CA Ideal™ for CA Datacom®

Creating Programs Guide

Version 14.02

This Documentation, which includes embedded help systems and electronically distributed materials (hereinafter referred to as the “Documentation”), is for your informational purposes only and is subject to change or withdrawal by CA at any time.

This Documentation may not be copied, transferred, reproduced, disclosed, modified or duplicated, in whole or in part, without the prior written consent of CA. This Documentation is confidential and proprietary information of CA and may not be disclosed by you or used for any purpose other than as may be permitted in (i) a separate agreement between you and CA governing your use of the CA software to which the Documentation relates; or (ii) a separate confidentiality agreement between you and CA.

Notwithstanding the foregoing, if you are a licensed user of the software product(s) addressed in the Documentation, you may print or otherwise make available a reasonable number of copies of the Documentation for internal use by you and your employees in connection with that software, provided that all CA copyright notices and legends are affixed to each reproduced copy.

The right to print or otherwise make available copies of the Documentation is limited to the period during which the applicable license for such software remains in full force and effect. Should the license terminate for any reason, it is your responsibility to certify in writing to CA that all copies and partial copies of the Documentation have been returned to CA or destroyed.

TO THE EXTENT PERMITTED BY APPLICABLE LAW, CA PROVIDES THIS DOCUMENTATION “AS IS” WITHOUT WARRANTY OF ANY KIND, INCLUDING WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT. IN NO EVENT WILL CA BE LIABLE TO YOU OR ANY THIRD PARTY FOR ANY LOSS OR DAMAGE, DIRECT OR INDIRECT, FROM THE USE OF THIS DOCUMENTATION, INCLUDING WITHOUT LIMITATION, LOST PROFITS, LOST INVESTMENT, BUSINESS INTERRUPTION, GOODWILL, OR LOST DATA, EVEN IF CA IS EXPRESSLY ADVISED IN ADVANCE OF THE POSSIBILITY OF SUCH LOSS OR DAMAGE.

The use of any software product referenced in the Documentation is governed by the applicable license agreement and such license agreement is not modified in any way by the terms of this notice.

The manufacturer of this Documentation is CA.

Provided with “Restricted Rights.” Use, duplication or disclosure by the United States Government is subject to the restrictions set forth in FAR Sections 12.212, 52.227-14, and 52.227-19(c)(1) - (2) and DFARS Section 252.227-7014(b)(3), as applicable, or their successors.

Copyright © 2015 CA. All rights reserved. All trademarks, trade names, service marks, and logos referenced herein belong to their respective companies.

CA Technologies Product References

This document references the following CA products:

- CA Datacom® CICS Services
- CA Datacom®/DB
- CA Datacom® SQL
- CA Ideal™ for Datacom® (CA Ideal)
- CA Ideal™ for DB2
- CA Ideal™ for VSAM

Contact CA Technologies

Contact CA Support

For your convenience, CA Technologies provides one site where you can access the information that you need for your Home Office, Small Business, and Enterprise CA Technologies products. At http://ca.com/support, you can access the following resources:

- Online and telephone contact information for technical assistance and customer services
- Information about user communities and forums
- Product and documentation downloads
- CA Support policies and guidelines
- Other helpful resources appropriate for your product

Providing Feedback About Product Documentation

If you have comments or questions about CA Technologies product documentation, you can send a message to techpubs@ca.com.

To provide feedback about CA Technologies product documentation, complete our short customer survey which is available on the CA Support website at http://ca.com/docs.

Contents

Chapter 8: Symbolic Debugger 143

Appendix A: Database Dependent Facilities 181

Chapter 1: Defining a Program

This guide provides information about creating a CA Ideal program with CA Ideal.

Programming Capabilities

The *Creating Programs Guide* introduces you to the CA Ideal program definition facility, integrating all aspects of creating a program in CA Ideal. It is organized according to functionality, presenting explanations of both the fill-in screens and the Program Definition Language (PDL) necessary to create the various parts of a program. You can learn how to create a program in CA Ideal from this guide. Once you learn the basics, you can use the *Programming Reference Guide* as a reference to the PDL programming language.

This chapter describes the Program Definition Facility that defines and maintains CA Ideal program definitions. This chapter includes the following topics:

- Creating a new program and its components
- Displaying and editing a program
- Duplicating a program to new name
- Deleting a program
- Printing a program
- Listing the programs that were already defined

Programs that CA Ideal uses can be either CA Ideal programs or programs written in the COBOL, PL/I, or Assembler language. CA Ideal defines and maintains CA Ideal programs. The CA Ideal programs are the ideal subprograms. The components of a CA Ideal program definition are explained in the following sections.

COBOL, PL/I, and Assembler programs are written and maintained outside of CA Ideal. You can use these non-Ideal programs only as subprograms of CA Ideal programs. Non-Ideal subprograms must be made known to CA Ideal by defining the program identification and parameter definition components in CA Ideal.

Components of a Program Definition Under CA Ideal

A CA Ideal program definition uses the following components:

- A *program identification* fill-in creates a program definition and provides descriptive information about it.
- A *program resources* fill-in specifies the dataviews to use, panels to transmit, reports to generate, and any subprograms called or initiated by this program.
- A *working data definition* fill-in that is used as a temporary "scratch pad" for local data.
- A *parameter definition* fill-in describes data passed to this program from a calling program.
- An *environment definition* fill-in specifies access plan (SQL precompile) options required for programs that use SQL to access CA Datacom/DB.
- A *procedure definition* fill-in enters PDL statements to express the logic, computations, report production, panel processing, database retrieval, or database maintenance procedures that make up the CA Ideal program. For information on PDL statements, refer to the *Programming Reference Guide*.

A COBOL, PL/I, or Assembler program definition uses only two components-the program identification and the parameter definition.

Accessing Program Definition Functions

The functions that CA Ideal provides define and maintain program definitions as presented in the Program Maintenance menu. To access this menu, select option 1 from the Main Menu.

```
 =>
--------------------------------------------------------------------------------
 IDEAL: PROGRAM MAINTENANCE   (001) TEST                        SYS: DOC  MENU

 Enter desired option number ===>        There are    7 options in this menu:

 1.  EDIT/DISPLAY         - Edit or display a program
 2.  CREATE               - Create a program
 3.  PRINT                - Print a program
 4.  DELETE               - Delete a program
 5.  MARK STATUS          - Mark program status to production or history
 6.  DUPLICATE            - Duplicate program to new name
 7.  DISPLAY INDEX        - Display index of program names in a system
```

The *Command Reference Guide* describes the syntax for the command equivalent to each menu selection.

When a fill-in is complete, press the Enter key or a function key to apply the modified data. Pressing the Enter key applies the data, but leaves the current fill-in displayed. To continue, enter the appropriate command or press the appropriate function key. Pressing the PF2 (RETURN) key returns the session to the CA Ideal Main Menu without applying the modified data (except in certain cases with RUN). Pressing the PA1 or PA2 key also ignores modified data. The PA1 key redisplays the panel with all fields blanked out (RESHOW). The PA2 key displays current function key assignments.

For more information about the CA Ideal environment, see the *Working in the Environment Guide*.

Program Function Key Assignments

The following functions are assigned to the PF keys in the Program Definition Facility. Commands shown below are assignments consistent throughout all CA Ideal facilities. The *Command Reference Guide* contains a complete description of all CA Ideal editing commands.

HELP (PF1)

Displays a panel or series of panels that contains information explaining how to complete the current function.

RETURN (PF2)

Returns from a help panel to the program component display or from the program to the menu that selects the program.

PRINT SCREEN (PF3)

Generates a printout of the current screen contents.

PROCEDURE (PF4)

Positions to the program procedure.

WORK (PF5)

Positions to the program's working data fill-in.

PARAMETER (PF6)

Positions to the program's parameter data fill-in.

SCROLL BACKWARD (PF7)

Displays the previous frame in the current component.

SCROLL FORWARD (PF8)

Displays the next frame in the current component.

FIND (PF9)

Finds the next occurrence of an alphanumeric literal previously specified in a full FIND command.

SCROLL TOP (PF10)

Positions to the first line of the component.

SCROLL BOTTOM (PF11)

Positions to the bottom of the component.

INPUT (PF12)

Opens a window of null lines preceding the first line of the component or at the current cursor position. Unused null lines in the window are deleted when you press the Enter key after INPUT.

Creating a New Program Definition

To create a new program definition, enter the CREATE PROGRAM command and fill in the required program definition panels to define the components of the program definition. As soon as you enter a name for the program on the command line or on the identification fill-in, an entity-occurrence is added to the dictionary for the new program. You can define the remaining components later on an as-needed basis.

The CREATE PROGRAM command and identification fill-in also identify a subprogram to CA Ideal.

The system's protection against editing CA Ideal production status programs does not apply to non-Ideal programs since they are not maintained in CA Ideal.

The following section describes how to define each component, whether the program is written in CA Ideal or in COBOL, PL/I, or Assembler.

Identifying the Program

The CREATE PROGRAM command initiates the creation process. After the CREATE command is executed, the program identification fill-in presented, and a program name entered, the new program definition becomes the current definition.

Version Number

CREATE PROGRAM creates the first version of a program definition. CA Ideal assigns a version number of 1 to a newly created program definition. This version of the definition is in TEST status. You can edit it as long as it remains in TEST status. After it is marked to PROD (production) status, you cannot edit it.

To mark a program to PROD status, enter the MARK STATUS command or select option 5 from the Program Maintenance Menu.

Note: The program must be successfully compiled before you can mark it to production status.

Program Identification Fill-in

The program identification fill-in enters descriptive information about the program when the program definition is initially created or when it is subsequently modified. All information entered into the program identification fill-in is stored in the Datadictionary. The following screen shows a completed fill-in.

When creating a program definition, the Identification fill-in automatically displays first. When editing an existing program definition, you must issue the IDENTIFICATION (IDE) command to display the Identification fill-in.

```
 =>
--------------------------------------------------------------------------------
 IDEAL: PGM IDENTIFICATION    PGM ADRMRPT   (001) TEST      SYS: DOC  FILL-IN
PROGRAM ADRMRPT

 Created          12/14/94                By JAEGER
 Last Modified    12/14/94 at 16:56       By JAEGER
 Last Compiled    12/18/94 at 16:50
 Run Status:         Private
 Short Description:  Pgm for IDEAL reports
 Language:  IDEAL    Target Date __ __ __ Actual Date __ __ __
 Description:
      ____________________________________________________________
```

The fields on the program identification fill-in are as follows:

PROGRAM

Displays the one- to eight-character name assigned to the program definition. CA Ideal initializes the program name to the name entered in the CREATE command or prompter if a name is supplied.

Created... By

Identifies the initial creation date of the program definition and the user ID of the creator. This information appears only after the program name was entered and accepted. CA Ideal maintains this date.

Last Modified ...at ...By

Identifies the date, time, and user ID of the last edit access. The field is blank until the program definition or an EDIT PGM command accesses it. The information appears only after the program name was entered and accepted. CA Ideal maintains this data.

Last Compiled at

Identifies the date and time of the last program compilation.

Run Status

For a CA Ideal program only, an indication of how the program is retained in memory while it is running. Possible values are SHARED, PRIVATE, or RESIDENT. The CA Ideal administrator determines the Run Status of a program. The default for newly created programs is PRIVATE. This line is omitted for non-Ideal programs. This does not apply to programs in load module format. For more information, see the *Administration Guide*.

Short Description (Optional)

Displays a brief description of the program definition the user entered. This is always in mixed case, regardless of the SET EDIT case specification.

New Copy on Call?

For a non-Ideal subprogram only, indicates whether to release the subprogram and load a new copy each time the subprogram is called in batch. When a new copy is loaded, the local data in the non-Ideal subprogram is reset to initial values on each call.

Enter Y for a new copy or N for no new copy. The default is N (do not release the program after each call). If you change this option, you must recompile all calling programs before they can run. This line is omitted for CA Ideal programs.

Update DB|Access DB2 (Non-Ideal subprograms only)

Indicates whether the subprogram performs database additions, changes, or deletions for CA Datacom/DB or any SQL access for DB2. Enter Y or N.

CA Ideal uses this information for CHECKPOINT/ROLLBACK processing and, for DB2, for dynamic plan allocation processing.

Language

Displays the language used in the program.

CA Ideal is the default and is used when the program is written in CA Ideal/PDL. The following designations are used:

ASM

Specifies Assembler programs.

PLI

Specifies PL/1 programs.

COBOL

Specifies COBOL programs.

You must identify a non-Ideal program to CA Ideal with a CA Ideal fill-in. If it is intended to receive parameters, it must have a CA Ideal parameter definition. However, all other coding and maintenance of a non-Ideal program must be performed outside of CA Ideal.

Target Date

Displays the date planned for the application to complete.

Actual Date

Displays the date when the application is actually completed.

Description

(Optional) Displays full description of the program definition, up to five lines long.

Defining Program Resources

The program resources fill-in specifies the resources the application uses. These resources can include dataviews, panels, reports, and subprograms.

When a CA Ideal user enters the Resource Editor, the related entities from Datadictionary are copied into a VLS member. At the end of the edit, Datadictionary is updated and the VLS member is deleted.

If there are errors, the CA Ideal user is notified and cannot leave edit until he fixes the errors or presses the Clear key to abandon the change completely.

To access the resources fill-in for the current program definition, issue the RESOURCES (RES) command.

After resources are specified, the next time the fill-in displays, the entries are ordered alphabetically by entity type. In each entity type, the entries are ordered alphanumerically by name.

The information that identifies program resources is stored in the Datadictionary. CA Ideal uses it for reports and to ensure that the program uses only resources for which it is authorized.

The resources of a program do not have to exist when they are entered in the resource fill-in. When these resources eventually are created, they automatically are associated with all programs that have named them as resources.

After the production version of a CA Ideal subprogram is specified as a resource of a program, the program always uses the production version of that subprogram-even if a different version becomes the production version.

You can use a test version of a subprogram in place of a production subprogram when running a production application by using the ASSIGN PROGRAM VERSION command. For more information about ASSIGN PROGRAM VERSION command, see the *Programming Reference Guide*.

- The definition of a non-Ideal subprogram describes the number of parameters the CA Ideal calling program passes and their structure and data types. This information becomes bound into the CA Ideal calling program during a compilation. If a new version of the subprogram definition is marked to PROD, existing PROD CA Ideal programs are not affected. They still call the non-Ideal program with the old parameter list format. To change a CA Ideal program to use the new parameter list format, duplicate it to a new TEST version, change the RESOURCE table to indicate the correct subprogram version, compile, and mark PROD.
- You can access additional lines for up to 99 entries for each type of resource by scrolling forward.
- You can use the CHECKPOINT and ROLLBACK commands during your editing session to set and return to a stable point in the definition of the program resources.

The following screen shows the program resources fill-in. A completed fill-in follows the description of the individual fields.

```
 =>
--------------------------------------------------------------------------------
 IDEAL: Resource Edit Panel   PGM TEST (001) TEST           SYS: DOC   FILL-IN
 Command Type    Name of DVW/PGM/PNL or RPT         Version  System  Qual?
 ======  ===     ========== T O P ==============     ====     ===     =
 ......
 ......
 ......
 ......
 ......
 ......
 ......
 ......
 ======  ===     ======= B O T T O M ===========     ====     ===     =
```

The components of the program resources fill-in are:

Type

Displays the type of resource. Valid entries are DVW, PNL, PGM, and RPT.

Name

Displays the 1- to 32-character name of a dataview or the one- to eight-character name of a program, panel, or report. If the resource is a dataview, you can specify the authorization ID with the object name, for example, IDEALDS.CUSTOMER.

Version

Identifies the one- to four-character version of the resource. You can specify versions as follows:

PROD

Specifies the production version of the resource.

nnn

Specifies the version number assigned to the resource when it was defined

Tnnn

Specifies the test status version number assigned to the dataview in the Datadictionary for modeled dataviews in TEST status.

If version is not specified for a dataview, the current setting of the SET DATAVIEW VERSION command determines the version used when the program is compiled.

System

The one- to three-character system ID for the system where the panel, report, or program belongs. If the system is not entered, the current system (the system of the program being defined) is used.

Qual (for SQL Dataviews)

Y (Default)-Qualify the SQL object name associated with this dataview using the authorization ID specified in the Name field. The object name is qualified in:

- All SQL statements generated by FOR constructs for this dataview
- Embedded SQL statements specifying the object with an unqualified name

N-Generate the SQL object name without qualification. This option lets you access an SQL object other than the one used to catalog the dataview by giving both objects the same table and view name, but different qualifiers. The tables should have the same structure.

SQL objects can be explicitly qualified in embedded SQL. An explicit qualifier is used even if N is specified here. An ASSIGN AUTHORIZATION command can override the authorization ID specified here.

For more information about embedded SQL and the ASSIGN AUTHORIZATION command, see the *Programming Reference Guide*.

```
  =>
--------------------------------------------------------------------------------
IDEAL: Resource Edit Panel   PGM TEST (001) TEST           SYS: DOC   FILL-IN
Command Type    Name of DVW/PGM/PNL or RPT         Version  System  Qual?
======  ===     ========== T O P ==============     ====     ===     =
......  DVW     IDEALDS.CUSTOMER                                    Y
......  PNL     CUSTPNL                            001      CUS
......  PGM     STDERROR                           PROD     $ID
......  RPT     CUSTLIST                           002
======  ===     ======= B O T T O M ===========     ====     ===     =
```

Displaying Program Resource Indexes

The DISPLAY INDEX PROGRAM command or equivalent DISPLAY INDEX prompter displays the names and status of all occurrences of program definitions in the current system. To display an index of program resources, use the RELATED TO clause of the DISPLAY INDEX command, as follows:

- To display the names and status of all program definitions related to a specified program, enter the command DISPLAY INDEX PROGRAM *name* RELATED TO PROGRAM command. The resulting display shows all programs called or initiated by the specified program and all programs that call the specified program.
- To display the names and status of all dataview definitions related to programs in the current system, enter the DISPLAY INDEX DATAVIEW RELATED TO PROGRAM command.
- To display the names and status of all panel definitions related to programs in the current system, enter the DISPLAY INDEX PANEL RELATED TO PROGRAM command.
- To display the names and status of all report definitions related to programs in the current system, enter the DISPLAY INDEX REPORT RELATED TO PROGRAM command.

For the complete syntax of the DISPLAY INDEX command, see the *Command Reference Guide*.

Displaying the Procedure Definition Panel

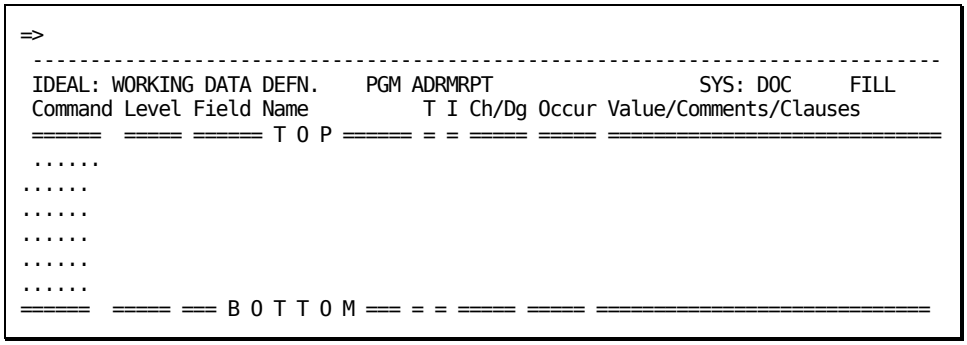
The PROCEDURE command or equivalent PF key displays the program procedure for the current program definition. If you enter this command before the procedure is defined, a blank screen appears that is ready for PDL statements. If you enter this command after a procedure is defined, then as many lines of the procedure (from the top) as fit in the region display.

