Type. Error

Cause. The PRESENT IF clause must reference an elementary Message Section item in the same message. The PRESENT IF clause cannot reference an 01 level item, a group item, or an OCCURS clause item.

374

```
PRESENT IF MAY NOT REFERENCE GROUP ITEM
```

Type. Error

Cause. The PRESENT IF clause cannot reference a group item because its item reference must specify a PICTURE clause as part of its specification.

375

```
DELIMITER MUST BE QUOTED CHARACTER, OR NUMERIC IN 0-255 RANGE
```

Type. Error

Cause. The delimiting character defined by the FIELD-DELIMITER or MESSAGE-DELIMITER clause must be specified in quotation marks ("P"), or the decimal representation of the desired ASCII or EBCDIC character (80 = P, 3 = ETX).

376

```
ALLOWED ON 01 LEVEL ITEM ONLY
```

Type. Error

Cause. The MESSAGE FORMAT clause is allowed only on the 01 level item.

377

```
PRESENT IF NOT ALLOWED FOR THIS MESSAGE FORMAT
```

Type. Error

Cause. The PRESENT IF clause is allowed only on messages of DELIMITED format.

378

```
RESULTING COUNT MUST REFERENCE NUMERIC DATA ITEM
```

Type. Error

Cause. The Working-Storage field referred to by a RESULTING COUNT clause must be a numeric data item.

379

```
PIC 1 NOT ALLOWED WITH DELIMITED FIELDS
```

Type. Error

Cause. Messages of delimited format cannot contain items with binary data PICTURE clauses (PIC 1 format), unless delimiters are specified as OFF.

380

```
DEPENDING NOT ALLOWED IN FIXED LENGTH MESSAGE
```

Type. Error

Cause. A fixed length message may not have a DEPENDING clause.

381

```
PIC N NOT SUPPORTED IN MESSAGE
```

Type. Error

Cause. An attempt was made to define a data item with a PIC N in the Message Section.

382

```
PIC N NOT ALLOWED UNLESS 'CHARACTER-SET IS KANJI-KATAKANA'
```

Type. Error

Cause. An attempt was made to define a PIC N data item without specifying the clause CHARACTER-SET IS KANJI-KATAKANA in the Environment Division.

383

```
DATA/SCREEN ITEM VALUE (PIC N) IS NOT A VALID DBCS
```

Type. Error

Cause. Data items or screen items defined as PIC N and having the VALUE attribute must use a valid double-byte character.

384

```
SCREEN ITEM DECLARED AS ADVISORY CANNOT REFERENCE PIC N DATA
ITEM
```

Type. Error

Cause. A screen item defined as an advisory field cannot reference a data item that is declared as a PIC N.

385

```
SCREEN ITEM FILL CHARACTER (PIC N) IS NOT A VALID DBCS
```

Type. Error

Cause. A screen item defined as PIC N and that uses the FILL attribute must use a valid double-byte character as the fill character.

386

```
SCREEN ITEM DECLARED WITH ILLEGAL FIELD ATTRIBUTE FOR PIC N
FIELD
```

Type. Error

Cause. An attempt was made to use a screen field attribute that is not supported for double-byte data items.

387

```
SCREEN ITEM 'FILL' CHARACTER CAN NOT BE DBCS (PIC N ONLY)
```

Type. Error

Cause. A double-byte character is being used for an item which is not PIC N.

388

```
SCREEN ITEMS (PIC N) MUST DECLARE PIC BEFORE OTHER ATTRIBUTES
```

Type. Error

Cause. PIC must come before attributes.

389

```
REDEFINE IS NOT SUPPORTED FOR PIC N SCREEN ITEMS
```

Type. Error

Cause. An attempt was made to use a REDEFINES clause for a PIC N screen data item.

390

```
SPECIFIED ATTRIBUTE COMBINATION IS ILLEGAL FOR THIS DEVICE
TYPE
```

Type. Error

Cause. Outline display attributes are available only on terminals in the IBM 3270 family.

391

```
PREVIOUS FOREGROUND OR BACKGROUND COLOR SETTING WILL BE RESET
```

Type. Warning

Cause. A foreground or background color will be reset.

392

```
FOREGROUND AND BACKGROUND ARE THE SAME COLOR
```

Type. Warning

Cause. The foreground and background colors are the same.

393

```
MNEMONIC NAME USED IN SCREEN SECTION MUST BE TYPE VIDEO
ATTRIBUTE
```

Type. Error

Cause. Mnemonic names must define supported extended field attributes.

394

```
OUTLINE ATTRIBUTES ARE ONLY ALLOWED FOR ELEMENTARY SCREEN
ITEMS
```

Type. Error

Cause. Outline display attributes can be in elementary screen items (not group items).

395

```
ILLEGAL ATTRIBUTE FOR OVERLAY AREA DEFINITION
```

Type. Error

Cause. Video or outline display attributes cannot be used for an overlay area in the Screen Section.

396

```
PIC 1 NOT ALLOWED WITH VARYING1/2 MESSAGE FORMAT
```

Type. Error

Cause. VARYING1 and VARYING2 may not be used with PIC 1.

397

```
OVERLAY SCREEN MUST HAVE SUBORDINATE ENTRIES (CAN'T BE EMPTY)
```

Type. Error

Cause. An overlay screen is not a group item.

398

```
SCREEN 'AREA' MUST NOT HAVE SUBORDINATE ENTRIES
```

Type. Error

Cause. "AREA" may not be a group item.

399

```
SCREEN SECTION MUST HAVE 01 LEVEL ENTRY
```

Type. Error

Cause. The SCREEN SECTION must begin with an 01 level item.

400

```
TOO MANY ERRORS
```

Type. Failure

Cause. The error limit has been exceeded.

402

```
UNABLE TO OPEN $RECEIVE
```

Type. Failure

Cause. Call to OPEN for $RECEIVE failed.

403

```
UNABLE TO OPEN COMMUNICATION FILE
```

Type. Failure

Cause. Unable to open SEND file.

404

```
UNABLE TO OPEN LIST FILE
```

Type. Failure

Cause. Call to OPEN for LIST file failed.

405

```
UNABLE TO USE LIST FILE
```

Type. Failure

Cause. The record length for the LIST file is less than 40 bytes.

406

```
UNABLE TO CREATE WORK FILE
```

Type. Failure

Cause. Attempt to create WORK file failed.

407

```
UNABLE TO OPEN WORK FILE
```

Type. Failure

Cause. Attempt to open WORK file failed.

410

```
COMPILER COMMUNICATION LOST
```

Type. Failure

Cause. Communication between the two compiler processes has been lost.

411

```
LIST FILE WRITE FAILURE
```

Type. Failure

Cause. Attempt to write to line failed.

414

```
UNABLE TO CREATE RUN UNIT FILE
```

Type. Failure

Cause. Attempt to create run unit file failed.

415

```
UNABLE TO OPEN RUN UNIT FILE
```

Type. Failure

Cause. Attempt to open run unit file failed.

416

```
FAILURE IN USING OR ALLOCATING EXTENDED SEGMENT
```

Type. Failure

Cause. Either a call to ALLOCATESEGMENT or a call to USESEGMENT has failed.

417

```
COMPILER LOGIC ERROR
```

Type. Failure

Cause. This error appears when a failure whose cause cannot be easily determined occurs.

421

```
CONTROL DATA SPACE OVERFLOW
```

Type. Failure

Cause. The compiler stack has exceeded 32K words.

423

```
FILE ERROR ON RUN UNIT FILE
```

Type. Failure

Cause. File operations on the run unit file repeated until a fatal file error occurred.

425

```
CODE FILE HAS EXCEEDED MAXIMUM POBJ CODE FILE SIZE
```

Type. Failure

Cause. The maximum size for a POBJ file has been exceeded.

431

```
NOT SUPPORTED
```

Type. Failure

Cause. Unsupported feature.

435

```
BLANK CONTINUATION LINE
```

Type. Warning

Cause. A continuation line is all blank.

436

```
ILLEGAL INDICATOR CHARACTER
```

Type. Warning

Cause. The indicator character is not valid on this line.

440

```
INCORRECT NUMBER OF PARAMETERS
```

Type. Error

Cause. Formal and actual parameter count do not match.

441

```
NAME CONFLICT
```

Type. Error

Cause. Two identifiers have the same name.

442

```
AMBIGUOUS REFERENCE
```

Type. Error

Cause. A reference is being made using a name which is satisfied by two or more objects.

444

```
SYNTAX ERROR DETECTED AT TOKEN
```

Type. Error

Cause. Syntax error with no extra action.

445

```
SYNTAX ERROR - REPLACING UNEXPECTED TOKEN BY <token>
```

Type. Error

Cause. Syntax error repaired by replacing current token.

446

```
SYNTAX ERROR - INSERTING MISSING TOKEN
```

Type. Error

Cause. Syntax error repaired by inserting missing token.

447

```
SYNTAX ERROR - DELETING UNEXPECTED TOKEN
```

Type. Error

Cause. Syntax error repaired by deleting token.

448

```
PARSING RESUMED AT TOKEN
```

Type. Error

Cause. Syntax error repaired by discarding tokens in buffer up to current token.

449

```
END-OF-FILE ENCOUNTERED DURING ERROR RECOVERY
```

Type. Error

Cause. Syntax error; end of file encountered trying to recover.

453

```
ILLEGAL SENDING OR RECEIVING ITEM IN MOVE STATEMENT
```

Type. Error

Cause. The message applies to one of the following conditions:

- A numeric data item cannot be moved into an alphabetic data item.
- A non-integer numeric data item cannot be moved into an alphanumeric data item.
- An alphabetic data item cannot be moved into a numeric data item.
- A double-byte data item cannot be moved into a single-byte data item.

454

```
READ-ONLY SPECIAL REGISTER;  MAY NOT BE ALTERED
```

Type. Error

Cause. The following special registers cannot be modified: PW-UNSOLICITED-MESSAGE QUEUED, PW-TCP-PROCESS-NAME, and PW-TCP-SYSTEM-NAME.

455

```
UNABLE TO OPEN SCREEN COBOL LIBRARY FILE
```

Type. Error

Cause. The specified SCREEN COBOL library cannot be accessed. The library either does not exist or could not be shared at compile time.

456

```
UNABLE TO LIST LOAD MAP
```

Type. Error

Cause. The object file cannot be opened to list the internal procedure load map.

457

```
UNDEFINED DATA NAME
```

Type. Error

Cause. The data item referred to is not described in the Environment Division or Data Division.

458

```
ONLY A FILE NAME IS ALLOWED IN THIS CONTEXT
```

Type. Error

Cause. A valid *Guardian* operating environment file name is expected here.

459

```
ONLY A MNEMONIC NAME IS ALLOWED IN THIS CONTEXT
```

Type. Error

Cause. A system name (mnemonic-name) is required in this context.

460

```
NO CORRESPONDING DATA NAMES
```

Type. Error

Cause. No correspondence was found between the specified groups.

461

```
UNDEFINED OR AMBIGUOUS PROCEDURE ACCESS
```

Type. Error

Cause. Either a procedure name referred to in a PERFORM or GO TO statement was not encountered in the source text, or the name was not sufficiently qualified to avoid ambiguity.

462

```
ILLEGAL ALTER CANDIDATE
```

Type. Error

Cause. A procedure marked as altered is not a valid alter candidate.

463

```
INDEPENDENT SEGMENTS NOT SUPPORTED
```

Type. Error

Cause. SCREEN COBOL does not support independent segments.

464

```
ILLEGAL DATA ITEM IN IF STATEMENT
```

Type. Error

Cause. An operand in a conditional statement is an illegally defined data item; the operand is defined as a numeric edited data item; or an attempt was made to use the IF ... DOUBLEBYTE statement to compare a PIC N data item or literal with a PIC 9 numeric data item.

465

```
ONLY A PROGRAM NAME IS ALLOWED IN THIS CONTEXT
```

Type. Error

Cause. A program name is expected here.

467

```
ONLY AN ALPHABET NAME IS ALLOWED IN THIS CONTEXT
```

Type. Error

Cause. The name specified must refer to an alphabetic name in this context.

468

```
EMPTY GO TO NOT LABELED
```

Type. Error

Cause. A GO TO statement of this form can only appear in a single statement paragraph, which by definition is labeled.

470

```
EXPECT ELEMENTARY NUMERIC DATA ITEM IN THIS CONTEXT
```

Type. Error

Cause. A numeric data item is required in this context.

471

```
EXPECT GROUP DATA ITEM IN THIS CONTEXT
```

Type. Error

Meaning. Only a group data item is legal in this context.

472

```
INVALID TABLE SUBSCRIPT OR INDEX
```

Type. Error

Cause. The item is not a data item; indexes are not allowed.

473

```
TOO MANY OR TOO FEW PARAMETERS
```

Type. Error

Cause. The number of parameters specified in the USING phrase of a CALL statement does not agree with the number specified in the USING phrase of the Procedure Division header.

474

```
EXPECT A DATA WORD OR IDENTIFIER IN THIS CONTEXT
```

Type. Error

Cause. Only a data item can be used in a class condition.

475

```
CATEGORY MUST BE ALPHANUMERIC OR ALPHABETIC
```

Type. Error

Cause. The category of the data item must be alphanumeric or alphabetic in this context.

476

```
CATEGORY MUST BE ALPHANUMERIC OR NUMERIC
```

Type. Error

Cause. The category of the data item must be alphanumeric or numeric in this context.

481

```
MISSING PROGRAM
```

Type. Error

Cause. A called program unit was neither found in a SCREEN COBOL program library nor encountered in the source text.

485

```
EXPECT A TABLE SPECIFIER
```

Type. Error

Cause. The description of the data item must contain an OCCURS clause.

486

```
INCORRECT NUMBER OF SUBSCRIPTS OR INDICES
```

Type. Error

Cause. An incorrect number of subscripts, possibly zero, are used to access the data item. (Indices are not allowed.)

487

```
REFERENCE DATA ITEM TOO LARGE
```

Type. Error

Cause. A data item has a PICTURE clause greater than 32,000 characters.

488

```
ONLY AN INDEX NAME IS ALLOWED IN THIS CONTEXT
```

Type. Error

Cause. Must be an index name to be used as an index.

489

```
AN UNEXPECTED ERROR OCCURRED WHILE BUILDING RUN UNIT
```

Type. Error

Cause. An irrecoverable I/O error occurred while building the object file. The compilation must be restarted.

490

```
EXPECT AN INDEX NAME OR INDEX DATA ITEM
```

Type. Error

Cause. An index name or index data item is expected.

491

```
ILLEGAL USE OF INDEX NAME OR INDEX DATA ITEM
```

Type. Error

Cause. An index name or index data item is not allowed.

495

```
INVALID VARYING ITEM
```

Type. Error

Cause. The VARYING identifier in the PERFORM statement must be described as a numeric elementary data item without any positions to the right of the assumed decimal point.

498

```
ILLEGAL COMPARISON BETWEEN DISPLAY AND COMPUTATIONAL DATA
```

Type. Error

Cause. A comparison between a numeric computational data item and a nonnumeric data item is illegal.

499

```
NON-INTEGER OR CONTAINS P'S
```

Type. Error

Cause. An integer data item containing no P symbols in its PICTURE clause is required in this context.

500

```
EXPECT ALPHANUMERIC DATA ITEM
```

Type. Error

Cause. The data item must have an explicit or implied alphanumeric category.

501

```
EXPECT DISPLAY USAGE
```

Type. Error

Cause. The data item must have explicit or implied DISPLAY usage.