Defining Working Data

The WORK command or equivalent function key displays the working data definition fill-in for the current program definition.

The working data definition fill-in names and describes data local to the application.

Working data is defined and maintained using the following fill-in.



The fields on the working data definition fill-in are:

Command

Designates an area where you can specify line commands.

Level

Specifies the number that hierarchically ranks fields. Elementary field names must be unique to the highest level (level-1) group name. Simple fields not in a group (level-1 names by themselves) must be unique to the program, and do not require a level number.

Field Name

Contains one of the following:

- A valid name for an elementary item (numeric field, date field, alphanumeric fixed or variable length field, or flag) or for a group item (including a copied dataview or SQLCA).

  You can continue the field name on a second line by including a semicolon (;) as the last character on the first line. You can break the field name at any character. You can specify the level number and the attributes T, I, Ch/Dg, and Occur on the first line only.

- A condition name for a condition name definition. Conditions can be continued.
- A valid subscript of the form (n), (n,o), or (n,o,p) for an initial value of a specific occurring item.
- A filler, an unnamed field of nulls, or blanks to reserve space. The other entries (for example, Level, Type, CH/DG, and so on) are blank.

This entry is blank when the Value/Comments/Clauses column contains a continuation line or a comment or when the rest of the line is blank.

T (Type)

Specifies the field type. Type defaults to N if the internal representation is specified. Type must be blank for a group item, continuation lines, and subscript initial values. Any types that can hold null values must be defined as nullable, which is identified with the keywords WITH IND in the Value/... column. The following are the valid entries:

X

Specifies an alphanumeric field. The value of the field can be any alphabetic, numeric, or special character, or the null value.

V

Specifies a variable length alphanumeric field. The value of the field can be any alphabetic, numeric, or special character or the null value.

N

Specifies a signed numeric field. The value of the field can be 0-9, a minus sign, or decimal symbol, or the null value.

U

Specifies a unsigned numeric field. The value of the field can be 0-9 or decimal symbol, or the null value.

D

Specifies a date field. This type is a numeric field containing an integer number, reflecting the number of days, plus or minus, from December 31, 1900 (day zero), or the null value.

C

Specifies a condition name assigned to a specific value of a field.

F

Specifies a flag that signifies a condition. The only valid values for a flag are TRUE and FALSE.

I (Internal Representation)

Specifies the internal representation of numeric (signed and unsigned) and date type fields:

P

Specifies a packed decimal field.

Z

Specifies a zoned decimal field.

B

Specifies a binary field.

Note: Internal representation must be blank unless the type is N, U, or D. P is the default.

Ch/Dg (Characters/Digits)

Specifies the length of the field value. Either the number of alphanumeric characters or the number of integers, a period, and the number of decimal places in a numeric or date field value. For a variable length alphanumeric field, the maximum number of alphanumeric characters. Date fields cannot have decimal places.

You must specify the characters and digits for all elementary field types except dates and flags. The default for date fields is 7. The minimum length for date fields is 5. For numeric and date fields, the maximum is 31 for packed and zoned, and 9 for binary.

A non-date entry with no type or length information is assumed to be a group name and must have subordinate fields following it.

For example, in the following table, 42 in the Ch/Dg column for the first alphanumeric field (Type X) indicates a length of 42 characters. The value 16 for the variable length field indicates a maximum length of 16 characters. A numeric field (Type N) with 7 specified in the Ch/Dg column indicates a seven-digit field with 7 integer positions. The next numeric field, with 10.3 specified in the Ch/Dg column, indicates a 13-digit field with 10 integer positions and 3 decimal places. A date field (Type D) with 5 in the Ch/Dg column can hold an internal five-integer date value of up to 273 years from the base year.

Type	Ch/Dg
X	42
V	16
N	7
N	10.3
D	5

Occur (Number of Occurrences)

Specifies the number of times a group or field occurs.

When a group or field repeats a fixed number of times, enter the number of occurrences in this column.

Occurring fields (elementary items) can optionally have initial values specified in the Value column. Enter these values on each successive line in the Value column. Enter a valid subscript number (of the form (*n*), (*n,o*), or (*n,o,p*)) on the corresponding line in the Field Name column.

You cannot specify initial values for repeating groups. However, specify initial values for individual occurrences by placing the subscript in Field Name.

A group or field can also occur a variable number of times that depends on the value of another field known as the Depending on field. In this case, the Occur column contains the maximum number of occurrences. The Depending on field name is specified in the Value/... column by using the phrase DEP ON *field-name*. Only one group or field is permitted in any level-01 group. It must be at the end of the level-01 group.

Value/Comments/ Clauses

Specifies one of the following: A value for the occurrence of the field, a REDEFINITION or REDEF keyword, a DEP ON clause, a WITH INDICATOR or WITH IND clause, a COPY DATAVIEW or COPY DVW clause, or a COPY SQLCA clause. You can specify a descriptive comment about the field alone or with any of the others.

Note: The case of the data entered in this column as CA Ideal retains it depends on the setting that was determined for case by default or with a SET EDIT CASE command. Keywords and identifier references must be in uppercase for compiling and editing.

Value

Specifies a specific value assigned to an occurrence of a field.

You can assign an elementary field a value in the Value/... column. This value can be a numeric or alphanumeric literal, depending on the type of the elementary field.

You must enclose alphanumeric literals in delimiters (“ or '). If an alphanumeric literal is longer than the space provided, continue the alphanumeric literal on the next line of the same column surrounded by a new pair of delimiters. Leave all other columns on the continuation line blank. Two or more delimited alphanumeric literals continued in this fashion are concatenated and treated as one.

The default value of a nullable field defined as WITH IND is null, regardless of type. Otherwise, the default value for a numeric field is zero. For a fixed alphanumeric field, it is spaces. For a variable length alphanumeric field, it is the empty string of zero length.

In the case of an occurring field, the Value column is left blank on the line that contains the field name. The following lines can contain occurrence subscripts.

You can assign a variable length field with a value of an alphanumeric literal enclosed in delimiters. The length of this literal is the initial length of the field. It must not exceed the maximum length specified in the Ch/Dg column.

A *flag* can have an initial value of either TRUE (or T) or FALSE (or F). The default is FALSE.

You cannot assign an initial value to a *date field*. The initial value is always zero.

If you omit the length for a character field, but provide a literal value, CA Ideal will set the field length to the length of the literal.

A value of CRLF (without quotes) represents the carriage return/line feed combination used in HTML and XML documents.

If you omit the length for a character field, but provide a literal value, CA Ideal will set the field length to the length of the literal.

A value of CRLF (without quotes) represents the carriage return/line feed combination used in HTML and XML documents.

Comments

Contains useful information about the field. A preceding colon (:) indicates a comment in this column. For more information about how to specify and use comments, refer to the *Programming Reference Guide*.

To continue a comment over multiple lines, start each line of the column with a colon. The other columns can be blank or the continuation of a field name.

REDEF

The keyword REDEFINITION (or REDEF) in this column indicates that this working data item is another view of the closest previously defined item at the same level that is not itself a redefinition.

This item cannot be larger than the item that defines it. The two items can be different types (such as alphanumeric and numeric), but neither item can be a variable length field or a nullable field. Neither item can be a group containing a variable length field or a nullable field.

CA Ideal determines the data type of the REDEF working data item by using the Type specified for the item it redefines. Types explicitly specified with the REDEF working data item are ignored. If the REDEF data item is a group, the data types in its subordinate field do not affect whether the group is alpha or non-alpha. Rather, the item that it redefines determines whether it is an alpha or non-alpha group.

An item with a REDEF cannot have an initial value. If a group has a REDEF, none of its subordinate fields can have initial values. However, the item that defines it can have an initial value.

DEP ON

Designates a field as a counter to limit the number of occurrences of a field that was defined as occurring a variable number of times. *Field-name* must be an elementary numeric field that appears previously in the same level-1 structure. The field must be defined with zero decimal places and cannot be specified with an Occur value. It cannot be a date field. You can specify an initial value for the field.

The keywords DEP and ON can be split over two consecutive lines without a continuation character. To continue a field name onto a second line, specify a semicolon (;) as the last character of the first line. You can break the field name at any character. Leading blanks on the continuation are stripped.

COPY DATAVIEW

Automatically copies the entire structure of a dataview into a working data definition.

- Enter a level-1 name in the Field Name column.
- Enter COPY DATAVIEW *dvwname* in the Value/Comments/Clauses column.
- The keywords COPY and DATAVIEW can be split from the dataview name over two consecutive lines without a continuation character.

To continue a dataview name onto a second line, specify a semicolon (;) as the last character of the first line. You can break the name at any character.

References to field names in a copied dataview must be qualified with the level-1 data name, even if they are the only references to the names. This distinguishes the copied fields from the dataview fields.

CA Ideal performs the copy when the program is compiled. To see the dataview structure, enter the DISPLAY DATAVIEW command or turn the EXD compiler option on. The dataview structure is never expanded in working data.

COPY SQLCA (SQL access only)

Automatically copies the entire structure of the SQLCA work area into a working data definition. The SQLCA contains information about the last SQL statement processed by this program. COPY SQLCA is not needed when using $SQL functions. To use this clause:

- Enter a level-1 data name in the Field Name column.
- Enter COPY SQLCA in the Value/Comments/Clauses column. You can split the keywords COPY and SQLCA over consecutive lines without a continuation character.

You can define only one SQLCA group in working data for each database management system that can be accessed using SQL. The level-1 data name cannot be DB-SQLCA, the name CA Ideal uses. The subordinate fields of each group are the SQLCA fields.

For a list of the SQLCA fields, refer to the $SQL functions in the Programming Reference Guide and to the SQL reference guide for the appropriate database management system.

If the resource table includes an SQL dataview or the EXD compiler listing option is turned on, the SQLCA structure is listed following the WOR/PAR sections in the compiler listing.

CA Ideal performs the copy when the program is compiled.

If you have both CA Datacom SQL and DB2 SQL:

- You can define two SQLCA groups, one for each type of database.
- You can specify which SQLCA to copy by indicating the type of database with the COPY clause. That is:

  ```
  COPY DB SQLCA or COPY DB2 SQLCA
  ```

- If you do not specify a database type, the COPY defaults to the current primary database as defined in the program environment fill-in.

WITH INDICATOR

Indicates (WITH INDICATOR or WITH IND, or IND) that this elementary field is nullable; that is, that it can receive null values. You can specify this clause with alphanumeric, variable length, numeric, date fields, and condition names. You can also specify an initial value but, if you do not, the null value is the default.

NULL

You can specify a nullable field with an initial null value with the keyword NULL in this column. In this case, WITH INDICATOR is optional.

Example

The following example shows the various types of data structures that you can define using the working data definition fill-in. The components of this example apply equally to the record structures of a CA Datacom/DB database as defined through the dictionary. Dataview definitions from the dictionary, as the CA Ideal user sees them, closely resemble the PDL working data structure, as does the definition of parameter data.

The example uses a working data definition fill-in to show how numeric and alphanumeric fields, groups, alpha-groups, and repeating groups are specified.

The following two screens show sample definitions of a flag, an alphanumeric group, and a non-alphanumeric group. ADDRESS, in the second fill-in, is the name of a non-alpha group.

```
  =>
--------------------------------------------------------------------------------
 IDEAL: WORKING DATA DEFN.    PGM JEPDX1  (001) TEST       SYS: DOC   DISPLAY
Command Level Field Name          T I Ch/Dg Occur Value/Comments/Clauses
 ------  ----- ------------------ - - ----- ----- ----------------------------
 ======  ===== ====== T O P ====== = = ===== ===== ============================
 000100  1     PART-NAME          X      16       'SAMPLE-PART-NAME' : DEFAULT
 000200                                           : PART NAME
 000300  1     PART-NUMBER        N       9       999999999 : DEFAULT PART NUM
 000400  1     IN-STOCK           F               FALSE : PART NOT FOUND
 ======  ===== === B O T T O M === = = ===== ===== ============================
```

```
=>
--------------------------------------------------------------------------------
 IDEAL: WORKING DATA DEFN.    PGM JEPDX3  (001) TEST         SYS: DOC  DISPLAY
 Command Level Field Name          T I Ch/Dg Occur Value/Comments/Clauses
 ------  ----- ------------------ - - ----- ----- ----------------------------
 ======  ===== ====== T O P ====== = = ===== ================================
 000100  1     EMPLOYEE-NAME                     : THIS GROUP NAME CAN BE USED
 000200                                          : WHEN FULL NAME IS NEEDED
 000300   2    LAST-NAME          X      15       : USE FOR LAST NAME ONLY
 000400   2    MIDDLE-INITIAL     X       1
 000500   2    FIRST-NAME         X      10       : USE FOR FIRST NAME ONLY
 000600                                          :
 000700  1     ADDRESS                           : THIS GROUP NAME CAN BE USED
 000800                                          : WHEN FULL ADDRESS IS NEEDED
 000900   2    STREET             X      20       : STREET NUMBER AND NAME
 001000   2    CITY               X      15
 001100   2    STATE              X       2       : 2-CHARACTER ABBREVIATION
 001200   2    ZIP-CODE           N       9       : SPACE FOR NEW 9-DIGIT CODE
 ======  ===== === B O T T O M === = = ===== ===== ============================
```

The next screen shows an example of a repeating field with initial values in the form of a table of months:

```
  =>
------------------------------------------------------------------------------
IDEAL: WORKING DATA DEFN.    PGM JEPDX2  (001) TEST          SYS: DOC  DISPLAY
Command Level Field Name          T I Ch/Dg Occur Value/Comments/Clauses
------  ----- ------------------- - - ----- ----- ----------------------------
======  ===== ====== T O P ====== = = ===== ===== ============================
000100  1     MONTH-TABLE         X       9    12
000200        (1)                                  'JANUARY'
000300        (2)                                  'FEBRUARY'
000400        (3)                                  'MARCH'
000500        (4)                                  'APRIL'
000600        (5)                                  'MAY'
000700        (6)                                  'JUNE'
000800        (7)                                  'JULY'
000900        (8)                                  'AUGUST'
001000        (9)                                  'SEPTEMBER'
001100        (10)                                 'OCTOBER'
001200        (11)                                 'NOVEMBER'
001300        (12)                                 'DECEMBER'
======  ===== === B O T T O M === = = ===== ================================
```

The next screen shows how to define several data types:

- Alphanumeric (LAST-NAME, MIDDLE-INITIAL, and so on).
- Signed numeric in packed internal format (EMPLOYEE-NUMBER and #DEPENDENTS) and unsigned numeric in zoned internal format (ZIP-CODE).
- Variable length (COM-ADDRESS) that holds a complete address.

It also shows several types of data structures:

- A repeating group (DEPENDENT) that derives its number of occurrences from another field (#DEPENDENTS).
- A field (MARITAL-STATUS) for which a series of condition names was specified.
- Two examples of alpha groups (EMPLOYEE-NAME and FULL-NAME).
- A non-alpha group (ADDRESS). This group includes an alphanumeric field (ZIP-CODE-ALT) that redefines a numeric field (ZIP-CODE). If the order of these two fields were reversed (if ZIP-CODE redefined ZIP-CODE-ALT), ADDRESS would then be an alpha group since the type of the field with the Redefines does not affect the group type.

```
  =>
--------------------------------------------------------------------------------
 IDEAL: WORKING DATA DEFN.    PGM JEPDX4  (001) TEST          SYS: DOC  DISPLAY
Command Level Field Name          T I Ch/Dg Occur Value/Comments/Clauses
 ------  ----- ------------------ - - ----- ----- ----------------------------
 ======  ===== ====== T O P ====== = = ===== ===== ============================
 000100  1     EMPLOYEE-NAME
 000200
 000300   2    LAST-NAME          X      15
 000400   2    MIDDLE-INITIAL     X       1
 000500   2    FIRST-NAME         X      10
 000600
 000601  1     EMPLOYEE-NUMBER    N P     7
 000602
 000700  1     ADDRESS
 000800
 000900   2    STREET             X     20
 001000   2    CITY               X     15
 001100   2    STATE              X      2
 001200   2    ZIP-CODE           U Z    9
 001201   2    ZIP-CODE-ALT       X      9        REDEF
 001202
 001203  1     COM-ADDRESS        V     46
 001300
 001400  1     MARITAL-STATUS     X      1
 001500        MARRIED            C               'M'
 001600        SINGLE             C               'S'
 001700        DIVORCED           C               'D'
 001800        WIDOWED            C               'W'
 001900        SEPARATED          C               'E'
 002000
 002100  1     DEPENDENTS
 002200   2    #DEPENDENTS        N P    2        : NUMBER OF DEPENDENTS
 002300   2    DEPENDENT                      12 DEP ON #DEPENDENTS
 002400    3   FULL-NAME
 002500     4  GIVEN              X     10        : DEPENDENT'S FIRST NAME
 002600     4  MIDDLE             X      1
 002700     4  LAST               X     15        : IF DIFFERENT FROM EMPLOYEE
 002800    3   BIRTH-DATE         D      5
 ======  ===== === B O T T O M === = = ===== ===== ============================
```

The following screen defines a working data field used as a report summary line. The multiple consecutive lines of values are concatenated into a single alphanumeric literal.

```
  =>


--------------------------------------------------------------------------------
 IDEAL: WORKING DATA DEFN.    PGM JEPDX6  (001) TEST          SYS: DOC  DISPLAY
Command Level Field Name          T I Ch/Dg Occur Value/Comments/Clauses
 ------  ----- ------------------ - - ----- ----- ----------------------------
 ======  ===== ====== T O P ====== = = ===== ===== ============================
 000100  1     RPT-LITERAL        X      44       '**** This Is the Last Line '
 000200                                           'of the Report ****'
 ======  ===== === B O T T O M === = = ===== ===== ============================
```

The following screen shows the definition of a two-dimensional array (ROWS,RANKS) representing the squares on a chess board. The initial values show the deployment of pieces at the start of a game.

```
  =>

------------------------------------------------------------------------------
 IDEAL: WORKING DATA DEFN.    PGM JEPDX5  (001) TEST          SYS: DOC  DISPLAY
Command Level Field Name          T I Ch/Dg Occur Value/Comments/Clauses
 ------  ----- ------------------ - - ----- ----- ----------------------------
 ======  ===== ====== T O P ====== = = ===== ===== ============================
 000100  1     CHESS-BOARD
 000200   2    ROW                                8
 000300    3   RANK               X       9     8
 000400        (1,1)                               'WH-ROOK'
 000500        (1,2)                               'WH-KNIGHT'
 000600        (1,3)                               'WH-BISHOP'
 000700        (1,4)                               'WH-KING'
 000800        (1,5)                               'WH-QUEEN'
 000900        (1,6)                               'WH-BISHOP'
 001000        (1,7)                               'WH-KNIGHT'
 001100        (1,8)                               'WH-ROOK'
 001101        (2,1)                               'WH-PAWN'
 001102        (2,2)                               'WH-PAWN'
 001103        (2,3)                               'WH-PAWN'
 001104        (2,4)                               'WH-PAWN'
 001105        (2,5)                               'WH-PAWN'
 001106        (2,6)                               'WH-PAWN'
 001107        (2,7)                               'WH-PAWN'
 001108        (2,8)                               'WH-PAWN'
 001114        (7,1)                               'BL-PAWN'
 001115        (7,2)                               'BL-PAWN'
 001116        (7,3)                               'BL-PAWN'
 001117        (7,4)                               'BL-PAWN'
 001118        (7,5)                               'BL-PAWN'
 001119        (7,6)                               'BL-PAWN'
 001120        (7,7)                               'BL-PAWN'
 001121        (7,8)                               'BL-PAWN'
 001200        (8,1)                               'BL-ROOK'
 001300        (8,2)                               'BL-KNIGHT'
 001400        (8,3)                               'BL-BISHOP'
 001500        (8,4)                               'BL-KING'
 001600        (8,5)                               'BL-QUEEN'
 001700        (8,6)                               'BL-BISHOP'
 001800        (8,7)                               'BL-KNIGHT'
 001900        (8,8)                               'BL-ROOK'
 ======  ===== === B O T T O M === = = ===== ===== ============================
```

Defining Parameters Used as Input

Parameter data consists of the names and descriptions of data items to pass to this program from a calling program or to this program using a RUN command. Parameter data is specified for the called subprogram on the parameter definition fill-in, shown in the following screen.

```
  =>
--------------------------------------------------------------------------------
 IDEAL: PARAMETER DEFINITION  PGM ADRMRPT  (001) TEST         SYS: DOC  FILL-IN
 Command Level Field Name          T I Ch/Dg Occur  U M  Comments/Dep on/Copy
 ======  ===== ====== T O P ====== = = ===== =====  = = ======================
  ......
 ......
 ......
 ......
  ......
 ......
 ......
======  ===== === B O T T O M === = = ===== ===== ==========================
```

The following pages describe the components of the parameter definition fill-in. Parameter data is specified in much the same way as working data, as you can see from the similarities between the fill-ins. However, the specification of parameter data differs in several ways from the definition of working data:

- Two columns on the parameter definition fill-in are not on the working data definition fill-in: **U** (update intent) and **M** (matching).
- You do not need to specify all attributes for a parameter for a called CA Ideal subprogram with dynamic parameter matching.