503

```
EXPECT LEVEL 01 OR 77 DATA ITEM
```

Type. Error

Cause. Only level 01 and level 77 data items can be specified in the USING phrase of a CALL statement.

504

```
INVALID DISPLAY ITEM
```

Type. Error

Cause. An item is not defined in the Data Division.

507

```
INCONSISTENT OBJECT LIBRARY REFERENCES
```

Type. Error

Cause. A reference to a library is invalid because the library is not open or is otherwise incorrect.

510

```
FEATURE NOT YET IMPLEMENTED
```

Type. Error

Cause. The program attempted to make use of a feature that is not yet available.

511

```
EXPECT A RECORD NAME
```

Type. Error

Cause. A record name is expected here.

515

```
INVALID OPERATOR OR OPERAND IN ARITHMETIC EXPRESSION
```

Type. Error

Cause. The expression contains operators other than + - / * (plus, minus, slash, or asterisk) or contains one or more nonnumeric operands.

516

```
INVALID OPERATOR OR OPERAND IN CONDITIONAL EXPRESSION
```

Type. Error

Cause. The expression contains an illegal operator or illegal identifier form.

518

```
ILLEGAL TABLE OCCURRENCE NUMBER (BOUNDS VIOLATION)
```

Type. Error

Cause. An illegal occurrence number was detected.

523

```
ONLY INTEGER EXPONENTS ARE ALLOWED
```

Type. Error

Cause. A noninteger is being used as an exponent.

524

```
EXPONENT RANGE ERROR: ABS(EXPONENT) > 18
```

Type. Error

Cause. An exponent cannot be larger than 18.

525

```
RECORD MUST BE ASSOCIATED WITH SORT MERGE FILE
```

Type. Error

Cause. Record is not associated with sort merge file.

526

```
ONLY A SORT MERGE FILE IS ALLOWED
```

Type. Error

Cause. A sort merge file is expected here.

527

```
ONLY AN ELEMENTARY ALPHANUMERIC DATA ITEM IS ALLOWED
```

Type. Error

Cause. The data item referred to must be an elementary alphanumeric data item in this context.

528

```
ILLEGAL USING FILE IN SORT OR MERGE
```

Type. Error

Cause. The using file is invalid in this context.

529

```
ILLEGAL DATA ITEM IN ARITHMETIC STATEMENT
```

Type. Error

Cause. An operand in an arithmetic statement is an illegally defined data item; the operand is defined as a numeric edited data item.

530

```
DIVIDE BY ZERO
```

Type. Error

Cause. Division by a literal with a value of zero was detected in a DIVIDE or COMPUTE statement.

531

```
ONLY A NUMERIC LITERAL IS ALLOWED
```

Type. Error

Cause. Only numeric literals and data items can be used in the composition of an arithmetic expression.

540

```
VARIABLE LENGTH OR CONTAINS OCCURS CLAUSE
```

Type. Error

Cause. An OCCURS item cannot be used here.

544

```
INPUT AND OUTPUT PROCEDURES MUST BE SECTIONS
```

Type. Error

Cause. Input and output procedures must be sections.

545

```
ILLEGAL PERFORM INVOCATION
```

Type. Error

Cause. A statement generates an illegal implied or explicit transfer of control between mutually exclusive sections. The statement is compiled as if the requested action were legal.

545

```
ILLEGAL PERFORM INVOCATION
```

Type. Warning

Cause. A statement generates an illegal implied or explicit transfer of control between mutually exclusive sections. The statement is compiled as if the requested action were legal. The message applies to one of the following conditions:

- A debug section performs a declaration procedure.
- A nondebug declarative section performs a nondebug declarative procedure.
- A nondeclarative section performs a nondebug declarative procedure.

546

```
ILLEGAL GO TO INVOCATION
```

Type. Error

Cause. A statement generates an illegal implied or explicit transfer of control between mutually exclusive sections. The statement is compiled as if the requested action were legal. The message applies to one of the following conditions:

- A GO TO transfers between a declarative section and a nondeclarative section.
- A GO TO transfers between a debug section and a nondebug section.

546

```
ILLEGAL GO TO INVOCATION
```

Type. Warning

Cause. A statement generates an illegal implied or explicit transfer of control between mutually exclusive sections. The statement is compiled as if the requested action were legal.

551

```
MISSING USE STATEMENT
```

Type. Error

Cause. A USE statement is expected here.

552

```
MULTIPLE USE DEBUGGING PROCEDURE ASSIGNMENT
```

Type. Error

Cause. There are conflicting debugging specifications.

553

```
ILLEGAL ITEM FOR NUMERIC TEST
```

Type. Error

Cause. Alphabetic or group item with operational-sign subordinates is not valid here.

554

```
THIS ITEM CAN NOT BE DEBUGGED
```

Type. Error

Cause. This item is invalid for debugging.

558

```
ALPHABETIC ITEM NOT ALLOWED
```

Type. Error

Cause. Destination in an ACCEPT statement cannot be alphabetic.

559

```
TOO MANY VARYING PHRASES
```

Type. Error

Cause. The nesting level of a VARYING phrase is nested too deeply.

560

```
RECEIVING ITEM INCONSISTENT WITH SENDING ITEM
```

Type. Error

Cause. The source and destination data items are incompatible in a MOVE statement.

561

```
ONLY AN INTEGER ALLOWED IN THIS CONTEXT
```

Type. Error

Cause. A non-integer is used as a literal or as a subscript.

562

```
ITEM IS BOTH SUBSCRIPTED AND INDEXED
```

Type. Error

Cause. Subscripting and indexing are mutually exclusive.

563

```
INTEGER NOT IN EXPECTED RANGE
```

Type. Error

Cause. This is a generic error for integer values exceeding different limits depending on the context.

564

```
EXPECT NUMERIC OR NUMERIC-EDITED DATA ITEM
```

Type. Error

Cause. A numeric item is expected here.

565

```
ACCESS TO DEBUG SPECIAL REGISTERS ALLOWED ONLY IN DEBUGGING
SECTIONS
```

Type. Error

Cause. Accessing debug special registers is not allowed while outside of debugging section.

567

```
DEBUGGING SECTIONS MUST BE FIRST
```

Type. Error

Cause. A debugging section must be first.

568

```
ADDRESSING RANGE EXCEEDED
```

Type. Error

Cause. The sizes of the Working-Storage variables in a SEND statement cannot exceed 32,000 bytes; this applies both to the list of variables used for sending and for storing the reply. The sum of the sizes in each list can be as large as 32,000 bytes, but no more.

569

```
SOURCE ITEM MAY NOT EXCEED 18 DIGITS
```

Type. Error

Cause. A MOVE statement attempts to store an alphanumeric literal exceeding 18 characters in a numeric item. Numeric data is limited to 18 digits.

573

```
PROTECTION ATTRIBUTE MAY NOT BE CHANGED
```

Type. Error

Cause. The PROTECTED attribute must not be changed for the T16-6510.

574

```
OVERLAY SCREEN LARGER THAN OVERLAY AREA
```

Type. Error

Cause. The overlay screen is larger than the overlay area.

575

```
ACCEPT TERMINATION MNEMONIC NAME MUST BE FUNCTION KEY
```

Type. Error

Cause. Display attributes must not be used to terminate an ACCEPT statement.

576

```
SERVER CLASS NAME MUST BE ALPHA OR ALPHANUMERIC
```

Type. Error

Cause. PATHCOM does not accept numeric server class names.

577

```
TURN MNEMONIC NAME MUST BE DISPLAY ATTRIBUTE
```

Type. Error

Cause. Function keys are not allowed in TURN statements.

578

```
MUST BE SCREEN NAME
```

Type. Error

Cause. Groups, overlay areas, and fields are not allowed in this context.

579

```
MUST BE BASE SCREEN NAME
```

Type. Error

Cause. Overlay screens must not be used in this context.