Level

Specifies the number that hierarchically ranks fields. Elementary field names must be unique to the highest level (level-1) group name. Elementary fields not in a group (level-1 names by themselves) must be unique to the program. All level-1 names must be unique to the program.

Field Name

A valid field name for a parameter that corresponds to the data item in the CALL statement. This name in the called program can be the same as or a different from the name of the data item in the calling program.

You can continue the field name on a second line by including a semicolon (;) as the last character on the first line. You can break the field name at any character. Leading blank characters on a continued line are stripped.

You can specify the level number and the attributes T, I, Ch/Dg, and Occur on the first line only.

- A condition name.
- A filler, an unnamed field of nulls or blanks that reserves space. The other entries (for example, Level, Type, CH/DG, etc.) are blank.
- Blanks when the Comment/Dep on/Copy column contains a continuation line or when the rest of the line is blank.

T (Type)

Specifies the parameter type.

Note: For non-Ideal subprograms, you must specify type X, N, or U. For a detailed explanation of each field type and its characteristics, see the *Programming* Reference *Guide*.

The types are as follows:

X

Specifies an alphanumeric field. The value of the field can be any alphabetic, numeric, or special character, or the null value.

V

Specifies a variable length alphanumeric field. The value of the field can be any alphabetic, numeric, or special character or the null value.

N

Specifies a signed numeric field. The value of the field can be 0-9, a minus sign, or decimal symbol, or the null value.

U

Specifies a unsigned numeric field. The value of the field can be 0-9 or decimal symbol, or the null value.

D

Specifies a date field. This type is a numeric field containing an integer number, reflecting the number of days, plus or minus, from December 31, 1900 (day zero), or the null value.

C

Specifies a condition name assigned to a specific value of a field.

F

Specifies a flag that signifies a condition. The only valid values for a flag are TRUE and FALSE.

I (Internal Representation)

Specifies the internal representation of numeric (signed and unsigned) and date type fields:

P

Specifies a packed decimal field.

Z

Specifies a zoned decimal field.

B

Specifies a binary field.

Note: Internal representation must be blank unless the type is N, U, or D. P is the default.

For CA Ideal subprograms, use the following:

- With dynamic parameter matching, the internal representation must be blank.
- With identical parameter matching, the internal representation is required for D, N, and U type fields.

For non-Ideal subprograms, the internal representation is required for N and U type fields.

Ch/Dg (Characters/Digits)

Specifies the length of the field value. Either the number of alphanumeric characters or the number of integers, a period, and the number of decimal places in a numeric or date field value For a variable length alphanumeric field, the maximum number of alphanumeric characters. Date fields cannot have decimal places.

You must specify the characters and digits for all elementary field types except dates and flags. The default for date fields is 7. The minimum length for date fields is 5. For numeric and date fields, the maximum is 31 for packed and zoned, and 9 for binary.

A non-date entry with no type or length information is assumed to be a group name and must have subordinate fields following it.

For example, in the following table, 42 in the Ch/Dg column for the first alphanumeric field (Type X) indicates a length of 42 characters. The value 16 for the variable length field indicates a maximum length of 16 characters. A numeric field (Type N) with 7 specified in the Ch/Dg column indicates a seven-digit field with 7 integer positions. The next numeric field, with 10.3 specified in the Ch/Dg column, indicates a 13-digit field with 10 integer positions and 3 decimal places. A date field (Type D) with 5 in the Ch/Dg column can hold an internal five-integer date value of up to 273 years from the base year.

Type	Ch/Dg
X	42
V	16
N	7
N	10.3
D	5

For CA Ideal subprograms with dynamic parameter matching, this specification is required for type X and V parameters and optional for types N, U, and D.

For type X parameters and for type V parameters passed with dynamic parameter matching and Input update intent, the length of the Characters/Digits of the subprogram parameter must be greater than or equal to the Characters/Digits specified in the calling program.

For type V parameters passed with dynamic parameter matching and Update intent, the characters/digits must match the specification in the calling program.

Numeric and date parameters do not require a characters/digits specification, because they are automatically assigned the same logical attributes as the corresponding data item in the calling program.

If you specify characters/digits, it must match the characters/digits of the corresponding data item in the calling program.

With identical parameter matching, the character/digits specification is required for all types. The characters/digits must match the characters/digits of the corresponding data item in the calling program.

For non-Ideal subprograms, the characters/digits attribute is required for all types.

Also see the rules for calling subprograms described in Executing Nested Programs.

Occur (Number of Occurrences)

Specifies the number of times a group or field occurs (or the maximum for dynamic parameters).

When a group or field occur a fixed number of times, the number of occurrences is entered in this column.

A group or field also occur a variable number of times, depending on the count in another field known as the Depending on field. In this case, the Occur column contains the maximum number of occurrences. The Depending on field name is specified in the Comments/Dep on/Copy column by using the phrase DEP ON field name. Only one such variably repeating group is permitted in any structure. It must be at the end of the structure. For dynamic parameters, this is the maximum number of occurrences that you can specify for the corresponding field in the calling program.

Note: You cannot specify initial values for repeating fields in parameter data. You cannot specify an Occur value for a level-1 parameter.

U (Update Intent)

Specifies the kind of parameter. This attribute is specified only for a level-1 item, never for subordinate fields of a group.

U

Specifies an update parameter, that is, one that the program can modify.

I

Specifies an input parameter, that is, one that the program cannot modify.

M (Parameter Matching)

(For CA Ideal subprograms only.) Specifies whether attributes are dynamically copied into the parameter from the corresponding data item in the calling program and only match the general type or must be completely specified and match exactly the corresponding data item in the calling program. Specify this attribute only for level-1 parameters. It applies only to that field or group.

D (Default)

Specifies dynamic matching. Attributes for this data item are copied from the calling program.

I

Specifies identical matching. Attributes must be completely specified and match those of the calling program data item. Identical parameter matching generates more efficient code since all attributes are known at compile time.

Comments/Dep on/Copy (Comment, Redefinition, Depending on field name, COPY dataview or SQLCA, or WITH INDICATOR)

The case of the text entered in this column as CA Ideal retains it is determined by default or with a SET EDIT CASE command. You can specify a descriptive comment about the field alone or with any of the others. The keywords and field names must be uppercase.

Comments

Specifies useful information about the field. A comment is indicated in this column by a preceding colon (:).

To continue a comment over multiple lines, start each line of the column with a colon. The other columns can be blank or the continuation of a field name.

REDEF

Indicates with the keyword REDEFINITION (or REDEF) in this column that this parameter item is another view of the closest previously defined item at the same level that is not itself a redefinition.

This item cannot be larger than the item that defined it. The two items can be different types (such as alphanumeric and numeric), but neither item can be a variable length field or a nullable field. None of the items can be a group containing neither a variable length field nor a nullable field.

If a parameter field with a REDEF is a subordinate field, its Type does not affect whether the group is alpha or non-alpha, nor types of the subordinate fields in the group. CA Ideal uses the type of the item that defined it. For an example, refer to the definition of alpha groups in the *Programming Reference Guide*.

Redefinitions are not allowed for level-1 parameters and are not allowed for non-Ideal subprograms.

DEP ON

Specifies a DEP ON field-name clause. This clause designates a field as a counter to limit the number of occurrences of a field that was defined as occurring a variable number of times. Field name must be an elementary numeric field that appears previously in the same level-1 structure. The field-name field must be defined with zero decimal places and cannot be specified with an Occur value. It cannot be a date field. You can specify an initial value for the field-name field.

You can split the keywords DEP and ON over two consecutive lines without a continuation character. To continue a field name onto a second line, specify a semicolon (;) as the last character of the first line. You can break the field name at any character. The other columns can be blank or the continuation of a field name.

COPY DATAVIEW

Specifies to automatically copy the entire structure of a dataview into a parameter definition. To use this clause:

- Enter a level-1 name in the Field Name column.
- Enter COPY DATAVIEW *dvwname* in the Comments/Dep On/Copy column.
- To continue a dataview name onto a second line, specify a semicolon (;) as the last character of the first line. You can break the name at any character.

The dataview being copied must be a cataloged dataview and must be specified as a resource of the program.

CA Ideal performs the copy when the program is compiled. To see the dataview structure, enter the DISPLAY DATAVIEW command. The structure is not shown in the parameter data. This is included in the compile listing if the EXD compiler option is on.

Note: You must include the dataview in the resources of the program even though there might not be any FOR construct in the procedure.

COPY SQLCA (SQL only)

Automatically copies the entire structure of the SQLCA work area into the parameter definition. The SQLCA contains information about the last SQL statement this program processed. COPY SQLCA is not needed if you use $SQL functions. To use this clause:

- Enter a level-1 data name in the Field Name column.
- Enter COPY SQLCA in the Comments/Dep On/Copy column. You can split the keywords COPY and SQLCA over consecutive lines without a continuation character.

You can define only one SQLCA group in parameter data for each database management system that SQL can access. The level-1 data name cannot be DB-SQLCA, the name CA Ideal uses. The subordinate fields of each group are the SQLCA fields.

For a list of the SQLCA fields, refer to the $SQL functions in the *Programming Reference Guide* and to the SQL reference guide for the appropriate database management system.

If the resource table includes an SQL dataview, the SQLCA structure is listed following the WOR/PAR sections in the compiler listing.

If the EXD compiler listing option is turned on, the SQLCA structure is listed following the WOR/PAR sections.

CA Ideal performs the copy when the program is compiled.

If you have both CA Datacom SQL and DB2 SQL, you can define two SQLCA groups, one for each type of database.

You can specify which SQLCA to copy by indicating the type of database with the COPY clause. That is COPY DB SQLCA or COPY DB2 SQLCA.

If you do not specify a database type, the COPY defaults to the current primary DBMS defined in the program environment fill-in.

WITH INDICATOR

Specifies with the clause WITH INDICATOR (or WITH IND, or IND) in this column that this field is nullable; that is, it can receive null values. You can specify this clause with alphanumeric, variable length, numeric, date fields, and with condition names.

You must define a parameter field as nullable if the corresponding field in the calling program is nullable, regardless of its value or matching selection.

The following screen shows how you can use these parameters:

- A numeric field (YTD-WAGES) defined as updateable with dynamic parameter matching.
- A non-alpha group (ADDRESS) with identical parameter matching and all attributes specified. You can use this group in group moves and all other non-alpha group capacities.
- An alphanumeric parameter (ZIP-ALPHA) in this group redefines an unsigned numeric field (ZIP).

  If the order of these two fields were reversed (if ZIP redefined ZIP-ALPHA), ADDRESS would then be an alpha group since the type of the field with the Redefines does not affect the group type.

The following screen also shows these forms of parameter definition:

- A date field (HIRE-DATE) with packed internal type.
- A variable length field (NAME) with a maximum size of 20 characters, the update intent is input, and identical parameter matching.
- Two signed numeric fields with identical parameter matching:
  - NUMBER with packed internal type
  - WAGES with zoned internal type

```
  =>
--------------------------------------------------------------------------------
 IDEAL: PARAMETER DEFINITION  PGM JEUPPER  (001) TEST         SYS: DOC  DISPLAY
Command Level Field Name          T I Ch/Dg Occur  U M  Comments/Dep on/Copy
 ------  ----- ------------------ - - ----- -----  - -  ----------------------
 ======  ===== ====== T O P ====== = = ===== =====  = =  ======================
 000100  1     HIRE-DATE          D P     6        I I
 000200  1     NAME               V      20        I I
 000300  1     NUMBER             N P     7        I I
 000400  1     WAGES              N Z  4.2         U I
 000500  1     YTD-WAGES          N                U D
 000600  1     ADDRESS                             I I
 000700   2    STREET             X      20
 000800   2    CITY               X      20
 000900   2    STATE              X       2
 001000   2    ZIP                U Z     9
 001100   2    ZIP-ALPHA          X       9            REDEF
 ======  ===== === B O T T O M === = = ===== =====  = =  ======================
```

In addition to the parameter definition fill-in, CA Ideal subprograms permit dynamic and identical parameter matching, while enforcing certain linkage conventions. See Calling a Subprogram section for an explanation of these processes.

Defining the Environment for SQL Access

Environment options are needed for any program that accesses CA Datacom/DB using SQL. They include SQL access plan options and the database for this program.

CA Datacom SQL Access Plan Options

CA Ideal uses the CA Datacom SQL access plan options to build the CA Datacom/DB access plan for the program during program compilation. These plan options are described in the *CA Datacom/DB Database SQL User Guide*.

For mixed sites, the CA Datacom/DB access plan options do not affect DB2 application plans.

The plan options are specified on the environment definition fill-in. You can use the SET DBSQL command to specify default values for the fill-in.

To display or edit the plan options, select option 1 from the Program Maintenance menu. Fill in field 1 and field 2 as appropriate. Specify *env* for field 3 and press the Enter key. The plan options fill-in displays.

Primary Database

For sites using both CA Datacom SQL and DB2 SQL, the primary database is the database management system SQL statements access, such as GRANT and REVOKE, which might apply to either database management system. These SQL statements do not reference an object associated with a dataview.

SQL DML statements such as INSERT do not use the primary database. They must reference an object associated with a dataview that specifies the database management system. You can have INSERTs against multiple SQL database management systems in one program, but you cannot mix dataviews for different database systems in one statement.

The SQL statements COMMIT and ROLLBACK apply to all databases that the program accesses. The WHENEVER statement applies to all SQL database management systems.

On the other hand, GRANT, REVOKE, and all similar SQL statements can access only one database management system per program-the primary database.

The primary database is specified on the environment definition fill-in. Use the SET ENVIRONMENT SQL command to specify a default value.

To access the environment definition fill-in, use the ENVIRONMENT option with the DISPLAY or EDIT command or, while the program is displayed, use the ENVIRONMENT command.

```
  =>
--------------------------------------------------------------------------------
 IDEAL: PGM ENVIRONMENT       PGM TEST (001) TEST            SYS: DOC   FILL-IN

 Primary SQL database:           DB        (DB or DB2)
 DB/SQL options
     Default Auth-id............ X
     SQL Mode................... ANSI86   (ANSI86, DATACOM, or FIPS)
     Cursor Isolation Level..... C        (U=User, C=Cursor, R=Repeatable)
     Optimization Mode.......... M        (M=Manual,P=Preptime,E=Exectime)
     Date Format................ USA      (DB, ISO, USA, EUR, JIS)
     Time Format................ USA      (DB, ISO, USA, EUR, JIS)
     CBSIO...................... 524286   (0-524287)
     Priority................... 15       (1-15)
     Wait Time.................. 030      (1-120)
                                 S        (M=Minutes, S=Seconds)
     Preptime optimization msgs. N        (N=None,D=Detail,S=Summary)
     Exectime optimization msgs. N        (N=None,D=Detail,S=Summary)
     DB/SQL Workspace........... 0000     (0-128)
     Decimal Point.............. P        (P=Period, C=Comma)
     String Delimiter........... A        (A=Apostrophe, Q=Quote)
```

Primary SQL DBMS (Mixed SQL Sites Only)

For sites with both CA Datacom SQL and DB2 SQL, the database to access by SQL statements that does not reference a dataview.

DB

Specifies the CA Datacom SQL.

DB2

Specifies the IBM DB2 SQL.

Default Auth-ID

Specifies the one- to eight-character authorization ID for the program's CA Datacom/DB plan.

SQL Mode

Specifies the mode in which CA Datacom/DB processes the program. The following are valid entries:

- ANSI86
- DATACOM
- FIPS

Cursor Isolation Level

Specifies the degree to which a unit of recovery is isolated from the updating operations of other units of recovery.

U

Specifies no locks are acquired.

C

Specifies cursor stability (required for updates, deletes, or inserts).

R

Specifies repeatable read.

Optimization Mode

Specifies the mode in which CA Datacom/DB optimizes table joins.

P

Specifies preptime, that is, order joins during bind processing (CA Ideal Compile). This is the default.

M

Specifies manual, that is, order joins as specified in FROM clauses.

E

Specifies exectime, that is, order joins at execution time.

Date Format

Specifies the display format for SQL date type items.

DB

Specifies to use the CA Datacom/DB default. In CA Datacom/DB, the date can be set to ISO, USA, EUR, or JIS formats.

ISO

Specifies the International Standards Organization format: *yyyy-mm-dd*.

USA

Specifies the U.S. standard format: *mm/dd/yyyy.*

EUR

Specifies the European standard format: *dd.mm.yyyy.*

JIS9

Specifies the Japanese Industrial Standard format: *yyyy-mm-dd.*

Time Format

Specifies the display format for SQL time type items.

DB

Specifies to use the CA Datacom/DB default. In CA Datacom/DB, the date can be set to ISO, USA, EUR, or JIS formats.

ISO

Specifies the International Standards Organization format: *hh.mm.ss*

USA

Specifies the U.S. standard format: *hh:mm* AM or PM

EUR

Specifies the European standard format: *hh.mm.ss*

JIS

Specifies the Japanese Industrial Standard format: *hh:mm:ss*

CBSIO

Specifies the I/O limit interrupt value for SQL statements that creates a set.

Priority

Specifies the priority of the SQL requests. The lowest priority is 1. The highest priority is 15.

Wait Time

Specifies the exclusive control wait limit. Enter a number from 1 through 120, then indicate on the next line the unit of measure, for example, S for seconds or M for minutes.

Preptime Optimization Msgs

Specifies the type of optimization messages CA Datacom/DB produces during bind processing.

N

Specifies no messages. This is the default.

D

Specifies detailed messages.

S

Specifies a summary of the messages.

Exectime Optimization Msgs

Specifies the type of optimization messages CA Datacom/DB produces at runtime.

N

Specifies no messages. This is the default.

D

Specifies detailed messages.

S

Specifies a summary of the messages.

DB/SQL Workspace

Specifies the amount of work space available at plan execution time used for error correction. Enter a number from 0 through 1024.

Decimal Point

Specifies the character used as the decimal point when data displays. This has no effect on how the data is stored.

P

Specifies to use the period (.) as the decimal point. The comma (,) is used as the digit separator. This is the default.

C

Specifies to use the comma (,) as the decimal point. The period (.) is used as the digit separator.

String Delimiter

Specifies the character that delimits string values in all SQL statements.

A

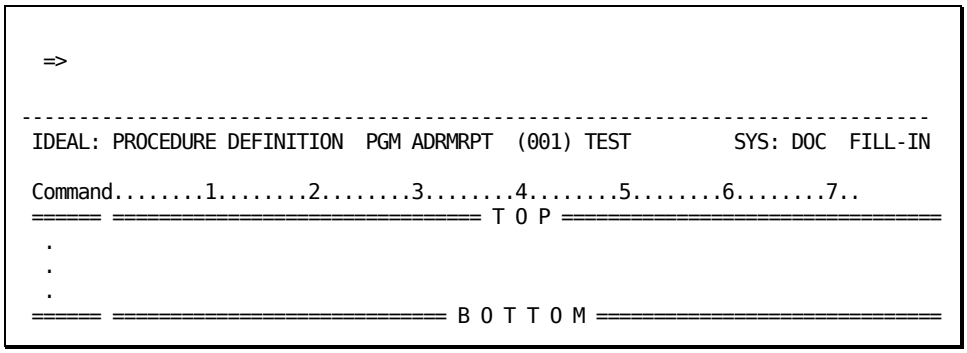
Specifies to use the apostrophe (') as the delimiter. This is the default.

Q

Specifies to use the quotation mark (") as the delimiter.

Entering the Procedure Definition

The PROCEDURE command or equivalent function key displays the program procedure for the current program definition. If you enter this command before you define the procedure, a blank screen appears, ready for PDL statements as shown in the following screen. If you enter this command after you define a procedure, then as many lines of the procedure (from the top) as fit in the region display.



The SET EDIT CASE command establishes the case of the text entered in the procedure fill-in. You must enter reserved words and identifier references in uppercase.

The remaining chapters of this guide explain how to use the CA Ideal Procedure Definition Language to read and write data, control the sequence of processing, perform calculations, use functions, and terminate processing. For more information about the syntax of the PDL statements, see the *Programming Reference Guide*. For information about the editing facilities available for the procedure fill-in, see the *Command Reference Guide*.

Displaying and Editing a Program

There are three ways to display or edit the components that make up a program. Use the display methods when you do not intend to make any changes to the program. Use the edit methods when you want to make changes to the program.

- Enter the following command on the command line to display the component:

  `DISPLAY PROGRAM program-name component-name`

  Enter the following command on the command line to edit the component:

  `EDIT PROGRAM program-name component-name`

  If you do not include a program name or component name, a prompter displays. Enter the missing program name or component name on the prompter to complete the command.

- Select option 1, DISPLAY/EDIT, from the Program Maintenance menu. This displays a prompter. You must specify not only the program name and component name, but also whether you want to display or edit the component.

- From the DISPLAY INDEX PROGRAM line command field, enter the line command DIS next to the program you want to display or enter EDI next to the program you want to edit. The procedure component of the program displays. To edit or display another component, such as RES, enter RES on the command line before pressing Enter.

For more information about these commands, see the *Command Reference Guide*.

Duplicating a Program to a New Name

There are two ways to duplicate a program to a new name. When a program is duplicated, the entire program definition with all of its components is duplicated to the new name or new version.

- Enter the following command on the command line to duplicate the program:

  `DUPLICATE PROGRAM program-name TO NEWNAME new-program-name`

  If you do not include the originating program name or the new program name, a prompter displays. Enter the missing program name on the prompter to complete the command.

- Select option 6, DUPLICATE, from the Program Maintenance menu. This displays a prompter. You must specify the originating program name and new program name.

Deleting a Program

There are three ways to delete a program. When a program is deleted, the entire program definition with all of its components is deleted.

- Enter the following command on the command line to delete the program:

  `DELETE PROGRAM program-name VERSION ver`

  If you do not include the program name and version, a prompter displays. Enter the missing program name on the prompter to complete the command.
- Select option 4, DELETE, from the Program Maintenance menu. This displays a prompter. You must specify the program name.
- From the DISPLAY INDEX PROGRAM line command field, enter the line command DEL next to the program you want to delete.

Printing a Program

The PRINT command or equivalent PRINT prompter prints a specific program definition. There are three ways to print a program. When a program is printed, the entire program definition with all of its components prints.

- Enter the following command on the command line to print the program:

  `PRINT PROGRAM program-name`

  If you do not include the program name, a prompter displays. Enter the missing program name on the prompter to complete the command.
- Select option 3, PRINT, from the Program Maintenance menu. This displays a prompter. You must specify the program name.
- From the DISPLAY INDEX PROGRAM line command field, enter the line command PRI next to the program you want to print.

Listing an Index of Defined Programs

You can display or print an index of defined programs in a CA Ideal system. The list shows the names of all programs in the specified system, the language used to create each program, a short description of each program, the date each program was created, and the date each program was last updated.

Displaying the Index

There are two ways to display the list of programs in a system.

- Enter the following command on the command line to print the list of programs.

  ```
  DISPLAY INDEX PROGRAM
  ```

- Select option 7, DISPLAY INDEX, from the Program Maintenance menu. This displays a prompter. You must type PGM in the field labeled (1) to complete the command.

You can initiate activities against any of the programs listed on the display by entering line commands in the Command field. See the *Command Reference Guide* for complete information about the DISPLAY INDEX command and the line commands that you can use on the display.

Printing the Index

To print the list of programs in a system, enter the following command on the command line:

```
PRINT INDEX PROGRAM
```

Chapter 2: Reading and Writing Data

Introduction to the Database

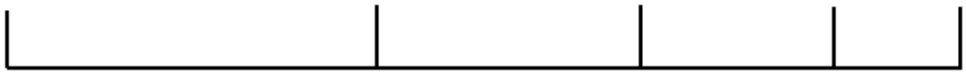
A CA Ideal accesses the relational database of CA Datacom/DB. To understand how CA Ideal handles the access, a basic description of the database terminology is helpful.

Basic Database Structure

The data in the database is organized into a series of logical collections called tables. For example, all customer information can be defined in a customer table; all order information in an order table; and so on. In each table, all of the data associated with an individual occurrence is a row. For example, all information about a single customer comprises one row.

The data in each table is further defined by type and function into columns. Each column is assigned a name and various attributes. For example, the customer name is defined in the column named CUSTNAME and the ZIP code in the column named ZIP. The data in the column CUSTNAME is alphanumeric and the data in ZIP is numeric.

CUSTNAME	ADDRESS	CITY	STATE	ZIP
Brown Jones	52 Green Street 103 Main Street	Toledo Burbank	OH CA	43697 91103



The power of a relational database lies in the fact that data can be accessed without first predetermining any access paths. For example, it is very straightforward to ask for the set of customers whose last name was Brown and lived in Ohio. No additional work other than describing the layout of the table is needed to access it.

Using Dataviews to Access the Database

In CA Ideal, a dataview is a logical view of external data that lets you make requests independent of the external storage mechanism. Dataviews can represent sequential files, VSAM files, CA Datacom/DB tables (for native command access or SQL access), views, and synonyms (for SQL access or DB2 tables and views). Regardless of the underlying table, file, or view, the programmer sees the definition presented in a consistent way and uses the same language statements to access the data.

Before CA Ideal can use a dataview, the table, view, or synonym must be defined in the database management system. Then, in CA Ideal, an authorized user must *catalog* the dataview to CA Ideal. The catalog function locates the database management system's definition and makes it accessible to CA Ideal. Any CA Ideal program can then access the dataview, provided that the program is in a system authorized to access the dataview.

a CA Datacom/DB table is defined using the dictionary facilities of Datadictionary. After the required entries are processed, the DBA (database administrator) creates a dataview entity occurrence in the Datadictionary and relates it to the appropriate database elements. a CA Datacom/DB native access dataview can include any subset of fields in the database record, including the entire record, but the fields can only be included from one record. SQL table or view is created with an SQL CREATE statement in a program and supports joins and projections for views. No further action is required before cataloging the table or view to CA Ideal.

Dataviews for unmodeled sequential files and VSAM files are created in CA Ideal and then cataloged.

When a dataview is included in the resource definition of a CA Ideal program, it serves two functions:

- It documents the fact that this program uses this dataview.
- It records the relationship in the Datadictionary.
- The dataview serves as a copybook for the CA Ideal program. CA Ideal automatically generates a list of host variables in the program, with names identical to the columns or fields in the original table or view. In this sense, the dataview layout defines a buffer to contain the current row.

Using Multiple Dataviews for One Table

Multiple dataviews can be created for a single table to provide flexibility. A program can contain one or more of these dataviews as needed.

In CA Datacom/DB, you can define native dataviews directly from the elements of the table. You can create one or more dataviews to contain any combination of these elements.

In SQL, you can define a dataview based on a table. This dataview contains all of the columns in the table. You can create only one dataview directly from the table. You must use a view to define multiple dataviews for a single table. You can specify the view to contain any of the columns in the table. In fact, views can contain columns from one or more tables. You can use a dataview cataloged on a joined view for read-only purposes.

Additional Information

For more information about the creation and maintenance of dataviews, see the *Creating Dataviews Guide*. The remainder of this chapter focuses on using CA Ideal to access and manipulate the data in the database. It is applicable to databases for CA Datacom/DB and DB2 databases. The programs written based on this general information about the FOR construct are portable between databases.

For more information about accessing sequential and VSAM files, see the FOR construct in the *Programming Reference Guide*.

Accessing Data from a Table or File

A CA Ideal PDL, Program Definition Language, provides the FOR construct to access the data in the database. There are several variations of the FOR construct. With any variation, a dataview is named to specify which data to obtain. An ENDFOR statement terminates the construct.

```
FOR ... dataview-name
   : statements
ENDFOR
```

Selecting and Processing Rows

Basically, the FOR statement selects the rows to process. Each row in the selected set is accessed individually and processed according to the statements specified between the FOR and ENDFOR statements.

The simplest FOR construct, FOR EACH, accesses every row in the table based on the named dataview. For example, the following FOR construct accesses all rows in the CUSTOMER table using the CUSTOMER dataview:

```
FOR EACH CUSTOMER            :select
   : statements              :process
ENDFOR                       :end^sprocess
```

Processing Rows with Implicit Iteration

The FOR EACH construct implicitly iterates. That is, it loops through the selected set, processing one row at a time. For example, the FOR EACH construct can produce a list of customers from the CUSTOMER table, as follows:

```
FOR EACH CUSTOMER            :select
   LIST CUSTNAME             :list customer name
ENDFOR                       :end process
```

This produces:

```
custA
custB
custC
. . .   and so on
```

To include a count with this list, the counter TX-COUNT is defined in working data and then incremented as each row is accessed. The value of the counter is written, along with the customer name, using the LIST statement:

```
SET TX-COUNT = 0
FOR EACH CUSTOMER
   ADD 1 TO TX-COUNT
   LIST TX-COUNT, CUSTNAME
   : statements
ENDFOR
```

The counter TX-COUNT is defined in working data as a numeric variable. TX-COUNT is initialized to 0 before the FOR construct. The FOR construct then accesses every row in the table. Each row is processed in turn by the statements in the construct. After all of the rows are accessed and processed, the FOR construct terminates.

Using $COUNT To Obtain Total of Accessed Rows

Since the task of counting the number of accessed rows is so commonly performed, PDL provides a function, $COUNT, to return the total number of rows:

```
<<CUST>>
FOR EACH CUSTOMER
   : statements
MOVE $COUNT(CUST) TO WCOUNT
LIST WCOUNT 'CUSTOMER RECORDS READ'
ENDFOR
```

For more information about using $COUNT, see the *Programming Reference Guide*.

Selecting Rows

Assume that all rows in the table are accessed and processed, but only certain rows are counted and listed based on specific conditions. The conditions are defined in the FOR construct. For example, if a column named STATE is provided in the CUSTOMER table, you can include an IF construct in the FOR construct to count only those customers in Texas, while still processing all of the customers in the table:

```
SET TX-COUNT = 0

FOR EACH CUSTOMER
   IF STATE EQ 'TX'
      ADD 1 TO TX-COUNT
      LIST TX-COUNT, CUSTNAME
   ENDIF
   : statements
ENDFOR
```

The counter reflects only the number of customers in Texas.

Several sets of criteria can be evaluated in a single FOR construct by using the SELECT construct. For example, SELECT can process the number of customers in each state. The entire table is accessed through a single execution of the FOR construct, but the customer rows are processed based on the value in the STATE column. In this example, 50 counters were defined in working data to contain the number of customers in each state. The counters were named using the two-character state abbreviations.

The code could be:

```
SET AL = 0
SET AR = 0
SET AZ = 0
. . .
SET WY = 0
FOR EACH CUSTOMER
    SELECT FIRST
       WHEN STATE EQ 'AL'
           ADD 1 TO AL
       WHEN STATE EQ 'AR'
           ADD 1 TO AR
       WHEN STATE EQ 'AZ'
           ADD 1 TO AZ
       . . .
       WHEN STATE EQ 'WY'
           ADD 1 TO WY
    ENDSEL
    : statements
ENDFOR
LIST AL AR AZ . . . WY
```

The processing initializes the counters, increments the appropriate state counter, performs the other statements, and, after the ENDFOR, displays the values.

Selecting a Set of Rows

You can select a set of rows from the entire table by coding IF or SELECT in a FOR construct. In this way, you can apply functions to specific rows in a table. This works well if it is necessary to access every row in the table. The selection criteria are specified on the FOR construct using the WHERE clause to limit the number of rows that are accessed.

You should code a WHERE clause when a specific set of rows is processed. It provides clear information on the set of rows being accessed. It is more efficient to have the database access only the rows, rather than access the entire table and code the selection process in the FOR construct. Not only is the access faster, but it limits the possibility of contention for data across applications. It is essential, when designing an application, to minimize the number of rows being accessed at any one time and the length of time those rows are held.

For example, to process only the customers in Texas, you can specify a FOR EACH construct with a WHERE clause. This code clearly reflects the function and the set of rows to access:

```
FOR EACH CUSTOMER
   WHERE STATE EQ 'TX'
       : statements
ENDFOR
```

Using Compound Selection Criteria

The WHERE clause can contain compound specifications. For example, to obtain only the rows of customers that are in Texas and have an outstanding balance greater than $500, specify:

```
FOR EACH CUSTOMER
    WHERE STATE EQ 'TX'
      AND OPEN$ GT 500
   : statements
ENDFOR
```

You can add further qualifications. Assume that CHECKDATE is a working data variable defined to contain a date value specified at runtime.

```
FOR EACH CUSTOMER
    WHERE STATE EQ 'TX'
      AND OPEN$ GT 500
      AND ACTDT LT CHECKDATE
    : statements
ENDFOR
```

CA Datacom/DB and DB2 handle data types differently. For more information about the data-dependent facilities, see the *Programming Reference Guide*.

Using a Variable as Selection Criteria

You can specify a variable, instead of a literal, as the selection criterion. The following example uses a variable for the selection specification and maintains a single FOR EACH construct to obtain the rows. Since the program prompts for the state and places the user response in a variable (working data or parameter data is valid) named STNAME, you can write the FOR construct to access all customers in the specified state:

```
FOR EACH CUSTOMER
    WHERE STATE EQ STNAME
    : statements
ENDFOR
```

Relational Operators and Conditionals

The WHERE clause can be very powerful not only because it can contain compound specifications and variables to note the selection criteria, but because a wide variety of relational operators are available. They include:

```
=          EQ          EQUAL
¬=         NE          NOT EQUAL          NOT=
           GT          GREATER [THAN]
>=         GE          NOT LESS           NOT<          ¬<
<          LT          LESS [THAN]
<=         LE          NOT GREATER        NOT>          ¬>
```

Valid conditionals include:

```
AND      &

OR       |

NOT      ¬
```

Note: Avoiding ambiguity caused by same names. Frequently, a column in one dataview has the same name as a column in another dataview, field, or variable. To avoid any ambiguity, use the fully qualified name. For example, the CUSTOMER dataview contains a column named STATE and the panel PROMPT contains a field named STATE:

```
FOR EACH CUSTOMER
   WHERE CUSTOMER.STATE EQ PROMPT.STATE
   : statements
ENDFOR
```

Note: Index or key columns speed access. Columns specified with the WHERE clause, do not have to be key columns. The program is usually faster and more efficient when the columns specified in the WHERE clause are key columns. There are instances when it is not necessary. If the occurrence of a column is in a non-critical program or the particular FOR construct is used infrequently, it might be preferable to retain that column as a non-key column since each key does require storage resources and maintenance.

Sequencing the Set of Rows

Once a set of rows is obtained from the database, the sequence in which the rows are processed can be important. For example, the sequence is important when displaying an alphabetical list of the customers in Texas. To retrieve the rows in a specific sequence, use the ORDERED BY clause. If you omit the ORDER BY clause, the order is undefined and the row sequence could vary over time due to either database design changes or changes in the data itself.

Ordering Based on One Column

Assuming that CUSTNAME is the column containing the customer name, you can use the ORDERED BY clause to order all of the customers in Texas alphabetically by name:

```
FOR EACH CUSTOMER
    WHERE STATE EQ 'TX'
    ORDERED BY CUSTNAME
    : statements
ENDFOR
```

Ordering Based on Multiple Columns

You can include additional column names for a more specific sequence. You can code the FOR EACH construct to order the customers in Texas by city and then by name in each city, as shown in the code below. List the column names in the order of precedence. Above, the high-order column, CITY, is named first.

```
FOR EACH CUSTOMER
    WHERE STATE EQ 'TX'
    ORDERED BY CITY, CUSTNAME
    : statements
ENDFOR
```

Ordering in Ascending or Descending Sequence

You can order the rows in ascending or descending sequence. The default sequence is ascending order. You can specify it as DESCENDING or, when you specify several columns, a combination of ASCENDING and DESCENDING. You can modify the previous example to sort by activity date (ACTDT column) in descending sequence (oldest first, rather than most recent first):

```
FOR EACH CUSTOMER
    WHERE STATE EQ 'TX'
    ORDERED BY DESCENDING ACTDT, ASCENDING CUSTNAME
    : statements
ENDFOR
```

It is necessary to specify the keyword ASCENDING for the column CUSTNAME because it follows DESCENDING ACTDT. If you did not specify ASCENDING here, CUSTNAME would be sorted in descending order.

Note: Indexes facilitate sequencing. As with the WHERE clause, it is usually fastest and most efficient to sequence the rows based on key columns. After careful consideration, it might be necessary to request new keys to suit program requirements. Since keys require storage and maintenance, you should probably not define columns specified on infrequently used ORDERED BY clauses or in non-critical programs as keys. If a column specified on the ORDERED BY clause is a component of more than one key, it might be more efficient to specify all of the component columns of the key. In all cases, the database determines which key to use at runtime.

Limiting the Number of Rows

The FOR EACH statement accesses every row that satisfies the specified criteria. FOR FIRST accesses a specified number of rows starting with the first one retrieved.

Obtaining a Specific Number of Rows

Since the FOR FIRST construct, like the FOR EACH construct, is a form of the FOR construct, the WHERE and ORDERED BY clauses specify the selection criteria and the row sequencing. A simple example can demonstrate the difference between FOR FIRST and FOR EACH. Rather than selecting every customer from Texas through FOR EACH, you can use FOR FIRST to access only the first ten customers from Texas.

```
FOR EACH CUSTOMER            FOR FIRST 10 CUSTOMER
       WHERE STATE EQ 'TX'          WHERE STATE EQ 'TX'
    : statements                 : statements
ENDFOR                       ENDFOR
```

Sequencing the Rows

The sequence in which the rows are accessed can be extremely important when selecting a limited number of rows. A very different set of rows could be accessed from one execution to the next. The ORDERED BY clause ensures the sequence. The ten rows are obtained from all customers in Texas based on the sequence specified in the ORDERED BY clause. The resulting ten rows could, for example, contain the customers whose names start with A, then with B, and so on until ten customers are obtained.

```
FOR FIRST 10 CUSTOMER

    WHERE STATE EQ 'TX'
    ORDERED BY CUSTNAME
    : statements
ENDFOR
```

You can specify any number of rows. If there are not a sufficient number of rows to satisfy the specified value, only the available rows are accessed. No error occurs.