580

```
MUST BE OVERLAY SCREEN NAME
```

Type. Error

Cause. Base screens must not be used in this context.

581

```
MUST BE SCREEN ITEM
```

Type. Error

Cause. Data items are not allowed in this context.

582

```
SCREEN ITEM TOO CLOSE TO START OF SCREEN
```

Type. Error

Cause. The space preceding the first screen item is insufficient. For detailed information on the rules for declaring screen fields, refer to the *Compaq NonStop™ Pathway/iTS TCP and Terminal Programming Guide*.

583

```
SCREEN ITEM TOO CLOSE TO END OF SCREEN
```

Type. Error

Cause. The space following the last screen item is insufficient.

584

```
SCREEN ITEM SPANS NOT-FULLWIDTH OVERLAY AREA LINE
```

Type. Error

Cause. A screen item cannot span an overlay screen line when that overlay screen is narrower than the base screen.

585

```
SCREEN ITEM OVERLAPS OR IS TOO CLOSE TO NEXT ITEM
```

Type. Error

Cause. Required separation between screen elements is not present.

586

```
SCREEN ITEM OVERLAPS OR IS TOO CLOSE TO PREVIOUS ITE
```

Type. Error

Cause. Required separation between screen elements is not present.

587

```
SCREEN ITEM WITHOUT OCCURS CLAUSE IS SUBSCRIPTED
```

Type. Error

Cause. Only a screen item with an OCCURS clause can be subscripted.

588

```
SCREEN ITEMS MAY HAVE ONE (1) SUBSCRIPT ONLY
```

Type. Error

Cause. A screen item with an OCCURS clause is considered to be a single table.

589

```
SCREEN ITEM SUBSCRIPT TOO LARGE
```

Type. Error

Cause. The subscript exceeds the count of screen items.

590

```
SCREEN ITEM SUBSCRIPT MUST BE INTEGER
```

Type. Error

Cause. All subscripts must be integers.

591

```
MUST BE OVERLAY AREA SCREEN NAME
```

Type. Error

Cause. The overlay area following the AT clause in a DISPLAY OVERLAY statement is not defined as an overlay area.

592

```
'LENGTH MUST BE' VALUE MUST BE LESS THAN 256
```

Type. Error

Cause. The maximum value that can be specified in a LENGTH MUST BE clause is 255.

593

```
INCORRECT NUMBER OF OPERANDS IN ARITHMETIC EXPRESSION
```

Type. Error

Cause. An arithmetic expression contains either too many or too few operands.

594

```
SCALE OF MUST BE VALUE EXCEEDS SCALE OF ASSOCIATED DATA ITEM
```

Type. Error

Cause. A numeric literal named in the MUST BE clause contains too many digits to the right of the decimal.

595

```
MUST BE VALUE TOO LARGE FOR ASSOCIATED DATA ITEM
```

Type. Error

Cause. A numeric literal named in the MUST BE clause exceeds the size of the data item PICTURE clause.

596

```
TYPE OF MUST BE VALUE IS INCOMPATIBLE WITH ASSOCIATED DATA
ITEM
```

Type. Error

Cause. The type of literal named in the MUST BE clause does not match the associated data item. A numeric literal must be associated with a numeric data item; a nonnumeric literal must be associated with a nonnumeric data item.

597

```
QUALIFIED NAME TOO LONG - CROSSREF LINE WILL BE TRUNCATED
```

Type. Warning

Cause. During compilation, SCOBOLX builds fully qualified names to send to CROSSREF. A name exceeded the length of the buffer and will appear in the CROSSREF listing in truncated form. This might occur with multiple levels of qualification.

598

```
MUST NOT BE OVERLAY AREA NAME
```

Type. Error

Cause. An overlay area name is not allowed in this context.

599

```
EXPECT LEVEL 01 MESSAGE NAME
```

Type. Error

Cause. Compiler is looking for an 01-level message entry or data item; this message or data item has higher level number.

600

```
EXPECT DATA ITEM OR MESSAGE NAME
```

Type. Error

Cause. Compiler expects specified name to identify an existing 01-level data item or message entry.

601

```
THIS VERB IS NOT LEGAL WHEN 'TERMINAL IS INTELLIGENT'
```

Type. Error

Cause. You cannot use this verb in a SCREEN COBOL program that runs in a Pathway environment where terminal type was specified as INTELLIGENT in a SET TERM, SET PROGRAM, or RUN PROGRAM command.

602

```
THIS VERB IS NOT LEGAL WITHOUT 'TERMINAL IS INTELLIGENT'
```

Type. Error

Cause. In order to use this verb, you must specify terminal type as INTELLIGENT in a SET TERM, SET PROGRAM, or RUN PROGRAM command.

603

```
MESSAGE FORMAT MUST BE THE SAME IN ALL REPLIES
```

Type. Error

Cause. All replies must have the same message format.

604

```
REPLY CODE FIELD MUST BE WORKING-STORAGE ITEM
```

Type. Error

Cause. The REPLY CODE FIELD IS clause specified within a SEND MESSAGE statement referenced a field that is not in Working-Storage.

605

```
ITEMS IN TRANSFORM LIST MUST BE ALL OF THE SAME TYPE
```

Type. Error

Cause. All items specified in either the input or output list of a TRANSFORM statement must be the same type, all Working-Storage or all Message Section items.

606

```
'MUST BE' AGGREGATE LENGTHS GREATER THAN 768 BYTE MAXIMUM
```

Type. Error

Cause. The total length, in bytes, of the literals specified in a Screen Section MUST BE clause plus 1 byte for each literal specified cannot exceed 768 bytes.

607

```
VARYING1 255-BYTE MAXIMUM LENGTH EXCEEDED
```

Type. Error

Cause. A message of VARYING1 format has a maximum length of 255 bytes.

608

```
ILLEGAL FIGURATIVE CONSTANT SPECIFIED IN MOVE TO DBCS ITEM
```

Type. Error

Cause. An attempt was made to move a figurative constant other than SPACE or SPACES to a double-byte data item.

609

```
QUOTED LITERAL STRING CONTAINS NON-DBCS IN MOVE TO DBCS ITEM
```

Type. Error

Cause. An attempt was made to move a quoted literal string containing non-double-byte characters to a PIC N data item.

610

```
ILLEGAL DATA ITEM OR LITERAL IN DOUBLEBYTE TEST
```

Type. Error

Cause. An attempt was made to use the IF ... DOUBLEBYTE statement on data items that are not PIC X, PIC A, or PIC N.

611

```
SCREEN ITEM REQUIRES 'AT <OVERLAY>' QUALIFIER
```

Type. Error

Cause. Screen item needs AT <OVERLAY> in this context.

612

```
MUST BE ELEMENTARY SCREEN ITEM
```

Type. Error

Cause. Only an elementary screen item is valid in this context.

613

```
SELECT or REPLY CODE ITEM HAS BEEN SYNCHRONIZED
```

Type. Warning

Cause. A SELECT or REPLY CODE item is synchronized.

614

```
CURSOR WITHIN CLAUSE USED WITH SCREEN FIELD THAT WRAPS
```

Type. Warning

Cause. This screen field wraps.

615

```
MNEMONIC USED MUST BE A DISPLAY VIDEO ATTRIBUTE
```

Type. Error

Cause. A display video attribute is expected here.

616

```
'CODE OTHER' MUST BE LAST STMT IN REPLY CODE LIST OF SEND
```

Type. Error

Cause. In a set of yield groups belonging to any one SEND statement the yield group containing OTHER must be last.

617

```
THERE CAN ONLY BE ONE 'CODE OTHER' IN A SEND CLAUSE
```

Type. Error

Cause. There is more than one OTHER in a SEND statement.

618

```
THERE IS AN ERROR IN THIS ACCEPT STATEMENT
```

Type. Error

Cause. This ACCEPT statement is missing a clause such as FROM, TO, and so on.