Using a Variable to Specify the Number of Rows

You can specify an operand to specify the number of rows as a numeric value or a valid numeric variable. By using a variable to specify the number of rows, you can include the preceding example in a program that prompts for the number of rows to obtain. The returned value can be used as an operand instead of the numeric literal. Assume that NUMROWS is defined as a working data numeric variable:

```
FOR FIRST NUMROWS CUSTOMER

    WHERE STATE EQ 'TX'
    ORDERED BY CUSTNAME
    : statements
ENDFOR
```

Accessing One Row

Frequently, it is necessary to access a single row from the database. This is done so routinely that when a number of rows is not specified on the FOR FIRST construct, 1 is assumed, as in the following, which accesses the first customer in Texas alphabetically.

```
FOR FIRST CUSTOMER
    WHERE STATE EQ 'TX'
    ORDERED BY CUSTNAME
    : statements
ENDFOR
```

The first customer can change from one execution to the next since new customers can be added.

Accessing One Unique Row

Usually a single specific row, in this case a unique customer, is accessed regardless of its relative position alphabetically or otherwise in the table. This is accomplished by the selection conditions. In other words, if the CUSTNAME column was defined to contain unique data for each row, specifying that column on the WHERE clause always retrieves the same row:

```
FOR FIRST CUSTOMER
    WHERE CUSTNAME EQ 'ANACONDA'
    : statements
ENDFOR
```

Since only one row can satisfy this WHERE criteria, an ORDERED BY clause is not included. The WHERE clause does not need to specify any other criteria in this situation. Using the same WHERE clause, the FOR EACH construct retrieves the same single row only because the column value is unique. But with the FOR FIRST construct, it is obvious that a single row is accessed.

No Rows Satisfy Criteria

When coding a FOR construct, consider the possibility that a row is not located to satisfy the selection criteria. If a row is not obtained, processing continues but, unless the program provides for this condition, the user is not notified. If processing depends upon accessing data from the database, unexpected results can occur. The WHEN NONE clause allows the program to handle this situation. For example, when attempting to obtain a single row, you can code the WHEN NONE clause to generate a message if that row is not located:

```
FOR FIRST CUSTOMER
    WHERE CUSTNAME EQ 'ANACONDA'
    : statements
WHEN NONE
    :issue message
    NOTIFY 'No customer named ANACONDA'
ENDFOR
```

Looping to Reprompt

Frequently a program has a FOR construct in a loop to reprompt the user for selection criteria and re-execute the FOR construct. For example, the previous FOR construct could specify a variable rather than a literal and prompt for the criteria. In the following example, the panel is named PROMPT. The panel fields, CUSTNAME and MSG, are fully qualified to enhance readability:

```
LOOP
    DO GET-REQST
UNTIL FINISHED
    SET PROMPT.MSG = $SPACES
    FOR FIRST CUSTOMER
       WHERE CUSTOMER.CUSTNAME EQ PROMPT.CUSTNAME
       : statements
    WHEN NONE
       SET PROMPT.MSG = 'Customer not found'
    ENDFOR
ENDLOOP
```

This code segment demonstrates using WHEN NONE and reprompting. The variable named FINISHED is an example of a flag to terminate the LOOP construct. For more information about coding LOOP, see the *Programming Reference Guide*. The subprocedure, GET-REQST, controls the input and output functions.

Note: Include the WHEN NONE clause with every use of the FOR construct, even when it is highly unlikely that it will execute (that is, no rows satisfy the selection criteria). A comment in the clause can note that the clause is currently not required:

```
FOR EACH CUSTOMER
  : statements
WHEN NONE
  : currently not required
ENDFOR
```

If the criteria on the FOR construct ever change, the WHEN NONE clause can become important. If the clause is always provided, it is more likely to be specified appropriately when the FOR construct is modified. The presence of the WHEN NONE acts as a reminder to code it.

If the code in the WHEN NONE clause is designed to terminate the run, the program should provide a message to the end user and perform cleanup to prevent confusion.

Handling Runtime Errors in Selection

The WHEN NONE clause only executes when there are no rows that satisfy the specified criteria. If, in fact, an error exists in specifying the criteria such that the data type does not conform to the comparison column, a runtime error occurs. For more information about dealing with runtime errors, see the "Error Handling" chapter.

Accessing Rows from Multiple Tables

In a relational database, it is important to be able to access multiple tables simultaneously. You can nest FOR constructs to do this as long as each FOR construct specifies a different cataloged dataview. You can nest any combination of FOR EACH and FOR FIRST. You can also nest FOR NEW. For information about FOR NEW, see Adding Rows. In SQL, dataviews can be defined to contain columns from more than one table. A dataview defined in this manner effectively joins the tables. You can use these dataviews on a read-only basis.

Joining Based on Common Columns

Tables are joined based on common columns. For example, assume two tables exist, a CUSTOMER table and an ORDER table. Each order is associated with a single customer. Each customer can have none, one, or more than one order. The tables are constructed such that they are joined by a common column, CUSTID, containing the customer identification number. This could be shown as:

```
CUSTID  CUSTNAME      |      CUSTID  ORDID
B1000   SMITH         |      B3000   X1234
B2000   JONES         |      B1000   X1222
B3000   GREEN         |      B3000   X1422
```

The previous example shows that customer B1000 has one order, customer B3000 has two orders, and customer B2000 does not have any orders.

To access every order of a specific customer as named in the variable PROMPT.CUSTNAME, specify:

```
FOR FIRST CUSTOMER
    WHERE CUSTOMER.CUSTNAME EQ PROMPT.CUSTNAME
    FOR EACH ORDER
        WHERE ORDER.CUSTID EQ CUSTOMER.CUSTID
        ORDERED BY ORDID
        : statements
    WHEN NONE
        NOTIFY 'Customer has no orders'
    ENDFOR
WHEN NONE
    NOTIFY 'No customer found'
ENDFOR
```

The WHERE clause on the inner FOR locates all of the orders for the customer. In this example, CUSTID is the column that joins the tables. This one column is defined in both tables to contain the customer ID. The column names are qualified to prevent any ambiguity.

Each FOR construct has its own WHERE clause and a WHEN NONE clause and ENDFOR statement to terminate the construct. The inner construct terminates first. Indentation is especially important with nested FOR constructs. Without it, the code becomes confusing. Indentation clarifies at a glance the boundaries of the FOR constructs and the relative nested position of each construct.

Whenever coding nested FOR constructs, without a WHERE clause to limit the set selection of the inner FOR, all rows of the table are read for each row in the outer FOR construct.

Note: Subprocedures promote modularity. To promote structured programming, you can code the FOR constructs in subprocedures. This is especially useful when complex evaluation or multi-level nesting is required. The resulting performance of this segment is the same as if coded in a linear manner:

```
FOR FIRST CUSTOMER
   WHERE CUSTOMER.CUSTNAME EQ PROMPT.CUSTNAME
   DO GETORDS
   : statements
ENDFOR
. . .
<<GETORDS>> PROCEDURE
  FOR EACH ORDER
      WHERE ORDER.CUSTID EQ CUSTOMER.CUSTID
      DO GETDTLS
      : statements
  ENDFOR
ENDPROC
<<GETDTLS>> PROCEDURE
  FOR EACH DETAIL
      WHERE DETAIL.ORDID EQ ORDER.ORDID
      : statements
  ENDFOR
ENDPROC
```

Each subprocedure reflects a level of the FOR construct nesting. This code is generally easier to read and maintain. The evaluations pertinent to each level of the nest are separated into their respective set of rows.

Accessing Multiple Rows from One Table

The dataviews must be unique when nesting FOR constructs, but you can define several dataviews to access the same table. These dataviews can then be used in nested FOR constructs to access more than one row from a single table.

For example, while processing an order, it might be necessary to find out if another order exists for the customer. The following code segment requires two dataviews, PROCORD and OTHERORD, for the same table, ORDER. The dataviews need not specify the same columns or have the same attributes.

```
FOR FIRST PROCORD

   WHERE PROCORD.ORDID EQ PROMPT.ORDID
   FOR EACH OTHERORD
       WHERE OTHERORD.CUSTID EQ PROCORD.CUSTID
         AND OTHERORD.ORDID NE PROCORD.ORDID
       : statements
   WHEN NONE
       : statements
   ENDFOR
ENDFOR
```

The ORDID column retrieves the unique row in the FOR FIRST PROCORD construct and skips the unique row on the FOR EACH OTHERORD construct. The additional qualification on the FOR EACH OTHERORD WHERE clause selects only those orders associated with the customer identified for the unique order in PROCORD.CUSTID.

Accessing Data from a Panel

For more information about this topic, see the *Programming Reference Guide*.

Statements	Functions
TRANSMIT	$CURSOR
MOVE	$EMPTY
REFRESH	$KEY
RESET	$PANEL-ERROR
SET ATTRIBUTE	$PANEL-FIELD-ERROR $PANEL-GROUP-OCCURS $PF $RECEIVED

Displaying Data from a Table or File

Once the set of rows is obtained, you can display the data on the screen. The NOTIFY statement, the LIST statement, and the TRANSMIT statement provide the facilities for writing to the screen. This section does not describe how to define panels, but rather how to display the panels that were defined.

Using the Message Line

The NOTIFY statement is an easy way to display the value of a column at the screen. The data is written to the message line. This provides a means of displaying messages at the screen during runtime without coding a message area in a panel. It does, however, limit the total length of the message to 79 characters.

The NOTIFY text displays when a panel is transmitted or when a run is terminated. NOTIFY can be useful when testing a program or when evaluating user input. For example, NOTIFY CUSTOMER.CUSTNAME displays the contents of the column on the message line. You could use the following segment during testing to ensure that the requested row is accessed:

```
FOR FIRST CUSTOMER

    WHERE CUSTOMER.CUSTNAME EQ PROMPT.CUSTNAME
    NOTIFY CUSTOMER.CUSTNAME ' found'
WHEN NONE
    NOTIFY PROMPT.CUSTNAME ' not found'
ENDFOR
```

Using an Output File

The LIST statement can write the contents of one or more columns to an output file. This file can then display online after the program terminates. The following example writes the customer identification number and name to the output file.

```
FOR FIRST CUSTOMER
   WHERE CUSTOMER.CUSTNAME EQ PROMPT.CUSTNAME
   LIST CUSTOMER.CUSTID  CUSTOMER.CUSTNAME
WHEN NONE
   NOTIFY PROMPT.CUSTNAME ' not found'
ENDFOR
```

LIST can also write the contents of a panel to an output file. In other words, you can assign the column values in a row to the appropriate panel fields, and using LIST, write the entire panel contents to an output file. You can then view the file online or print it.

In the following example, the FOR construct accesses each row of data from the CUSTOMER table. The row is moved to the panel fields by the MOVE BY NAME statement. A panel, named CUSTPNL, was defined to contain the data:

```
FOR FIRST CUSTOMER
    WHERE CUSTOMER.CUSTNAME EQ PROMPT.CUSTNAME
    MOVE CUSTOMER TO CUSTPNL BY NAME
    LIST CUSTPNL
WHEN NONE
    NOTIFY PROMPT.CUSTNAME ' not found'
ENDFOR
```

For more information about using LIST to generate output, see the *Generating Reports Guide*.

Using a Panel for Display

Usually a panel is sent to the screen to display one or more rows of data in a preformatted manner and, optionally, to return data the user typed.

The TRANSMIT statement sends or transmits the panel to the screen. It also pause program execution while waiting for user input. Once the user signals an end to input by pressing a program function key or Enter, CA Ideal returns control to the program at the statement coded directly after TRANSMIT. CA Ideal handles the transaction boundary processing inherent in executing the TRANSMIT statement and restores all current program values. The user input is available as data in panel fields or by evaluating the panel terminating key.

TRANSMIT is important in updating the database. It is explained in Updating a Table or File.

Displaying One Row at a Time

The following segment displays one row from the CUSTOMER table in a panel named CUSTPNL. The MOVE statement is necessary to assign the column values to the panel fields before performing the TRANSMIT for the panel.

```
FOR FIRST CUSTOMER
    WHERE CUSTOMER.CUSTNAME EQ PROMPT.CUSTNAME
    MOVE CUSTOMER TO CUSTPNL BY NAME
WHEN NONE
    NOTIFY PROMPT.CUSTNAME ' not found'
ENDFOR
TRANSMIT CUSTPNL
```

Displaying Multiple Rows at a Time

To display multiple rows of data in a single panel, the panel is populated in the FOR construct and transmitted outside of the FOR construct to ensure that the panel contains the data from all of the rows before the actual display.

In the following example, the panel CUSTPNL was defined to contain a repeating group, CUSTDATA. Each occurrence of the group is assigned the values in one row accessed from the table. A variable, PNLINDX, was defined in working data to act as an index to increment through the repeating group in the panel as the assignment proceeds row by row in the FOR construct:

```
SET PNLINDX = 0
FOR FIRST 10 CUSTOMER
   ORDERED BY CUSTNAME
   SET PNLINDX = PNLINDX  1
   MOVE CUSTOMER TO CUSTPNL.CUSTDATA(PNLINDX) BY NAME
  ENDFOR
  TRANSMIT CUSTPNL
```

This example assumes that the number of rows retrieved does not exceed the size of the panel. Examples are provided in Appendix D, which uses the $PANEL-GROUP-OCCURS function to limit the number of rows retrieved to the actual size of the panel.

Updating a Table or File

Data maintenance includes three basic functions: Modifying existing data, deleting data, and adding data. The performance of these functions is linked to CA Ideal transaction handling.

Modifying Rows

You can only modify existing rows explicitly; that is, by direct assignment. There are several statements that you can use. The following are all valid:

```
SET ACTDT = $TODAY
MOVE PROMPT.CUSTNAME TO CUSTOMER.CUSTNAME

ADD 10 TO OPEN$

SUBTRACT 100 FROM OPEN$
```

You must make all modifications to the database in the scope of a FOR construct. Additionally, in CA Datacom/DB, the native dataview must be defined as updateable.

To understand how the updates are made in the FOR construct, review the following example. All customers in Texas are assigned to a new salesman. This example selects all customers in Texas as the set, accesses each row, one at a time, and assigns the new salesman ID value to the column SALESMAN in each row. The ENDFOR statement denotes the end of processing for the current row when the database is updated.

```
FOR EACH CUSTOMER
    WHERE CUSTOMER.STATE EQ 'TX'
    SET CUSTOMER.SALESMAN = NEWID
...
ENDFOR
```

Controlling Updates

You can code the FOR construct to contain the evaluation that limits which retrieved rows are updated. In the following example, an IF construct is used. The update to the current row only occurs if the city is Amarillo.

```
FOR EACH CUSTOMER
    WHERE CUSTOMER.STATE EQ 'TX'
    IF CITY EQ 'AMARILLO'
       SET CUSTOMER.SALESMAN = NEWID
    ENDIF
...
ENDFOR
```

Abandoning an Update

The modifications are applied to the database when the ENDFOR statement is reached. Before the ENDFOR is executed, you can abandon changes by exiting or quitting the FOR construct.

When QUIT is coded in a FOR construct, the update to the current row and any subsequent rows does not occur. The QUIT statement exits the FOR construct. Therefore, the ENDFOR statement is never reached to apply the changes to the current row in the database and the subsequent rows are never processed. The database still contains the changes made to previous rows processed before the QUIT statement executed.

You can also code a PROCESS NEXT statement in a FOR construct. This statement abandons the current update, but processin continues with the next iteration of the FOR construct allowing subsequent updates to take place.

Abandoning Multiple Changes

You can also abandon changes to rows in nested FOR constructs. Notice the use of a label to identify which FOR construct is the object of the QUIT.

```
<<CUST>>
  FOR FIRST CUSTOMER
    WHERE CUSTOMER.CUSTNAME EQ PROMPT.CUSTNAME
    : statements
    <<ORD>>
      FOR FIRST ORDER
      WHERE ORDER.CUSTID EQ CUSTOMER.CUSTID
      : statements
      IF condition1
         QUIT ORD
      ENDIF
      IF condition2
         QUIT CUST
      ENDIF
   ENDFOR
   IF condition3
      QUIT CUST
   ENDIF
ENDFOR
```

In this example, the changes to the current ORDER row are bypassed if condition1 is true. Changes to the current ORDER row and the CUSTOMER row are ignored if condition2 is true. Changes to the CUSTOMER row are aborted if condition3 is true. A label is not required on the QUIT statement of the current FOR construct used in the IF construct for condition1 and condition3; however, the program is clearer and easier to maintain if labels are specified.

Deleting Rows

Deleting rows from a table is an integral part of database maintenance. Unlike modifying the row, deleting the row takes place immediately. The entire row is deleted, regardless of the columns specified in the dataview. In other words, when modifying the contents of a row, only the columns specified on the dataview are available. The system evaluates the actual modifications and performs them at the ENDFOR. The DELETE statement is executed immediately, not at the ENDFOR.

In the following example, the DELETE statement is used. Assume a specific row is deleted if the user presses F4. Otherwise, the subprocedure EVALINP evaluates the row.

```
FOR FIRST CUSTOMER

   WHERE CUSTOMER.CUSTNAME EQ PROMPT.CUSTNAME
     MOVE CUSTOMER TO CUSTPNL BY NAME
     IF $PF4
         DELETE CUSTOMER
         NOTIFY 'DELETE successful'
     ELSE
         DO EVALINP
     ENDIF
WHEN NONE
    NOTIFY PROMPT.CUSTOMER 'not found'
ENDFOR
```

Since DELETE is a destructive statement, code it carefully. For example, the previous example displays the row and only executes the DELETE if you press a specific key. You can include a message to indicate that the function was successfully performed.

Restoring a Deleted Row

The BACKOUT statement, since it restores the database to the previous CHECKPOINT or stable condition, restores the deleted row if an intervening CHECKPOINT or TRANSMIT statement with implicit CHECKPOINT was not executed.

Adding Rows

The FOR NEW construct adds new rows to a table. Unlike the other FOR constructs, FOR NEW does not iterate. You can code a FOR NEW construct in a LOOP construct to iterate adding rows to a table. Then evaluation is coded in the LOOP to terminate it. For example, in the following code, a subprocedure GETDATA obtains the information for the row and evaluates that information before executing FOR NEW. The subprocedure also sets the flag FINISHED to TRUE based on user input (the user presses a function key) to terminate the loop.

```
LOOP
   DO GETDATA
UNTIL FINISHED
   FOR NEW CUSTOMER
      MOVE CUSTPNL TO CUSTOMER BY NAME
   ENDFOR
ENDLOOP
```

The new row is added to the database at the ENDFOR. Before the ENDFOR, you can make any modifications to the table without directly impacting the database. Also, you can code QUIT to bypass adding the row.

Maintaining Unique Data

When the ENDFOR statement executes, the new row is evaluated and an attempt is made to insert it into the database. If the database table was set up to not allow duplicates, a runtime error occurs if adding the row results in a duplicate. A runtime error causes CA Ideal to display a message on the message line and terminate the run. To avoid this, the WHEN DUPLICATE clause allows the program to recover from the error. Although the error cannot be corrected in the program, the program does not terminate. For example, in the following code segment, a message displays. The LOOP construct prompts the user to re-enter the information for the new row:

```
LOOP
   DO GETDATA
UNTIL FINISHED
   FOR NEW CUSTOMER
      MOVE CUSTPNL TO CUSTOMER BY NAME
   WHEN DUPLICATE
      NOTIFY 'Duplicate Row, Respecify'
   ENDFOR
ENDLOOP
```

Note: When using FOR NEW in a LOOP construct, you might want to keep track of the number of rows that are added. You can include a counter in the FOR construct to perform this function. Assume that a variable named CTR was defined in working data and is used in the following example to accumulate the number of new rows:

```
FOR NEW CUSTOMER
   MOVE CUSTPNL TO CUSTOMER BY NAME
   ADD 1 TO CTR
WHEN DUPLICATE
   NOTIFY 'Duplicate Value, Please Respecify'
   SUBTRACT 1 FROM CTR
ENDFOR
```

CTR is decremented if the WHEN DUPLICATE clause is executed. This is necessary. The value of the variable is incremented in the FOR construct, regardless of whether the row is successfully added to the database. Since rows are evaluated and added to the database at the end of the FOR construct, a duplicate row is not encountered before the ENDFOR statement. If the row is a duplicate and not added to the database, the CTR must be decremented in the WHEN DUPLICATE clause to maintain an accurate value in the counter.