619

```
A PROMPT IS NOT ALLOWED IN BLOCK MODE
```

Type. Error

Cause. Block mode programs cannot use the PROMPT clause.

620

```
THE MAXIMUM NUMBER OF REPLY CODES PER YIELD GROUP IS 255
```

Type. Error

Cause. Each yield group can have at most 255 reply codes.

D

Errors for Message Section Statements

This appendix contains the information necessary to process errors that occur during the execution of the Message Section statements used to implement Intelligent Device Support (IDS).

If the ON ERROR clause is included in the statement and an error occurs, the TERMINATION-STATUS special register is set to the specified error number. These error numbers are described in the following pages.

If ON ERROR is not specified, TERMINATION-STATUS is not set to an error number. The system suspends the program with a fatal error. The TCP error is logged to the PATHMON log file.

TERMINATION-STATUS 1 Pathway Error 3161

```
I/O ERROR INTELLIGENT DEVICE
```

Cause. A file-system error occurred during an input or output operation. This error applies to the following statements:

- SEND MESSAGE statement
- RECEIVE UNSOLICITED MESSAGE statement
- REPLY TO UNSOLICITED MESSAGE statement

Effect. TERMINATION-SUBSTATUS contains the file system error number.

If no ON ERROR clause is specified, the system suspends the program unit with a fatal error. The TCP error is logged to the PATHMON log file.

TERMINATION-STATUS 2 Pathway Error 3162

```
RECEIVED MESSAGE SMALLER THAN EXPECTED
```

Cause. The message applies to one of the following conditions.

- If the error is for a SEND MESSAGE statement or a TRANSFORM statement, the received message was shorter than the reply message specified in the YIELDS list.
- If the error is for a RECEIVE UNSOLICITED MESSAGE statement, the unsolicited message was shorter than the YIELDS specification.

Effect. The message applies to one of the following conditions.

- If the error is for a SEND MESSAGE statement or a TRANSFORM statement, TERMINATION-SUBSTATUS contains the number of bytes received.

- If the error is for a RECEIVE UNSOLICITED MESSAGE statement, the message is rejected. An error 10 (COULD NOT DELIVER) is returned by the TCP to the sender of the message.

If no ON ERROR clause is specified, the system suspends the program unit with a fatal error. The TCP error is logged to the PATHMON log file.

TERMINATION-STATUS 3 Pathway Error 3163

```
RECEIVED MESSAGE LARGER THAN EXPECTED
```

Cause. The message applies to one of the following conditions.

- If the error is for a SEND MESSAGE statement or a TRANSFORM statement, the received message was longer than the reply message specified in the YIELDS specification.
- If the error is for a RECEIVE UNSOLICITED MESSAGE statement, the message is rejected.

Effect. The message applies to one of the following conditions.

- If the error is for a SEND MESSAGE statement, TERMINATION-SUBSTATUS contains the number of bytes received. Note that due to the nature of data overflow I/O errors, the number of bytes received may be less than the number of bytes sent.
- If the error is for a TRANSFORM statement, TERMINATION-SUBSTATUS contains the number of bytes of the <trans-rec-out> data to be transformed.
- If the error is for a RECEIVE UNSOLICITED MESSAGE statement, the message is rejected. An error 10 (COULD NOT DELIVER) is returned by the TCP to the sender of the message.

If no ON ERROR clause is specified, the system suspends the program unit with a fatal error. The TCP error is logged to the PATHMON log file.

TERMINATION-STATUS 4 Pathway Error 3164

```
REPLY CODE OF RECEIVED MESSAGE UNDEFINED
```

Cause. The message applies to one of the following conditions.

- If the error is for a SEND MESSAGE statement or a TRANSFORM statement, the code in the received message did not match any of the reply codes or select codes specified in the YIELDS or SELECT list, respectively.
- If the error is for a RECEIVE UNSOLICITED MESSAGE statement, the code received did not match any of the receive code values specified.

Effect. If the error is for a RECEIVE UNSOLICITED MESSAGE statement, the unsolicited message is rejected. An error 10 (COULD NOT DELIVER) is returned by the TCP to the sender.

If no ON ERROR clause is specified, the system suspends the program unit with a fatal error. The TCP error is logged to the PATHMON log file.

TERMINATION-STATUS 5 Pathway Error 3165

```
EDIT ERROR OCCURRED ON MESSAGE INPUT
```

Cause. The message applies to one of the following conditions.

- If the error is for a SEND MESSAGE statement or a TRANSFORM statement, an edit error occurred on input.
- If the error is for a RECEIVE UNSOLICITED MESSAGE statement, an edit error occurred on input of an unsolicited message.

Effect. This error is returned only for messages mapped through the Message Section. If the error is for a RECEIVE UNSOLICITED MESSAGE statement, the unsolicited message is rejected. An error 10 (COULD NOT DELIVER) is returned by the TCP to the sender of the unsolicited message. If no ON ERROR clause is specified, the system suspends the program unit with a fatal error. The TCP error is logged to the PATHMON log file.

TERMINATION-STATUS 6 Pathway Error 3166

```
RECEIVED MESSAGE EXCEEDS MAXIMUM ALLOWABLE SIZE
```

Cause. The message received was larger than 32,000 bytes. This error applies to the following statements:

- SEND MESSAGE statement
- TRANSFORM statement

Effect. If no ON ERROR clause is specified, the system suspends the program unit with a fatal error. The TCP error is logged to the PATHMON log file.

TERMINATION-STATUS 7 Pathway Error 3167

```
MESSAGE TO SEND EXCEEDS MAXIMUM ALLOWABLE SIZE
```

Cause. The assembled message for output was larger than 32,000 bytes. This error applies to the following statements:

- SEND MESSAGE statement
- TRANSFORM statement

Effect. If no ON ERROR clause is specified, the system suspends the program unit with a fatal error. The TCP error is logged to the PATHMON log file.

TERMINATION-STATUS 8 Pathway Error 3168

```
DEVICE SUBCLASS INVALID
```

Cause. In the PATHCOM SET TERM command, the TYPE parameter is specified by *terminal-type*:*terminal-subtype*. For *terminal-type* INTELLIGENT, *terminal-subtype* was a value other than 0 or 1.

This error applies to the SEND MESSAGE statement.

Effect. If no ON ERROR clause is specified, the system suspends the program unit with a fatal error. The TCP error is logged to the PATHMON log file.

TERMINATION-STATUS 9 Pathway Error 3169

```
ILLEGAL TIMEOUT VALUE
```

Cause. An illegal timeout value was specified for the statement. The illegal value is either lesser than zero or greater than 21474836. This error applies to the following statements:

- SEND MESSAGE statement
- RECEIVE UNSOLICITED MESSAGE statement

Effect. If no ON ERROR clause is specified, the system suspends the program unit with a fatal error. The TCP error is logged to the PATHMON log file.

TERMINATION-STATUS 10 Pathway Error 3170

```
INVALID END OF MESSAGE CHARACTER ENCOUNTERED
```

Cause. An invalid end-of-message was detected. This error is returned only for messages that are mapped through the Message Section. This error applies to the following statements:

- SEND MESSAGE statement
- RECEIVE UNSOLICITED MESSAGE statement

Effect. For unsolicited messages, the message is rejected. An error 10 (COULD NOT DELIVER) is returned by the sender of the unsolicited message.

If no ON ERROR clause is specified, the system suspends the program unit with a fatal error. The TCP error is logged to the PATHMON log file.

TERMINATION-STATUS 11 Pathway Error 3171

```
FIELD LENGTH EXCEEDS MAXIMUM ALLOWABLE
```

Cause. The data in a particular message field was too large to be mapped through the corresponding field in the Message Section template. This error applies to the following statements:

- SEND MESSAGE statement
- TRANSFORM statement
- REPLY TO UNSOLICITED MESSAGE statement

Effect. If no ON ERROR clause is specified, the system suspends the program unit with a fatal error. The TCP error is logged to the PATHMON log file.

TERMINATION-STATUS 12 Pathway Error 3172

```
MESSAGE LENGTH EXCEEDS MAXIMUM ALLOWED
```

Cause. The message that you attempted to send was longer than 12,288 bytes. This error applies to the following statements:

- SEND MESSAGE statement
- TRANSFORM statement

Effect. If no ON ERROR clause is specified, the system suspends the program unit with a fatal error. The TCP error is logged to the PATHMON log file.

TERMINATION-STATUS 13 Pathway Error 3173

```
I/O ERROR ON CONTROL-26 OPERATION
```

Cause. An unsolicited message arrived. The file system reported an I/O error while attempting to terminate an outstanding READ operation. This error applies to the SEND MESSAGE statement.

Effect. TERMINATION-SUBSTATUS contains the error. If no ON ERROR clause is specified, the system suspends the program unit with a fatal error. The TCP error is logged to the PATHMON log file.

TERMINATION-STATUS 14 Pathway Error 3174

```
CONTROL-26 OPERATION DID NOT COMPLETE IN TIME
```

Cause. An attempted termination of an outstanding READ operation, or its associated I/O, did not complete within 5 minutes. This error applies to the SEND MESSAGE statement.

Effect. If no ON ERROR clause is specified, the system suspends the program unit with a fatal error. The TCP error is logged to the PATHMON log file.

TERMINATION-STATUS 15 Pathway Error 3175

```
EDIT ERROR OCCURRED ON MESSAGE OUTPUT
```

Cause. An edit error was detected while the TCP was building the output message. This error applies to the following statements:

- SEND MESSAGE statement
- TRANSFORM statement
- REPLY TO UNSOLICITED MESSAGE statement

Effect. A TERMINATION-SUBSTATUS of 15 is returned only for messages mapped through the Message Section. If no ON ERROR clause is specified, the system suspends the program unit with a fatal error. The TCP error is logged to the PATHMON log file.

TERMINATION-STATUS 16 Pathway Error 3176

```
ATTEMPT TO RECEIVE UNSOLICITED MESSAGE WITH ONE NOT YET
REPLIED TO
```

Cause. An attempt was made to receive an unsolicited message but no reply has been made to the previous message. This error applies to the RECEIVE UNSOLICITED MESSAGE statement.

Effect. If no ON ERROR clause is specified, the system suspends the program unit with a fatal error. The TCP error is logged to the PATHMON log file. The SCREEN COBOL program unit needs to execute the REPLY TO UNSOLICITED MESSAGE statement.

TERMINATION-STATUS 17 Pathway Error 3177

```
NO UNSOLICITED MESSAGE TO REPLY TO
```

Cause. An attempt was made to execute the REPLY TO UNSOLICITED MESSAGE statement but no RECEIVE UNSOLICITED MESSAGE statement had been previously issued. This error applies to the REPLY TO UNSOLICITED MESSAGE statement.

Effect. If no ON ERROR clause is specified, the system suspends the program unit with a fatal error. The TCP error is logged to the PATHMON log file.

TERMINATION-STATUS 18 Pathway Error 3178

```
ATTEMPT TO RECEIVE UNSOLICITED MESSAGE WHEN TERM MAXINPUTMSGS
= 0
```

Cause. The PATHCOM SET TERM parameter MAXINPUTMSGS is set to 0 (which is the default). Therefore, an attempt was made to receive an unsolicited message, but the front-end process is not queueing unsolicited messages. This error applies to the RECEIVE UNSOLICITED MESSAGE statement.

Effect. If no ON ERROR clause is specified, the system suspends the program unit with a fatal error. The TCP error is logged to the PATHMON log file.

TERMINATION-STATUS 19 Pathway Error 3179

```
DATA LEFT OVER ON SCATTER TO WORKING STORAGE
```

Cause. The output (source) record was longer than the input (destination) record specified in the YIELDS list. This error applies to the TRANSFORM statement.

Effect. If no ON ERROR clause is specified, the system suspends the program unit with a fatal error. The TCP error is logged to the PATHMON log file.

TERMINATION-STATUS 20 Pathway Error 3180

```
NOT ENOUGH DATA FOR SCATTER TO WORKING STORAGE
```

Cause. The output (source) record was shorter than the input (destination) record specified in the YIELDS list. This error applies to the TRANSFORM statement.

Effect. If no ON ERROR clause is specified, the system suspends the program unit with a fatal error. The TCP error is logged to the PATHMON log file.

TERMINATION-STATUS 21 Pathway Error 3181

```
VARIABLE FIELD SIZE WOULD EXCEED DECLARED FIELD SIZE
```

Cause. There is an illegal RESULTING COUNT value in relation to the Message Section field's declared size. This error applies to the following statements:

- SEND MESSAGE statement
- TRANSFORM statement
- RECEIVE UNSOLICITED MESSAGE statement
- REPLY TO UNSOLICITED MESSAGE statement

Effect. This error is returned when a field being processed during delimited field processing is larger than its declared size. This error is returned only for messages mapped through the Message Section.

If the error is for a RECEIVE UNSOLICITED MESSAGE statement, the unsolicited message that the RECEIVE UNSOLICITED MESSAGE attempted to receive is rejected. An error 10 (COULD NOT DELIVER) is returned by the TCP to the sender of the message.

If no ON ERROR clause is specified, the system suspends the program unit with a fatal error. The TCP error is logged to the PATHMON log file.

TERMINATION-STATUS 23 Pathway Error 3183

```
DEPENDING VALUE IS OUT OF BOUNDS
```

Cause. For a field defined by an OCCURS DEPENDING ON clause, the number of occurrences specified by the depending variable's value is outside of the range declared by the OCCURS DEPENDING ON clause. This error applies to the following statements:

- SEND MESSAGE statement
- TRANSFORM statement
- RECEIVE UNSOLICITED MESSAGE statement
- REPLY TO UNSOLICITED MESSAGE statement

Effect. If no ON ERROR clause is specified, the system suspends the program unit with a fatal error. The TCP error is logged to the PATHMON log file.

TERMINATION-STATUS 24 Pathway Error 3184

```
CONFLICT OF DATA TYPES DURING 'PRESENT IF' PROCESSING
```

Cause. The PRESENT IF clause defines a field within a message to be optionally present based on the value of a control field. The data in this control field is not of the declared data type. (For example, a numeric data item contains nonnumeric data.) This error applies to the following statements:

- SEND MESSAGE statement
- TRANSFORM statement
- RECEIVE UNSOLICITED MESSAGE statement
- REPLY TO UNSOLICITED MESSAGE statement

Effect. If no ON ERROR clause is specified, the system suspends the program unit with a fatal error. The TCP error is logged to the PATHMON log file.

TERMINATION-STATUS 25 Pathway Error 3185

```
FIELD OCCURRENCE EXCEEDS WORKING STORAGE MAXIMUM
```

Cause. A message field's occurrences exceed its related Working-Storage data item's maximum occurrences.

Effect. If no ON ERROR clause is specified, the system suspends the program unit with a fatal error. The TCP error is logged to the PATHMON log file.

TERMINATION-STATUS 25 Pathway Error 3186

```
STATISTICS GATHERING SUSPENDED, FILE ERROR
```

Cause. A file system error has occurred on the performance statistics collection file.

Effect. The performance statistics collection process is suspended.

TERMINATION-STATUS 25 Pathway Error 3187

```
STATISTICS GATHERING STARTED, OUTPUT FILE
```

Cause. The TCP statistics gathering process has started; a result of issuing a START TCP or CONTROL <tcp-name> MEASURE ON command.

Effect. None

TERMINATION-STATUS 25 Pathway Error 3188

```
STATISTICS GATHERING STOPPED, OUTPUT FILE
```

Cause. The TCP statistics gathering process has stopped; a result of issuing a CONTROL <tcp-name> MEASURE OFF command while in operation.