Transmitting in FOR NEW

You can code TRANSMIT in the FOR NEW construct. Although a TRANSMIT performs a CHECKPOINT, the current new row is not included since the new row is not written to the database until the ENDFOR. A TRANSMIT after the ENDFOR commits the new row.

Note: When adding rows to the database, use a dataview containing all of the columns in the table. That way, the columns are initialized either explicitly by the program or implicitly by the default values as specified for the columns.

When default values are not specified for the columns included in the dataview, values are defined based on the data type. Numeric data is set to zero and alphanumeric data is set to blank. Nullable columns are set to NULL. All time, date, and timestamp columns are set to the respective current value at the time the row is added.

Any columns not included in the dataview must also be initialized when the row is added to the database, otherwise, a runtime error can occur because CA Datacom/DB initializes the column to blank regardless of data type, unless Datadictionary field attribute, DBEDITS, is specified as Y. To avoid initialization errors in CA Ideal, use a dataview that includes all of the columns in the row or specify Datadictionary field attribute DBEDITS=Y. For more information, see the *[set the ddb variable for your book] Datadictionary Attribute* Reference Guide.

Committing the Changes

You can modify and commit the changes to the database. The following section explains it in detail.

Updating and Committing

You modify and formally commit the changes to the database in two separate steps. That is, changes to the database are evaluated and applied to each row in the FOR construct when the ENDFOR statement is reached. You can include code in the FOR construct to control the actual changes. This is distinct from committing the changes.

A checkpoint is required to commit the changes to the database. A checkpoint establishes the current stable state of the database. Any changes to the database become permanent or a part of the current stable state. A checkpoint can occur explicitly using the CHECKPOINT statement or implicitly using the TRANSMIT statement.

Before a checkpoint, either implicit or explicit, you can remove or back out the updates from the database through the BACKOUT statement. As a safeguard, BACKOUT is automatically executed when a program abends. You can code it in a program to ensure that changes are not committed for a set of rows when some function cannot complete.

Explicit CHECKPOINT

In the following example, the CHECKPOINT statement commits the changes to the database.

```
FOR FIRST CUSTOMER                : access the row
    WHERE CUSTOMER.CUSTNAME EQ PROMPT.CUSTNAME
    : statements                  : modify the row
ENDFOR                            : apply the changes
                                  :   to the database
CHECKPOINT                        : changes committed
```

In this example, a single customer row is accessed and processed using the statements. The changes are made to the database at the ENDFOR. The CHECKPOINT outside of the FOR construct formally commits those changes. Once checkpointed, the changes cannot be lost or backed out of the database.

When a FOR EACH or FIRST construct accesses and modifies several rows, the CHECKPOINT commits the changes made to all of those rows. For example, all of the customers in Texas are accessed and assigned to a new salesman. Although each row is processed individually and the database updated one row at a time, the CHECKPOINT statement outside of the construct commits all of the rows at the same time:

```
FOR EACH CUSTOMER                  : access the row
   SET CUSTOMER.SALESMAN = NEWID   : modify the row
ENDFOR                             : apply the change
                                   :    to the database
CHECKPOINT                         : all rows committed
```

Removing Changes from the Database

The BACKOUT statement removes all of the updates to the database that were made since the last CHECKPOINT. BACKOUT returns the database to the most recent stable state. For example:

```
CHECKPOINT                         : changes committed

FOR EACH CUSTOMER                  : access the row
   SET CUSTOMER.SALESMAN = NEWID   : modify the row
ENDFOR                             : apply the changes
                                   :   to the database
BACKOUT                            : changes backed out
                                   :  to checkpoint
```

In the previous example, modifications are made to the row in the FOR construct. The BACKOUT statement following the FOR construct, however, removes those changes and returns the database to the previous CHECKPOINT state.

Note: You can specify CHECKPOINT or BACKOUT in the FOR construct. Be aware though, that CHECKPOINT in the FOR construct does not include the current row. The modifications to the current row are not applied to the database until the ENDFOR statement is executed. Similarly, BACKOUT does not affect the current row since the changes are not yet in the database. A CHECKPOINT or BACKOUT statement following ENDFOR includes the last accessed row.

CHECKPOINT and BACKOUT release exclusive control of the current row. For further information, see the CA Datacom/DB documentation.

TRANSMIT with CHECKPOINT

When you use mainframe CA Ideal in a CICS environment, executing a TRANSMIT statement always performs a CHECKPOINT. This ends the current transaction and protects modifications against loss if the system abnormally terminates during TRANSMIT.

The TRANSMIT statement causes a CHECKPOINT.

The following shows the sequence of events that occurs when the program is updating a row and TRANSMIT (with a CHECKPOINT) is executed in the FOR construct:

```
FOR FIRST CUSTOMER                : access the row
   WHERE CUSTOMER.CUSTNAME EQ PROMPT.CUSTNAME
   : statements                   : modify the row
   TRANSMIT CUSTPNL               : commit previous row
ENDFOR                            : apply the changes
                                  :    to the database
```

The CHECKPOINT action of the TRANSMIT commits any changes made to the database *before* the current iteration of the FOR construct. Any changes made to the *current* row are not applied to the database because ENDFOR was not executed for the current row. A TRANSMIT or CHECKPOINT following ENDFOR commits the current modifications.

For online applications, terminal I/O causes a database checkpoint to occur. Because user input at the terminal can be time-consuming and has the potential to obtain control of multiple rows for indefinite periods of time, the TRANSMIT statement causes a CHECKPOINT.

In other words, with CA Datacom/DB, control is not maintained across CICS transaction boundaries. Each TRANSMIT is considered a CICS transaction. With DB2 and DBSQL ANSI mode, the database cursor is released when any checkpoint occurs.

Checkpointing while Updating

More than likely, an online application transmits a panel containing the data to update. Just as with display, the data is moved to the panel fields before the TRANSMIT statement is executed. The user then modifies the data displayed on the panel. When the user returns control to the program, the data in the panel fields must then be assigned or moved to the dataview. The actual update to the dataview can be performed based on the evaluation of some condition before executing the MOVE statement. For example, assume that if the user presses PF4, the changes are applied. You can code the program as:

```
FOR FIRST CUSTOMER

   WHERE CUSTOMER.CUSTNAME EQ PROMPT.CUSTNAME
     MOVE CUSTOMER TO CUSTPNL BY NAME
     TRANSMIT CUSTPNL
     IF $KEY EQ 'PF4'
        MOVE CUSTPNL TO CUSTOMER BY NAME
     ENDIF
ENDFOR
```

CA Ideal ensures that the modified row has not changed from the initial access to the point of update. An error condition occurs if an attempt is made to apply changes to a row that was modified between the time it was accessed and the time of the actual update to the row. If the checkpoint occurs outside of the FOR construct, there should be no conflict. Access is maintained for the update. However, if you code TRANSMIT in the FOR construct, it causes an automatic checkpoint and the row is released. Another user can access and modify the row between the time of access and time of actual update. If the row was changed from the original, control passes to the error procedure. You can design a program-defined error procedure to handle this situation by re-accessing the row or bypassing the changes. For more information, see the "Error Handling" chapter.

A CHECKPOINT is automatically issued when an application terminates successfully. A BACKOUT is issued when an application terminates abnormally due to a system error. In all other cases, a BACKOUT is only issued if coded in the program.

Multi-User Considerations

It is important to consider database access and row availability in a multi-user system. The row is not updated in the database until the ENDFOR when accessing data from a CA Datacom/DB CBS database or a DB2 database. The control of the row from retrieval to release is different in these databases. In either, a CHECKPOINT, TRANSMIT, or BACKOUT releases the row.

CA Datacom/DB CBS

There are two levels of exclusive control: Primary and secondary. In the FOR construct, the row currently updated is held with primary exclusive control. Once ENDFOR is reached, the updates are applied to the database and the row is held with secondary exclusive control. Primary exclusive control indicates the current row, secondary exclusive control indicates the updated rows that were not checkpointed. All control is released when a checkpoint is performed. This is demonstrated by an example:

```
FOR ...                 : access with exclusive control
   SET or MOVE          : update dataview
ENDFOR                  : update database row and downgrade
                        : to secondary control
CHECKPOINT              : release all control
```

The program that controls the row can access any row, even those held with primary and secondary exclusive control. Another user cannot update rows held under any level of control.

DB2

There is one level of access in a program. DB2 supports table level locking and page level locking for all releases. With Release 4, DB2 also supports row level locking, if implemented, and the no lock isolation level. When a row is locked with update intent, no other users can access that row until it is released. The updates are performed at the ENDFOR. When page or table level locking is used, a CHECKPOINT is required to release the lock. When row level locking is used, the lock is released when another row is read from the same set.

If the flag is on, the buffer image of the originally accessed row held in a dataview buffer is compared to the dataview on the FOR construct. If these match, the data has not changed and no update is necessary. If they do not match, an update is being performed. The exact image is then compared to the row currently in the database. If these match, the row is accessed and updated. If they do not match, the row has been modified by another user between the time of the original access and the attempt to apply the updates. An error condition occurs.

The error condition is identified by evaluating $ERROR-DVW-STATUS for 'I3' for CA Datacom/DB and $ERROR-TYPE 'D72' for DB2. For more information about handling this error condition, see the
chapter 6, "Error Handling".

CHECKPOINT in FOR EACH

Thus far, the examples using TRANSMIT have modified a single row. When accessing a CA Datacom/DB native dataview, you can issue TRANSMIT in the FOR EACH construct. Even though the accessed rows are released when a TRANSMIT with implicit CHECKPOINT, CHECKPOINT, or BACKOUT statement is executed, the set definition and current position in the set are not lost. This means that you can code these statements in any FOR construct.

The following example shows changes were not made to the current row. The TRANSMIT statement commits the previous row. A CHECKPOINT or TRANSMIT statement outside of the FOR construct is needed to commit the last row accessed.

```
FOR EACH CUSTOMER                    : access the row
   MOVE CUSTOMER TO CUSTPNL BY NAME  : fill panel
   TRANSMIT CUSTPNL                  : transmit panel
   MOVE CUSTPNL TO CUSTOMER BY NAME  : update row
ENDFOR                               : apply the change
                                     :  to the database
CHECKPOINT                           : commit last row
```

DB2 Considerations

In DB2, you cannot update multiple rows using TRANSMIT or CHECKPOINT in the FOR construct. The database cursor is released along with the accessed rows when TRANSMIT, CHECKPOINT, or BACKOUT is executed. CA Ideal is only able to locate the current unique row and cannot locate the current row in a multi-row set. For that reason, you can only specify TRANSMIT, CHECKPOINT, and BACKOUT in a FOR FIRST 1 construct that accesses a unique row.

Accessing Released Dataview Data

Once the dataview is released by a TRANSMIT with implicit CHECKPOINT, CHECKPOINT, or BACKOUT statement, the dataview fields of the last accessed row are still accessible but not modifiable. For example, as part of an error processing routine, BACKOUT can be followed by a LIST statement to output the dataview fields.

Additionally, the data in the last accessed dataview row is available outside of the FOR construct on a read-only basis. Updates can only occur in the FOR construct.

Chapter 3: Subprograms

A subprogram is any program that another program calls. CA Ideal can pass data between a CA Ideal calling program and a subprogram. The subprogram may be another CA Ideal program or a *non-Ideal* program written in COBOL, PL/I, or assembler.

Calling a Subprogram

A CALL statement passes control from a CA Ideal program to a subprogram, and, optionally, passes data (in the form of input or update data items) between the two. After the called program terminates, control is returned to the calling program at the next sequential statement in the calling procedure.

In the following example, the MAINMENU panel is transmitted and then, depending on the selection entered, one of the update programs is invoked. When the update program terminates, control returns to the NOTIFY statement in the calling program.

```
LOOP
    TRANSMIT MAINMENU
  UNTIL $PF3
    SELECT FIRST ACTION
       WHEN SELECTION='1'
          CALL CUSTUPDT
          NOTIFY 'Customer updates completed.  Press PF3 to quit.'
       WHEN SELECTION='2'

              CALL ORDUPDAT
          NOTIFY 'Order updates completed.  Press PF3 to quit.'
       WHEN SELECTION='3'
          CALL INVENUPD
          NOTIFY 'Inventory updates completed.  Press PF3 to quit.'
       WHEN OTHER
          NOTIFY 'Select option 1, 2, or 3, or press PF3 to quit.'
    ENDSEL
ENDLOOP
```

Passing Data to a Subprogram

Subprograms let CA Ideal access external routines and share procedures among several applications. The calling program references data items with a CALL statement. A data item can be the name of an elementary field, the name of a group, or a literal. The subprogram includes parameter definitions that describe these data items. CA Ideal manages the logical connections between the two.

In the following example, the program CHK-BACK determines whether the item entered is back-ordered. The ITEM-ID from the panel is passed to the CHK-BACK program as input. The working data field ANSWER receives the response from the CHK-BACK program.

```
<<UPDATE-ITEM>> PROCEDURE
   FOR FIRST ITEM
     WHERE ITEM.ITMID EQ INCOMING.ITMID
        MOVE INCOMING TO ITEM BY NAME
        CALL CHK-BACK USING INPUT INCOMING.ITMID UPDATE ANSWER
        IF ANSWER = 'YES'
            NOTIFY 'Press PF5 for a report on back-ordered item.'
        ENDIF
     WHEN NONE
        DO ADD-ITEM
   ENDFOR
ENDPROC
```

You must define ITEM-ID and ANSWER as parameters for the CHK-BACK subprogram, although they need not have the same names.

Executing a Subprogram Asynchronously

In the previous example, the user was notified to press PF5 for a report. Producing reports can be a time-consuming task, and you might want to run the report program asynchronously. This way, the user can proceed to other tasks online while the report is produced. To execute a subprogram asynchronously, use the INITIATE statement instead of the CALL statement.

In the next example, the PF5 key prints the report of outstanding orders for a back-ordered product.

```
<<PROCESS-INCOMNG>> PROCEDURE
  TRANSMIT INCOMING
  SELECT FIRST ACTION
    WHEN $PF3
      QUIT PROCEDURE
    WHEN $PF5
      INITIATE PRNT-ORD USING INPUT INCOMING.ITEM-ID
    WHEN $ENTER-KEY
      DO UPDATE-ITEM
    WHEN NONE
      DO BAD-KEY
  ENDSEL
ENDPROC
```

The PRNT-ORD program uses the ITEM-ID entered on the panel INCOMING to find the orders that could not be filled because they included that item. Meanwhile, the user can continue with the next incoming item.

Requirements for Subprograms

Any subprogram called by a CA Ideal program must follow certain conventions.

- You must define the subprogram to CA Ideal using the program definition identification fill-in. For CA Ideal subprograms, any parameters passed to the subprogram from the calling program must be defined on the program identification fill-in. For non-Ideal subprograms, any parameters passed to the subprogram from the calling program must be defined in the program identification fill-in and the subprograms linkage section. When defining the parameters, keep the following in mind:
  - Include the descriptions of each data item expected from the calling program.
  - Specify level-1 parameter definitions in the same order as the corresponding data items are specified in the CALL statement of the calling program. Each data item in the CALL statement is matched, in order, with a level-1 parameter in the called program.
  - The passed data item and the corresponding level-1 parameter can have subordinate fields. However, the data types and structures of the passed data items must match that of the receiving data items.
  - The names assigned to parameters in a subprogram need not be the same as the names of the data items referenced in the CALL statement.
- The calling program must include the subprogram as a resource on the program resource fill-in.
- Make a distinction between programs run online or in batch. A CA Ideal program run online can call a CA Ideal subprogram, a non-Ideal online subprogram or initiate an asynchronous run of a CA Ideal program. A CA Ideal program run in batch can CALL any CA Ideal subprogram that does not contain a TRANSMIT or a non-Ideal batch subprogram.

Beyond these facts, there are a number of significant differences between the requirements for the design and use of CA Ideal subprograms and of non-Ideal subprograms. For example, when the subprogram is coded in COBOL, PL/I, or Assembler, it is written and maintained outside of CA Ideal. When the subprogram executes, CA Ideal relinquishes control to the subprogram.

You should be careful that the non-Ideal subprogram does not modify the CA Ideal environment. When a CA Ideal subprogram executes, CA Ideal maintains control of the environment. Due to this major difference, the remainder of this section is divided into two parts: Calling CA Ideal Subprograms and Calling non-Ideal Subprograms.

Note: CA Ideal programs cannot be called as subprograms by programs written in other languages when executing in batch. Online, a CICS command level program can invoke a CA Ideal session by using the EXEC CICS START command to start a CA Ideal transparent sign-on. For more information about transparent sign-on, see the *Administration Guide*.

Parameter Matching for CA Ideal Subprograms

When a CA Ideal program calls a CA Ideal subprogram, CA Ideal can make certain assumptions and accommodate some differences in attributes that might exist at runtime. This is because CA Ideal controls the runtime environments of both programs.

Since CA Ideal controls the runtime environment, you can define one of the two methods of parameter matching-dynamic or identical-for each level-1 parameter.

Dynamic Matching

With dynamic matching, you must specify only the type and, for alphanumeric and variable length types, the length of the parameter. CA Ideal copies the remaining attributes from the corresponding item in the calling program. For example, if the parameter is simply defined as signed numeric with dynamic matching, on one call, the parameter might be packed; on the next call, it might be binary.

The rules for characters/digits specification for the parameter depend on the field type. For more information about specifying parameters, see the chapter “Reading and Writing Data.”

The flexibility provided by dynamic matching carries a cost in overhead. It limits how you can use some kinds of parameters. For this reason, you can define parameters with identical matching.

Identical Matching

With identical matching, you must specify the attributes needed to define the parameter (at least Type, Internal Type for numeric and date, and Characters/Digits). CA Ideal checks that the corresponding data item in the calling program has the same attributes.

Identical matching allows CA Ideal to generate more efficient code at compile time because the field attributes are already known. In addition, you must define group parameters with identical matching to use in several CA Ideal contexts. See the MOVE and SET statements and conditional expressions.

Linkage Conventions for CA Ideal Subprograms

CA Ideal passes the address of each data item in the CALL statement. When the CA Ideal subprogram refers to a parameter, whether as a source or target, the subprogram is actually referring directly to the corresponding data item in the calling program.

Since CA Ideal controls the entire runtime environment, any attempt to modify the value of an update parameter that corresponds to a data item that was named in the CALL statement as an input data item results in a runtime error.

Parameter Rules

The following rules apply to defining parameters in a CA Ideal subprogram:

- You must specify a parameter in the subprogram for each data item to pass. The parameter definitions must match the type of each data item.
- The following restrictions apply to specific attributes:
  - **alphanumeric**

    fixed length-If you are using dynamic matching, then the length must be equal to or greater than the actual passed data item. The Characters attribute specifies the maximum length of the data item. If the item length is less when passed, then the actual length is the length assigned to the parameter.

    If you are using identical matching, the length must be the same.

    variable length-If you are using dynamic matching with update intent, length must be equal to or greater than actual passed data item.