Effect. None.

E

SCREEN COBOL Reserved Words

ABORT
ABORT-INPUT
ABORT-TRANSACTION
ABSENT
ACCEPT
ACCESS
ADD
ADVANCING
ADVISORY
AFTER
ALARM
ALL
ALPHABETIC
ALSO
ALTER
ALTERNATE
AND
APPROXIMATE
ARE
AREA
AREAS
ASCENDING
ASSIGN
AT
ATTR
AUDIBLE
AUTHOR

BASE
BE
BEFORE
BEGIN-TRANSACTION
BLANK
BLOCK
BOTTOM
BY

CALL
CANCEL
CD
CF
CH
CHARACTER
CHARACTERS
CHARACTER-SET
CHECKPOINT
CLEAR
CLOCK-UNITS
CLOSE
COBOL
CODE
CODE-SET
COLLATING
COLUMN
COLUMNS
COMMA
COMMUNICATION
COMP
COMPUTATIONAL
COMPUTE
CONFIGURATION
CONTAINS
CONTROL
CONTROLLED
CONTROLS
CONVERSATIONAL
CONVERSION
COPY
CORR
CORRESPONDING
COUNT
CROSSREF
CURRENCY

DATA
DATE
DATE-COMPILED
DATE-WRITTEN
DAY
DE
DEBUG-CONTENTS
DEBUG-ITEM
DEBUG-LINE
DEBUG-NAME
DEBUG-SUB-1
DEBUG-SUB-2
DEBUG-SUB-3
DEBUGGING
DECIMAL-POINT
DECLARATIVES
DELAY
DELETE
DELIMITED
DELIMITER
DEPENDING
DESCENDING
DESTINATION
DETAIL
DIAGNOSTIC-ALLOWED
DISABLE
DISPLAY
DIVIDE
DIVISION
DOUBLEBYTE
DOWN
DUPLICATES
DYNAMIC

EGI
ELSE
EMI
ENABLE
END
END-OF-INPUT
END-OF-PAGE
END-TRANSACTION
ENTER
ENVIRONMENT
EOP
EQUAL
ERROR
ERROR-ENHANCEMENT
ESCAPE
ESI
EVERY
EXCEPTION
EXCLUSIVE
EXIT
EXTEND

FD
FIELD-SEPARATOR
FILE
FILE-CONTROL
FILL
FILLER
FINAL
FIRST
FIXED-LENGTH
FOOTING
FOR
FROM
FULL

GENERATE
GENERIC
GIVING
GO
GREATER
GROUP
GROUP-SEPARATOR

HEADING
HIGH-VALUE
HIGH-VALUES

I-O
I-O-CONTROL
I-O-ERROR
IDENTIFICATION
IF
IN
INDEX
INDEXED
INDICATE
INITIAL
INITIATE
INPUT
INPUT-OUTPUT
INSPECT
INSTALLATION
INTO
INVALID
IS

JUST
JUSTIFIED

KANJI-KATAKANA
KEY

LABEL
LAST
LEADING
LEFT
LENGTH
LESS
LIKE
LIMIT
LIMITS
LINAGE
LINAGE-COUNTER
LINE
LINE-COUNTER
LINES
LINKAGE
LOCK
LOCKFILE
LOGICAL-TERMINAL-NAME
LOW-VALUE
LOW-VALUES
LP-ATTENTION-KEY
LP-ENTER-KEY
LP-SELECTABLE

MEMORY
MERGE
MESSAGE
MINIMUM-ATTR
MINIMUM-COLOR
MODE
MODEM
MODULES
MOVE
MULTIPLE
MULTIPLY
MUST

NATIVE
NEGATIVE
NEW-CURSOR
NEW-CURSOR-COL
NEW-CURSOR-ROW
NEXT
NO
NOSHADOW
NOT
NUMBER
NUMERIC
NUMERIC-SHIFT

OBJECT-COMPUTER
OCCURS
OF
OFF
OFFSET
OLD-CURSOR
OLD-CURSOR-COL
OLD-CURSOR-ROW
OMITTED
ON
ONE
OPEN
OPTIONAL
OR
ORGANIZATION
OUTPUT
OVERFLOW
OVERLAY
PAGE
PAGE-COUNTER
PATHWAY
PERFORM
PF
PH
PIC
PICTURE
PLUS
POINTER
POSITION
POSITIVE
PRINT
PRINTING
PROCEDURE
PROCEDURES
PROCEED
PROGRAM
PROGRAM-ID
PROGRAM-STATUS
PROGRAM-STATUS-1
PROGRAM-STATUS-2
PROMPT
PROTECT
PW-INPUT-FIELDS-MISSING
PW-TCP-PROCESS-NAME
PW-TCP-SYSTEM-NAME
PW-TERMINAL-ERROR-OCCURRED
PW-UNSOLICITED-MESSAGE-QUEUED
PW-USE-NEW-CURSOR

QUEUE
QUOTE
QUOTES

RANDOM
RD
READ
RECEIVE
RECEIVE-CONTROL
RECONNECT
RECORD
RECORDS
RECOVERY
REDEFINES
REDISPLAY
REEL
REFERENCES
RELATIVE
RELEASE

REMAINDER
REMOVAL
RENAMES
REPLACING
REPLY
REPORT
REPORTING
REPORTS
RERUN
RESERVE
RESET
RESTART-COUNTER
RESTART-INPUT
RESTART-TRANSACTION
RETURN
REVERSED
REWIND
REWRITE
RF
RH
RIGHT
ROUNDED
RUN

SAME
SCREEN
SCREEN-CONTROL
SCROLL
SD
SEARCH
SECTION
SECURITY
SEGMENT
SEGMENT-LIMIT
SELECT
SEND
SENTENCE
SEPARATE
SEQUENCE
SEQUENTIAL
SET
SHADOWED
SHARED
SIGN
SIZE
SKIP
SKIPPING
SORT
SORT-MERGE
SOURCE
SOURCE-COMPUTER
SPACE
SPACES
SPECIAL-NAMES
STANDARD
STANDARD-1
START
STARTBACKUP
STATUS
STOP
STOP-MODE
STRING
SUB-QUEUE-1
SUB-QUEUE-2
SUB-QUEUE-3
SUBTRACT
SUM
SUPPRESS
SYMBOLIC
SYNC
SYNCDEPTH
SYNCHRONIZED
SYSTEM

TAB
TABLE
TAL
TALLYING
TAPE
TELL-ALLOWED
TEMP
TEMPORARY
TERMINAL
TERMINALINFO
TERMINAL-ERROR-OCCURRED
TERMINAL-FILENAME
TERMINAL-PRINTER
TERMINATE
TERMINATION-STATUS
TERMINATION-SUBSTATUS
TEXT
THAN
THROUGH
THRU
TIME
TIMEOUT
TIMES
TO
TOP
TRAILING
TRANSACTION-ID
TRANSFORM
TRANSPARENT
TURN

UNDER
UNIT
UNLOCK
UNLOCKFILE
UNLOCKRECORD
UNSOLICITED
UNSTRING
UNTIL
UP
UPON
UPSHIFT
USAGE
USE
USER
USING

VALUE
VALUES
VARYING

WHEN
WITH
WITHIN
WORDS
WORKING-STORAGE
WRITE

YIELDS
YYYYDDD
YYYYMMDD

ZERO
ZEROES
ZEROS

Data Type Correspondence and Return Value Sizes

The following tables contain the return value size generated for each data type by Compaq language compilers for Compaq *NonStop*™ *Himalaya* systems. Use this information when you need to specify values with the Accelerator ReturnValSize option. These tables are also useful if your programs use data from files created by programs in another language, or your programs pass parameters to programs written in callable languages.