    If you are not using matching with update intent, whether dynamic or identical matching, length must be the same.
  - **Numeric**-If dynamic matching, the internal type is taken from the data item in the calling program. Digit values, if specified, for integer and decimal places on parameter must match data item. Otherwise, it is taken from data item.

    If you are using identical matching, internal type and digits values must match.
  - **date**-Same as numeric.
  - **flag**-No restrictions.
  - **group name**-The structures must be compatible using the tests for the MOVE BY POSITION command.
  - **condition name**-Although a passed group field can contain a condition name, it cannot be passed as an elementary data item.

- **occur value with DEP ON clause-**If dynamic matching, the object of the DEP ON clause must match. The object is the field that determines the number of occurrences. If identical matching, the Occur and DEP ON clauses must match. The subscripted name of an occurrence is treated as an elementary data item.
- **occur value without DEP ON clause-**If dynamic matching, value must be equal to or greater than data item. If the value specifies the maximum number occurrences, then the actual number passed is assigned to the parameter. If identical matching, the Occur and DEP ON clauses must match. The subscripted name of an occurrence is treated as an elementary data item.
- **Redefines-**If not defined on the parameter, it is ignored. If specified on the parameter but not on the data item, a runtime error occurs.
- **Nullable fields (defined using WITH IND)-**If passed by the calling program, it must be defined using WITH IND in the called subprogram.

- A panel passed as a group must be dynamic.
- A CA Ideal program cannot be called recursively; that is, a program cannot call itself or call a program that is already active in the current CA Ideal run-unit.
- A runtime error occurs if a CALL statement passes more data items than the number of level-1 parameters defined for the called subprogram. However, if a CALL statement passes fewer data items than the number of level-1 parameters defined for the called subprogram, no runtime error occurs unless the subprogram attempts to reference a parameter for which no data item was passed. Since parameters are passed and referenced positionally, you can omit parameters from the end of the CALL statement list without affecting the other parameters. You cannot omit values from the list. The $RECEIVED function can determine which level-1 parameters were actually passed.

Defining Non-Ideal Subprograms

To create, compile, and call a non-Ideal subprogram, complete the following steps:

1. On the CA Ideal command line, issue a CREATE PGM command for the subprogram. Use the same name as the subprogram.
2. On the IDENTIFICATION panel, change the language prompt from IDEAL to one of the following:
   - COBOL
   - PLI
   - ASM
3. On the PARAMETER panel, fill in all parameters passed to and from the non-Ideal subprogram.
4. On the CA Ideal command line, issue an EDIT PGM *programname* command, where *programname* is the name of the calling program.

5. Define the subprogram as a RESOURCE of the calling program.
6. To call the subprogram, code an IDEAL CALL statement in the calling program. You can pass a maximum of 16 parameters.
7. Compile the main program.
8. Compile and link the subprogram. For instructions to compile, refer to the documentation for your compiler. For instructions to link, see Linkage Conventions for Non-Ideal Subprograms.
9. Run the main program. When you run the calling program, CA Ideal calls the subprogram and passes the proper parameters to it. In addition, CA Ideal automatically performs any internal data-type conversions.

CA Ideal does not control the runtime environment of a non-Ideal subprogram; therefore, certain situations are handled with less flexibility than when both programs are CA Ideal.

Identical Parameter Matching

Non-Ideal subprograms always use identical parameter matching and must specify the Type, Internal Type for numeric parameters, and characters/digits. You can specify only alphanumeric, numeric signed, and numeric unsigned types.

Linkage Conventions for Non-Ideal Subprograms

Each time a non-Ideal subprogram is called online, CICS determines whether the program is reloaded. CICS requires all programs to be at least quasi-reentrant. You can, however, specify that non-Ideal subprograms called in batch, which may be only reusable and not reentrant, remain in memory for the entire run or until they are explicitly released (by a RELEASE PROGRAM statement). The New Copy on Call specification is part of the program identification for non-Ideal subprograms.

If the Digits specification contains any decimal (fraction) places, CA Ideal aligns the numeric value according to the Digits specification, but you must design the non-Ideal subprogram to account for the implied position of the decimal point. For example, if the Digits specification is 3.2 and the value to pass is 123.45 in packed decimal format, the internal value is X'12345C'. You must design the non-Ideal subprogram to expect the implied decimal point to be between the third and fourth digits.

To be flexible with other language conventions, CA Ideal does not align binary operands in non-Ideal subprogram parameter definitions. For example, if the SYNC option is not used in COBOL, binary fields are not aligned. If CA Ideal always forced alignment, it might be impossible to pass a group data item that contained one or more binary fields to a COBOL subprogram that does not use the SYNC option.

If the COBOL subprogram does use the SYNC option or if the Assembler subprogram defines the binary fields with the H (halfword) or F (fullword) attribute, the CA Ideal programmer should insert the appropriate filler items in the non-Ideal subprogram parameter definition. For example, if the COBOL description is:

```
001000 LINKAGE SECTION.
001010 01  GROUP-ITEM.
001020     02  ALPHA-1               PIC X(1).
001030*       NOTE: COBOL INSERTS A "SLACK" BYTE HERE
001040     02  BINARY-2              PIC S9(4) COMP SYNC.
```

It is equivalent to the Assembler description:

```
             DS                 0D
GROUPITM     DS                 0XL4
ALPHA1       DS                 CL1
NOTE: ASSEMBLER INSERTS A "SLACK" BYTE HERE
BINARY2      DS                 H
```

The following is an example of a non-Ideal subprogram parameter definition:

```
1 GROUP-ITEM
2 ALPHA-1                       X   1
2 SLACK-1                       X   1
2 BINARY-2                      B   4
```

CA Ideal does not support binary numbers larger than nine decimal (base-10) digits. If it is necessary to pass a larger number to a non-Ideal subprogram, pass it as a zoned decimal or packed decimal field. If the field passed is a binary field larger than four bytes in a dataview, see the summary of storage requirements for each internal type in the concepts section of the *Programming Reference Guide*.

To determine what specification for the CA Ideal Characters/Digits column (number of decimal, or base-10, digits) is needed for a given number of bytes of storage for a non-Ideal subprogram, see the summary of storage requirements for each internal type in the concepts section of the *Programming Reference Guide*.

CA Ideal applications that access SQL databases might need to define fields that can receive null values. Every parameter field declared as a nullable field (WITH IND) passed as a parameter to a non-Ideal subprogram generates a two-byte NULL indicator field. For each parameter passed, the indicator fields are located following all the fields in the actual data structure passed to a non-Ideal subprogram. The location corresponds to the order in which the nullable parameters in the parameter data definition panel are defined.

If a parameter field is the object of OCCURs, then as many indicator fields are generated as there are occurrences of the parameter field. For example, given the following parameter data structure:

```
01      PARM-1
  02     A      . . . WITH IND
  02     B      . . . OCCUR 3
    03  C       . . . WITH IND
    03  D       . . . OCCUR 3 WITH IND
```

The following null indicator fields are generated:

- One indicator for A
- Three indicators for C
- Nine indicators for D in the following order:

```
D(1,1), D(1,2), D(1,3)
D(2,1), D(2,2), D(2,3)
        D(3,1), D(3,2), D(3,3)
```

The COBOL description is:

```
01      PARM-1
  02     A      .       . .
  02     B              OCCURS 3 TIMES . . .
    03  C               . . .
    03  D               OCCURS 3 TIMES . . .
  02     A-IND  PIC S9(4) COMP.
  02     C-IND  OCCURS 3 TIMES  PIC S9(4) COMP.
  02     D-IND  OCCURS 9 TIMES  PIC S9(4) COMP.
```

Parameter Rules

The following rules apply when a CA Ideal program calls a non-Ideal subprogram:

- The name of the called program must be in the resource table of the calling program and must conform to CA Ideal naming conventions, with the additional restriction that the name cannot contain hyphens (due to operating system requirements).
- You must specify a parameter in the subprogram for each data item to pass. The parameter definitions must match the type of each data item.

  You can pass a maximum of 16 data items from a CA Ideal program to a non-Ideal subprogram.

  A compile-time error occurs if a CALL statement attempts to pass more data items than the number of level-1 parameters defined for the subprogram.

  Unpredictable results occur when a subprogram references a parameter that corresponds to a data item that was not passed.
- The following restrictions apply to specific attributes:
  - **alphanumeric-**The corresponding parameter must be alphanumeric. When the program is called, a MOVE BY POSITION is performed. When the length of the data item is not the same as the parameter, the data is padded or truncated as required to fit.
  - **numeric-**The corresponding parameter must be defined as a packed decimal, zoned decimal, or binary. When the program is called, a MOVE BY POSITION is performed. The data item can be converted and decimal aligned.
  - **date-**Same as numeric.
  - **flag-**Not allowed.
  - **group name-**The structures must be compatible using the tests for the MOVE BY POSITION command.
  - **condition name-**Not allowed.
  - **redefines-**Ignored.
- The parameter definitions of a non-Ideal subprogram actually become part of the calling program. Therefore, you must define parameter data for the non-Ideal subprogram before the calling program is compiled. The calling program must be recompiled if the subprogram's parameter data changes.
- The parameters for non-Ideal programs are linked as follows:
  - In batch (z/OS and VSE), standard linkage is used (R1 points to a parameter list)
  - In CICS (z/OS and VSE), the address list for the parameters is stored in the beginning of the TWA (Transaction Work Area).

- The designation of the parameters on the CALL statement should match the parameter definition. When the designation of a data item as UPDATE or INPUT in the CALL statement does not match the parameter definition of U or I for the called program, the following applies:
  - If the parameter is defined as I-input and sent as U-update on the CALL statement, an error does not occur as long as the called program only references the parameter. If the non-Ideal subprogram attempts to update the parameter, the passed data item is not updated.
  - If the parameter is defined as U-update and sent as I-input, a compile error occurs regardless of whether the subprogram attempts to modify the data item.

Passing Parameters to Non-Ideal Subprograms

When calling non-Ideal subprograms, CA Ideal does not have control over the entire runtime environment, but only over the way the subprogram is called. For example, if the user coded the following and described an input (I) parameter in the non-Ideal subprogram definition for the subprogram COBOL1 to correspond to the data item named ALPHA, the COBOL subprogram could actually modify the contents of the data item outside the CA Ideal environment.

```
CALL COBOL1 USING INPUT ALPHA
```

To protect the integrity of the CA Ideal environment, CA Ideal uses the following technique: When a CA Ideal program calls a non-Ideal subprogram, the parameter definition for the non-Ideal subprogram constructs an intermediate data storage area with the defined characteristics contained in the CA Ideal calling program. For this reason, the non-Ideal subprogram definition must actually exist when the calling program is compiled.

Data items in the intermediate storage area are *not* aligned. Any alignment required for correct function of the non-Ideal subprogram is the responsibility of the programmer. You can establish this alignment by inserting the appropriate FILLER items in the non-Ideal subprogram parameter definition fill-in.

The field names in the non-Ideal subprogram parameter definition are internally prefixed with a percent sign (%) to avoid duplicate naming problems between the main program and the called program. However, two non-Ideal subprograms called by the same CA Ideal calling program should not have duplicate parameter field names because at compile time, if there is an error on one of the duplicate fields, you might not be able to tell which subprogram contains the error.

When the calling program CALL statement executes, CA Ideal moves the data from the data items named in the USING clause to the intermediate data storage area, using the rules for MOVE BY POSITION, and carrying out any necessary conversion. CA Ideal internal numeric format is converted to zoned decimal, packed decimal, or binary, depending on the non-Ideal subprogram parameter definition.

The addresses of the intermediate storage area data items are passed to the subprogram instead of the original CALL statement data items. Any reference in the non-Ideal subprogram to a parameter is actually a reference to a data item in the intermediate storage area.

When control returns from the called non-Ideal subprogram, CA Ideal then moves only those data items defined as UPDATE from the intermediate data storage area back to the original data items named in the USING clause. In this way, CA Ideal ensures that CA Ideal rules for linkage are observed.

As a result, any data items described as INPUT in the CALL statement cannot be modified by the CALL to the non-Ideal subprogram, even if the non-Ideal subprogram modified its copy of the data item. This also means that a called subprogram cannot alter the address of any updateable intermediate storage area.

The area containing the parameters is not necessarily at the same address each time the subprogram is called. Subprograms should not store the address of any parameter data between calls, as these addresses will be invalid in future calls. If a non-Ideal subprogram requires a workarea for re-entrancy, then fields within the workarea should be addressed relative to the workarea address, storing offsets rather than addresses, or relocating any pointers on each call.

Calling Non-Ideal Subprograms that Access CA Datacom/DB

This section describes how to call non-Ideal subprograms that access CA Datacom/DB.

Guidelines for Batch Programs

To indicate that the non-Ideal subprogram is mainline and CA Datacom/DB is a subroutine of the program, link edit the non-Ideal subprogram with ENTRY *pgm-name*.

In the User Requirements Table (URT) for the non-Ideal subprogram, specify OPEN=USER in the DBURINF macro.

If you are *not* sharing the URT, include OPEN and CLOSE logic in the non-Ideal subprogram. Execute the OPEN and CLOSE logic in the non-Ideal subprogram only once during the application run to reduce overhead. You can do this by including a parameter on the CALL statement in the CA Ideal program that invokes the non-Ideal subprogram. This parameter indicates whether to open, close, or update the tables in the non-Ideal subprogram. The call using the OPEN parameter is done at the beginning of the CA Ideal program. The CALL using CLOSE executes upon ending the CA Ideal program.

This type of logic makes it very easy to adapt existing COBOL programs to use with CA Ideal, although it is only one example of how you can control the OPEN and CLOSE logic.

Using IDENTIFY in z/OS Batch

A CA Ideal batch job must specify PARM='IDENTIFY' in order for the non-Ideal program to use the same DB URT and therefore a single DB task for the CA Ideal batch job.

The non-Ideal subprogram must be linked AMODE(31) and must call DBNTRY dynamically (compile option DYNAM). The CA Ideal calling program must access at least one CA Datacom/DB dataview. The SET RUN URT statement for the batch RUN must include the table to be accessed in the non-Ideal subprogram.

The following is sample JCL for the CA Ideal batch execution:

```
//xxxx     JOB xxxxxx
//PROC JCLLIB  ORDER=xxxx.proclib
//xxxx     EXEC idlbatch,PARM='IDENTIFY'
//SYSIN DD *
SEL SYS xxx
SET RUN URT dburtxxx
RUN idlpgmx
```

Guidelines for CICS Programs

It is not necessary to open and close any of the URTs in CICS programs because CA Datacom/DB Option for CICS Services takes care of this.

Accessing Same Tables in CA Ideal and Non-Ideal Programs

Accessing the same CA Datacom/DB table in a non-Ideal subprogram and the FOR construct in which the subprogram was called can have serious consequences. If both the CA Ideal calling program and the non-Ideal subprogram try to read the same set of records for update, a “deadly embrace” occurs and the job must be purged.

To access the record, the subprogram must wait until the CA Ideal calling program releases exclusive control before it can access the same record with exclusive control. But the CA Ideal program does not release exclusive control until the ENDFOR statement, which cannot execute until the call to the subprogram is complete and control returns to CA Ideal. The same situation online can result in the non-Ideal subprogram getting an CA Datacom/DB return code 18-EXCLUSIVE CONTROL DUPLICATE.

However, as long as one of the programs, either the CA Ideal calling program or the non-Ideal subprogram, is doing read-only access or each program is accessing a different table, the call can be made successfully from the CA Ideal FOR construct.

AMODE/RMODE Considerations for Non-Ideal Subprograms

Non-Ideal subprograms can be linked AMODE=31,RMODE=ANY or AMODE=24,RMODE=24.

Guidelines for Batch and Online Non-Ideal Subprograms

Since CICS linkage conventions differ from batch linkage conventions, a CA Ideal program running in CICS cannot call the same non-Ideal subprogram designed to run in batch. Likewise, a CA Ideal program running in batch cannot call a non-Ideal subprogram designed to run in CICS.

The following guidelines are separated into online and batch.

Online Non-Ideal Subprograms

COBOL, PL/I, and Assembler subprograms called by an online CA Ideal application must follow certain guidelines.

The program must be assembled or compiled in accordance with the requirements of CICS.

- Define the program name in CICS.
- In z/OS, the load library where the subprogram is located must appear in one of the DFHRPL statements.
- In VSE, the name of the core image library where the subprogram is located must appear in one of the LIBDEF PHASE,SEARCH= statements.

The following are prohibited:

- Terminal I/O.

  Note: CA Ideal loses the last message on the message line at the top of the CA Ideal screen when control returns from the non-Ideal subprogram.
- Freeing, releasing, or modifying any resource, including temporary storage that CA Ideal allocated.
- If the subprogram is called in the scope of an updated CA Ideal FOR construct, the non-Ideal subprogram cannot access the same set of CA Datacom/DB CBS, DB2, or VSAM records with update intent.
- Any scheduled abend.
- In z/OS: Any SPIE or (E)STAE macro.
- In VSE: Any STXIT macro.
- Enqueuing or dequeuing on a CA Ideal internal (system) name. See the *CA Ideal Problem Determination Guide* for information on CA Ideal internal names.
- In CICS, accessing a temporary storage area that has a name starting with X'5B'.
- In CICS, any return other than a normal EXEC CICS RETURN. COBOL programs cannot use STOP RUN, EXIT PROGRAM, or GOBACK.
- When passing return codes back to the main CA Ideal program, use the data area. If a non-Ideal subprogram returns to CA Ideal with Register 15 set to 1, 2, or 3, a runtime error with a misleading message occurs.

Note: Non-Ideal programs developed for CICS do not work in batch. Non-Ideal programs developed for batch do not work in CICS.

In addition, see the sections on restrictions for COBOL and Assembler programs in the *IBM CICS Application Programmers Reference Manual* for the appropriate environment.

Calling a CICS Subprogram

The following example illustrates how a CA Ideal program passes parameters to a CICS non-Ideal subprogram and how the subprogram, handles these parameters.

For example, a CA Ideal program defines the following fields in working data:

- ALPHA (four-character alphanumeric)
- BETA (nine-digit numeric)

The program's procedure definition includes the following CALL statement:

```
CALL COBSUB USING ALPHA,BETA
```

The COBOL subprogram's program definition includes the following parameter definition.

```
=>
=>
=>
-------------------------------------------------------------------------------
IDEAL: PARAMETER DEFINITION   PGM COBSUB (001) TEST         SYS: DOC   DISPLAY
Level Field Name                  T I Ch/Dg Occur  U   Comments/Dep on/Copy   Command
----- --------------------------- - - ----- -----  -   ---------------------- ------
===== ==============TOP========== = = ===== =====  =   ====================== ======
1     PAR-1                       X       4         U   :4 BYTES ALPHANUMERIC   000100
1     PAR-1                       U B     9         U   :BINARY FULLWORD        000200
===== ============BOTTOM ======= = = ===== =====  =   ====================== ======
```

CICS Subprogram

When calling a non-Ideal subprogram, CA Ideal places the parameter list into the TWA before the call. To access the parameters, you must establish addressability to the data with the COBOL subprogram.