Refer to the appropriate Compaq *NonStop*™ Structured Query Language/MP (NonStop™ SQL/MP) programming manual for a complete list of SQL data type correspondences. Note that the return value sizes given in these tables do not correspond to the storage size of SQL data types.

Note. Information labeled as "COBOL" applies to COBOL 74, COBOL85, and SCREEN COBOL unless otherwise noted.

If you are using the Data Definition Language (DDL) utility to describe your files, you might not need these tables. See the *Data Definition Language (DDL) Reference Manual* for more information.

Table F-1. Integer Types, Part 1 (page 1 of 2)

	8-Bit Integer	16-Bit Integer	32-Bit Integer
C	char [1] unsigned char signed char	int short unsigned	long unsigned long
COBOL	Alphabetic Numeric DISPLAY Alphanumeric-Edited Alphanumeric Numeric-Edited	PIC S9(n) COMP or PIC 9(n) COMP without P or V, 1 ð n ð 4 Index Data Item [2] NATIVE-2 [3]	PIC S9(n) COMP or PIC 9(n) COMP without P or V, 5 ð n ð 9 Index Data Item [2] NATIVE-4 [3]
FORTRAN	—	INTEGER [4] INTEGER*2	INTEGER*4

[1] Unsigned Integer.

[2] Index Data Item is a 16-bit integer in COBOL 74 and a 32-bit integer in COBOL85.

[3] COBOL85 only.

[4] INTEGER is normally equivalent to INTEGER*2. The INTEGER*4 and INTEGER*8 compiler directives redefine INTEGER.

Table F-1. Integer Types, Part 1 (page 2 of 2)

	8-Bit Integer	16-Bit Integer	32-Bit Integer
Pascal	BYTE Enumeration, unpacked, ð 256 members Subrange, unpacked, n…m, 0 ð n and mð 255	INTEGER INT16 CARDINAL [1] BYTE or CHAR value parameter Enumeration, unpacked, > 256 members Subrange, unpacked, n…m, -32768 ð n and m ð 32767, but at least n or m outside 0…255 range	LONGINT INT32 Subrange, unpacked n…m, –2147483648 ð n and m ð 2147483647, but at least n or m outside -32768…32767 range
SQL	CHAR	NUMERIC(1)…NUMERIC(4) PIC 9(1) COMP…PIC 9(4) COMP SMALLINT	NUMERIC(5)…NUMERIC(9) PIC 9(1) COMP…PIC 9(9) COMP INTEGER
TAL	STRING UNSIGNED(8)	INT UNSIGNED(16)	INT(32)
Return Value Size (Words)	*1*	*1*	*2*

[1] Unsigned Integer.

[2] Index Data Item is a 16-bit integer in COBOL 74 and a 32-bit integer in COBOL85.

[3] COBOL85 only.

[4] INTEGER is normally equivalent to INTEGER*2. The INTEGER*4 and INTEGER*8 compiler directives redefine INTEGER.

Table F-2. Integer Types, Part 2 (page 1 of 2)

	64-Bit Integer	Bit Integer of 1 to 31 Bits	Decimal Integer
C	long long	—	—
COBOL	PIC S9(n) COMP or PIC 9(n) COMP without P or V, 10 ð n ð 18 NATIVE-8 [1]	—	Numeric DISPLAY
FORTRAN	INTEGER*8	—	—

[1] COBOL85 only.

Pascal	INT64	UNSIGNED(n), 1 ð n ð 16 INT(n), 1 ð n ð 16	DECIMAL

Table F-2. Integer Types, Part 2 (page 2 of 2)

	64-Bit Integer	Bit Integer of 1 to 31 Bits	Decimal Integer
SQL	NUMERIC(10)… NUMERIC(18) PIC 9(10) COMP… PIC 9(18) COMP INTEGER	—	DECIMAL (n,s) PIC 9(n) DISPLAY
TAL	FIXED(0)	UNSIGNED(n) , 1 ð n ð 31	—
Return Value Size (Words)	*4*	*1, 1 or 2 in TAL*	*1 or 2, depends on declared pointer size*

[1] COBOL85 only.

Table F-3. Floating, Fixed, and Complex Types

	32-Bit Floating	64-Bit Floating	64-Bit Fixed Point	64-Bit Complex
C	float	double	—	—
COBOL	—	—	PIC S9(n–s)v9(s) COMP or PIC 9(n–s)v9(s) COMP, 10 ð n ð 18	—
FORTRAN	REAL	DOUBLE PRECISION	—	COMPLEX
Pascal	REAL	LONGREAL	—	—
SQL	—	—	NUMERIC (n,s) PIC 9(n-s)v9(s) COMP	—
TAL	REAL	REAL(64)	FIXED(s), -19 ð s ð 19	—
Return Value Size (Words)	*2*	*4*	*4*	*4*

Table F-4. Character Types

	Character	Character String	Varying Length Character String

C	signed char
	unsigned char
		pointer to char	struct {
			int len;
			char val [n]
			};

COBOL	Alphabetic Numeric DISPLAY Alphanumeric-Edited Alphanumeric Numeric-Edited	Alphabetic Numeric DISPLAY Alphanumeric-Edited Alphanumeric Numeric-Edited	01 name. 03 len USAGE IS NATIVE-2 [1] 03 val PIC X(n).
FORTRAN	CHARACTER	CHARACTER array CHARACTER*n	—
Pascal	CHAR or BYTE value parameter Enumeration, unpacked, ð 256 members Subrange, unpacked n…m, 0 ð n and m ð 255	PACKED ARRAY OF CHAR FSTRING(n)	STRING(n)
SQL	PIC X CHAR	CHAR(n) PIC X(n)	VARCHAR(n)
TAL	STRING	STRING array	—
Return Value Size (Words)	*1*	*1 or 2, depends on declared pointer size*	*1 or 2, depends on declared pointer size*

[1] COBOL85 only.

Table F-5. Structured, Logical, Set, and File Type (page 1 of 2)

	Byte-Addressed Structure	Word-Addressed Structure	Logical (true or false)	Boolean	Set	File
C	—	struct	—	—	—	—
COBOL	—	01-level RECORD	—	—	—	—
FORTRAN	RECORD	—	LOGICAL [1]	—	—	—
Pascal	RECORD, byte-aligned	RECORD, word-aligned	—	BOOLEAN	Set	File

[1] LOGICAL is normally defined as 2 bytes. The LOGICAL*2 and LOGICAL*4 compiler directives redefine LOGICAL.

SQL	—	—	—	—	—	—

Table F-5. Structured, Logical, Set, and File Type (page 2 of 2)

	Byte-Addressed Structure	Word-Addressed Structure	Logical (true or false)	Boolean	Set	File
TAL	Byte-addressed standard STRUCT pointer	Word-addressed standard STRUCT pointer	—	—	—	—
Return Value Size (Words)	*1 or 2, depends on declared pointer size*	*1 or 2, depends on declared pointer size*	*1 or 2, depends on compiler directive*	*1*	*1*	*1*

[1] LOGICAL is normally defined as 2 bytes. The LOGICAL*2 and LOGICAL*4 compiler directives redefine LOGICAL.

Table F-6. Pointer Types

	Procedure Pointer	Byte Pointer	Word Pointer	Extended Pointer
C	function pointer	byte pointer	word pointer	extended pointer
COBOL	—	—	—	—
FORTRAN	—	—	—	—
Pascal	Procedure pointer	Pointer, byte-addressed BYTEADDR	Pointer, byte-addressed WORDADDR	Pointer, extended-addressed EXTADDR
SQL	—	—	—	—
TAL	—	16-bit pointer, byte-addressed	16-bit pointer, word-addressed	32-bit pointer
Return Value Size (Words)	*1 or 2, depends on declared pointer size*	*1 or 2, depends on declared pointer size*	*1 or 2, depends on declared pointer size*	*1 or 2, depends on declared pointer size*

Index

Numbers

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Special Characters