The following command-level CICS COBOL II subprogram uses two parameters:

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. COB2PGM.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. IBM-370.
 OBJECT-COMPUTER. IBM-370.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 LINKAGE SECTION.
 01  TWA-LAYOUT.
    02  TWA-ADDR-1        USAGE IS POINTER.
    02  TWA-ADDR-2        USAGE IS POINTER.
 01  FIRST-PARM            PIC X(20).
 01  SECOND-PARM.
    02  ZONED-UNSIGNED    PIC 99.
    02  ZONED-SIGNED      PIC S99.
    02  PACKED-SIGNED     PIC S9(4) COMP-3.
    02  BINARY-SIGNED     PIC S9    COMP.
 PROCEDURE DIVISION.
 ADDRESS-TWA.
    EXEC CICS
        ADDRESS TWA(ADDRESS OF TWA-LAYOUT)
    END-EXEC.
    SET ADDRESS OF FIRST-PARM TO TWA-ADDR-1.
    SET ADDRESS OF SECOND-PARM TO TWA-ADDR-2.
 SET-PARM-VALUES.
    MOVE 'FROM COBOL II ' TO FIRST-PARM.
    MOVE 22 TO ZONED-UNSIGNED.
    MOVE 33 TO ZONED-SIGNED.
    MOVE 4444 TO PACKED-SIGNED.
    MOVE 1 TO BINARY-SIGNED.
 RETURN-TO-IDEAL.
    EXEC CICS
        RETURN
    END-EXEC.
```

The following illustration shows program parameter definitions for COBOL II subprogram:

```
IDEAL: PARAMETER DEFINITION  PGM COB2PGM (001) TEST
COMMAND LEVEL FIELD NAME          T I CH/DG OCCUR  U
------  ----- ------------------- - - ----- -----  -
======  ===== ======= TOP ======= = = ===== =====  =
000100  1     COB-PARM1           X      20        U
000200  1     COB-PARM2                            U
000300   2    COB-ZONE-U          U Z     2
000400   2    COB-ZONE-SIGN       N Z     2
000500   2    COB-PACKED          N P     4
000600   2    COB-BINARY          N B     1
======  ===== ===== BOTTOM ====== = = ===== =====  =
```

The following illustration shows command-level CICS Assembler subprogram, two parameters:

```
SAMPASM  DFHEIENT CODEREG= (12),DATAREG=(13),EIBREG=(11)
         B   SAMPEP
         DC  AL1(*-SAMPEP)
         DC  CÆSAMPLE CMD LEVEL ASSEMBLERÆ

*-------------------------------------------------------------------------------*
*  This is a sample assembler application that could be called from CA Ideal.     *
*  It passes two 01 level parameters.  The first parameter contains two numeric  *
*  fields that will be added together and passed back in the second parameter.   *
*                                                                               *
*    REGISTER USAGE:                                                            *
*       1 -  INCOMING PARAMETER LIST                                            *
*       5 -  BASE FOR FIRST PARAMETER                                           *
*       6 -  BASE FOR SECOND PARAMETER                                          *
*       12 - BASE FOR THIS CODE                                                 *
*       13 - CALLER'S SAVE AREA                                                 *
*       14 - RETURN ADDRESS                                                     *
*-------------------------------------------------------------------------------*
SAMPEP   DS    OH
         EXEC  CICS ASSIGN TWALENG(TWALENG)
         CLC   TWALENG,=H'0'
         BE    RETURN
         EXEC  CICS ADDRESS TWA(1)
 *-------------------------------------------------------------------------------*
*  Two parameters passed, two addresses in the TWA                              *
*-------------------------------------------------------------------------------*
         LM    5,6,0(1)
         USING PARAM1,5       Establish addressability to parms
         USING PARAM2,6
*-------------------------------------------------------------------------------*
*  The following code should be replaced with your own                          *
*-------------------------------------------------------------------------------*
         ZAP   SUM,ADD1
         AP    SUM,ADD2
*-------------------------------------------------------------------------------*
*  Return control to CA Ideal                                                    *
*-------------------------------------------------------------------------------*
RETURN   DS    OH
         EXEC  CICS RETURN
*-------------------------------------------------------------------------------*
*  PARAMETER DSECTS                                                             *
*  The following DSECT maps the first parameter                                 *
*-------------------------------------------------------------------------------*
PARAM1   DSECT
ADD1     DS    PL3
ADD2     DS    PL3
*-------------------------------------------------------------------------------*
*  The following DSECT maps the second parameter                                *
*-------------------------------------------------------------------------------*
PARAM2   DSECT
SUM      DS    PL4
*-------------------------------------------------------------------------------*
*  The following DSECT is the program's local storage                           *
*-------------------------------------------------------------------------------*
DFHEISTG DSECT
TWALENG  DS    H
         END
```

The following shows a non-Ideal subprogram parameter definition:

```
--------------------------------------------------------------------------------
IDEAL: PARAMETER DEFINITION  PGM SAMPASM (001) TEST             SYS: $ID    DISPLAY
Level Field Name          T I Ch/Dg Occur  U      Comments/Dep on/Copy      Command
===== ======= TOP ======= = = ===== =====  =      ======================  =======
1     ADDENDS                                                               000100
 2    FIRST_ADDEND        N P    5                :DS PL3 ASSEMBLER          000200
 2    SECOND_ADDEND       N P    5                :DS PL3 ASSEMBLER          000300
1     RESULT              N P    6         U      :DS PL4 ASSEMBLER          000400
===== ===== BOTTOM ====== = = ===== =====  =      ==================      =======
```

Batch Non-Ideal Subprograms

COBOL, PL/I, and Assembler subprograms that a CA Ideal application running in batch calls must follow these guidelines.

- **z/OS-**The load library where the subprogram is located must appear in one of the STEPLIB statements.
- **COBOL-**Specify the NOENDJOB compiler option.
- **VSE-**The name of the core image library where the subprogram is located must appear in one of the LIBDEF PHASE,SEARCH= statements.

The following are prohibited:

- Terminal I/O.
- Freeing, releasing, or modifying any resource that CA Ideal allocated.
- If the subprogram is called in the scope of an updated FOR construct, the non-Ideal subprogram cannot access the same set of CA Datacom/DB CBS, DB2, or VSAM records with update intent.
- Enqueuing or dequeuing on a CA Ideal internal system name. For more information about CA Ideal internal names, see the *Problem Determination Guide*.
- Any return other than a normal return. COBOL programs must use GOBACK. Assembler programs must branch to the address contained in R14 at entry. In VSE, you cannot use the EOJ macro.
- **In z/OS**-Any SPIE or (E) STAE macro
- **In VSE**-Any STXIT macro.
  - Any scheduled abend
  - Attempting to read the primary input file:

    In z/OS, SYSIN

    In VSE, SYSIPT

Calling COBOL in z/OS Batch

The following sample z/OS JCL sets the COBOL II runtime option RTEREUS, which improves the performance of COBOL II programs that CA Ideal batch calls and preserves the working storage between calls of the COBOL II subprogram.

```
//jobname JOB ....
//* ------------------------------------------------------- *
//* JCL TO ASSEMBLE THE COBOL II RUNTIME OPTION MODULE FOR   *
//* USE WITH IDEAL/COBOL II BATCH SUBPROGRAMS.              *
//* -------------------------------------------------------*
//ASSEMBLE EXEC PGM=IFOX00,PARM='DECK',COND=(0,NE),
 //         REGION=1024K
//SYSLIB   DD  DSN=SYS1.MACLIB,DISP=SHR
//         DD  DSN=MVSSYS.COB2.V1R4M0.COB2LSRC,DISP=SHR
//SYSUT1   DD  DSN=&&SYSUT1,UNIT=VIO,SPACE=(1700,(600,100))
 //SYSUT2   DD  DSN=&&SYSUT2,UNIT=VIO,SPACE=(1700,(300,50))
 //SYSUT3   DD  DSN=&&SYSUT3,UNIT=VIO,SPACE=(1700,(300,50))
 //SYSPRINT DD  SYSOUT=*,DCB=BLKSIZE=1089
//SYSPUNCH DD  DSN=object.library.name(IGZEOPT),DISP=SHR
//SYSIN DD *
      IGZOPT SYSTYPE=OS,RTEREUS=YES
      END
/*

//jobname JOB ...
//* ------------------------------------------------------- *
//* JCL TO COMPILE & LINK EDIT A COBOL II PROGRAM WHICH IS   *
//* TO BE CALLED BY A CA Ideal BATCH PROGRAM.                *
//* NOTE THE COBOL II AND LINK-EDIT PARAMETERS AND THE       *
//* LINK-EDIT INCLUDE STATEMENT FOR IGZEOPT (WHICH MUST      *
//* BE ASSEMBLED WITH RTEREUS=YES)                           *
//* ------------------------------------------------------- *
//COB     EXEC PGM=IGYCRCTL,
 //             PARM='APOST,NOOPT,RES,LIB,DATA(24),RENT'
//STEPLIB  DD  DSNAME=MVSSYS.COB2.V1R4M0.COB2COMP,DISP=SHR
//SYSLIB   DD  DSN=program.source.library,DISP=SHR
//SYSLIN   DD  DISP=(,PASS),
 //             UNIT=SYSDA,SPACE=(TRK,(5,5))
 //SYSPRINT DD  SYSOUT=*
//SYSUT1   DD  UNIT=SYSDA,SPACE=(CYL,(1,5))
 //SYSUT2   DD  UNIT=SYSDA,SPACE=(CYL,(1,5))
 //SYSUT3   DD  UNIT=SYSDA,SPACE=(CYL,(1,5))
 //SYSUT4   DD  UNIT=SYSDA,SPACE=(CYL,(1,5))
 //SYSUT5   DD  UNIT=SYSDA,SPACE=(CYL,(1,5))
 //SYSUT6   DD  UNIT=SYSDA,SPACE=(CYL,(1,5))
 //SYSUT7   DD  UNIT=SYSDA,SPACE=(CYL,(1,5))
 //COB.SYSIN DD *
INC cobol2pg
/*
//* ------------------------------------------------------- *
//LKED   EXEC  PGM=IEWL,
 //             COND=(5,LT),
 //             PARM='AMODE(24) RMODE(24) MAP'
//SYSPRINT DD  SYSOUT=*
//SYSLIB   DD  DSN=MVSSYS.COB2.V1R4M0.COB2LIB,DISP=SHR
//OBJLIB   DD  DSN=object.library.name,DISP=SHR
//SYSLMOD  DD  DSN=target.library(cobol2pg),DISP=SHR
//SYSUT1   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
 //SYSLIN   DD  DDNAME=SYSIN
//SYSIPT DD DSN=*.COB.SYSLIN,DISP=(OLD,PASS)
 //SYSIN DD *
       INCLUDE OBJLIB (IGZEOPT)
       INCLUDE SYSIPT
          NAME cobol2pg(R)
```

Calling COBOL in VSE Batch

The following sample VSE JCL sets the COBOL II runtime option RTEREUS, which improves the performance of COBOL II programs that CA Ideal batch calls and preserves the working storage between calls of the COBOL II subprogram.

```
* $$ JOB JNM=IDLCOBII,CLASS=B,PRI=6,DISP=D
* $$ LST DISP=D,CLASS=L
// JOB    IDLCOBII
// OPTION CATAL
   PHASE  IDLCOBII,*
// EXEC   IGYCRCTL,SIZE=256K
 CBL LIB,RENT,RES,NODYNAM
.... COBOL II SOURCE GOES HERE.
/*
// EXEC   ASSEMBLY
         IGZOPTV RTEREUS=YES
         END
/*
// LIBDEF PHASE,CATALOG=CAI.USER
// EXEC LNKEDT
/&
* $$ EOJ
```

Make sure that the phase is available to your CA Ideal batch JCL LIBDEF statement. You can now call the program directly from a CA Ideal batch program.

Calling a PL/I Subprogram

A PL/I subprogram is called from an Assembler program so the normal parameter lists are not constructed.

PL/I subprograms that a CA Ideal application calls must follow these guidelines:

- You must identify PL/I non-Ideal subprograms with a language of PL/I, PLI, PL/1, or PL1 on the program IDEntification screen.
- If you specify OPTIONS(MAIN), you must use entry point PLICALLA. If program is not MAIN, you must link edit to it and call it from MAIN program.
- The parameters passed to the PL/I program from CA Ideal follow assembler linkage, not PL/I. The PL/I programs should code the parameters as follows. You can reference them. Arrays make no difference in the way the variables are coded.

Character String/Structures-The following coding allows access. You must refer to the structure as a based variable, but any data type is allowed in the structure.

```
USERPGM:  PROCEDURE (P1,P2)REORDER;
DCL P1              FIXED BIN(31);
DCL P2              FIXED BIN(31);
DCL PTR1            PTR;
DCL PTR2            PTR;
     PTR1 = ADDR(P1);
     PTR2 = ADDR(P2);
DCL  PARM1          CHAR(xx) BASED(PTR1);   /* Character String */
DCL1 PARM2          BASED(PTR2),           /* Structure */
     3  NAME        CHAR(xx),
     3  SSN         FIXED DEC(x)
     3  EMPLOYE#    FIXED BIN(31);
```

Numeric-The following coding allows access for the numeric types of half and fullword binary, fixed decimal, and zoned decimal.

```
USERPGM: PROCEDURE (FIX_BIN,FIX_DEC,ZONED) REORDER;
DCL FIX_BIN     FIXED BIN(xx);   /* Half of Fullword Binary */
DCL FIX_DEC     FIXED DEC(x);    /* Fixed Decimal */
DCL ZONED       PIC '(x)9';      /* Zoned Decimal */
```

Chapter 4: Performing Calculations

Introduction

For information about performing calculations on ADD, SUBTRACT, and SET statements, see the *Programming Reference Guide.*

Statements	Functions
ADD	$ABS
SUBTRACT	$NUMBER
SET	$NUMERIC $REMAINDER $ROUND $SQRT

In addition, see the topics Condition and Arithmetic Expression in the *Programming Reference Guide*.

Optimizing Arithmetic in CA Ideal

By far, the largest proportion of a typical business application is spent in database retrieval and formatting data for screens and reports. However, if you are doing a great deal of computation in an application, you want to make sure that it is done in the most efficient way possible.

Here are some things you can do to optimize arithmetic in CA Ideal:

- Use packed decimal format. This is the CA Ideal default and, as the test results show, the most efficient for CA Ideal computations.
- Where possible, make sure operands in complex and frequently used arithmetic expressions have the same decimal alignment (the same number of decimal positions). Adding two numbers with Char/Digits of 5.2 is faster than adding a number with 5.1 to a number with 5.2.

- When you use dataview fields (columns) repeatedly in calculations, move them to working data first and do the computations from there. This is because CA Ideal continually checks the dataview fields to ensure that they are actually numeric (since the values are not guaranteed correct), whereas CA Ideal knows that working data fields are always valid and does not keep checking. When you move the dataview fields to working data, the validity check is done just once, so it really does not save anything if you only reference a given dataview field once per record.

  You can use the COPY DATAVIEW clause in working data to define a structure that matches the dataview. Then use a MOVE...BY NAME statement to move the dataview fields.

- If you use a subscripted field repeatedly in a calculation, move it to a separate non-occurring work field in working data and use it in the computation. This is a standard source optimization technique. It applies to any language, not just to CA Ideal.

- Ensure that computations are not done superfluously in a loop. If a computation is in a loop, make sure that it really needs to be there. Otherwise, move it outside the loop. Again, this is a standard optimization technique.

- If a parameter field is defined as dynamic and is used repeatedly in a computation, move it to working data first.

- If a field is redefined or is itself a redefinition and is used repeatedly in a computation, move it to a separate working data field first. This is because REDEFINES means that the contents of either field-the REDEFINES subject or object-can be made invalid by an operation on the other, so CA Ideal always validates REDEFINES subjects and objects before computations.

- In general, CA Ideal arithmetic is most efficient when the fields are in working data, unsubscripted, not part of a redefinition, are packed decimal format, and have identical decimal alignment. If a given field that is used more than once in a computation fails one of these criteria, then moving it to a separate work field that does comply probably improves efficiency. However, if the field is only used once, it is probably better to let CA Ideal handle the conversion internally.

- To the extent that you have the choice, external data (dataview fields) benefit from following these guidelines too-but not at the expense of good database design.

  For example, if you have a field that is a database key and you have to choose between the best format for database retrieval or for internal computation, choose the best database retrieval format (although this might depend on the specific case).

- In certain cases, where you have a really heavy-duty computation (for example, you are generating a mortgage rate book and calculating monthly payment amount for each combination of sale price, interest rate, and number of payments), you might consider calling a COBOL or Assembler subroutine to do the calculation. Carefully evaluate the overhead since you must weigh the overhead of calling the subroutine against whatever arithmetic overhead you would save.

A word, finally, about some additional features of arithmetic processing in CA Ideal.

- CA Ideal ensures that abends do not occur. For example, a SOC7 abend cannot occur.
- CA Ideal checks for overflow and subscript range automatically.
- Programming arithmetic computations using CA Ideal saves debugging and dump reading time and the time spent in programming and maintenance.

Chapter 5: Using Functions

This chapter presents lists of functions. For information about the specific functions, see the *Programming Reference Guide*.

Date Functions

The following functions are date functions:

$DATE
$DAY
$INTERNAL-DATE
$MONTH
$TIME
$TODAY
$VERIFY-DATE
$WEEKDAY
$YEAR

Error Functions

The following functions are error functions:

$ERROR-CLASS
$ERROR-CONSTRAINT-NAME
$ERROR-DESCRIPTION
$ERROR-DVW-DBID
$ERROR-DVW-INTERNAL-STATUS
$ERROR-DVW-STATUS
$ERROR-NAME
$ERROR-PGM
$ERROR-PROC
$ERROR-STMT
$ERROR-SUBSCRIPT
$ERROR-TYPE
$ERROR-VALUE
$PANEL-ERROR
$PANEL-FIELD-ERROR
$RETURN CODE
$SUBSCRIPT-POSITION

Numeric Functions

The following functions are numeric functions:

$ABS
$NUMBER
$REMAINDER
$SQRT
$COUNT
$NUMERIC
$ROUND

Panel Functions

The following functions are panel functions:

$CURSOR
$KEY
$PANEL-FIELD-ERROR
$PF
$EMPTY
$PANEL-ERROR
$PANEL-GROUP-OCCURS
$RECEIVED

String Functions

The following functions are string functions:

$ALPHABETIC
$EDIT
$FIXED-MASK
$HIGH
$LENGTH
$LOW
$SPACES
$SUBSTR
$TRIM
$CHAR-TO-HEX
$EMPTY
$HEX-TO-CHAR
$INDEX
$PAD
$STRING
$TRANSLATE
$VERIFY

System Functions

The following functions are system functions:

$ACCOUNT-ID
$ENTER-KEY
$FINAL-ID
$NETWORK-ID
$PLAN
$RECEIVED
$TERMINAL-ID
$USER-ID
$CURRENT-TRAN-ID
$ENVIRONMENT
$INIT-TRAN-ID
$OP-SYSTEM
$PACKAGESET
$SQL
$TRANSACTION-ID
$USER-NAME

Chapter 6: Error Handling

This chapter presents information about handling errors. CA Ideal provides facilities for evaluating and resolving runtime errors. There are two main categories of errors:

- Errors due to code that does not process the data properly
- Errors caused by unpredictable conditions at runtime

Unpredictable errors cause an application to abend. This is somewhat different from logic errors that generally cause unexpected results, although those results can lead to an abend. The following section assumes that logic errors are handled by good design and adequate testing.

Preventing Errors

Code all programs with careful consideration to potential runtime errors. The errors most frequently encountered involve improper handling of data. These errors are attempts to assign a value that is of an inappropriate data type, format, or size to a column or field. You can test the values thoroughly and manipulate them before making an assignment using the comprehensive set of functions in the CA Ideal Procedure Definition Language (PDL).

The following are simple examples of functions that ensure that the data assigned to a column conforms to the column attributes:

The $ROUND function assigns a numeric value with two decimal positions to the field OPEN$:

```
SET OPEN$ = $ROUND(value,FACTOR = .01)
```

- The $SUBSTR function assigns a two-character fixed-length value to STATE:

```
SET STATE = $SUBSTR(value,START = 1,LENGTH = 2)
```

- The $VERIFY function tests a value to ensure that the value assigned to CUSTID is an alphanumeric value containing only uppercase alphanumeric characters and all digits. In this code segment, the subprocedure BADVALUE executes when another character is encountered.

```
IF $VERIFY(value,AGAINST = (UCALPHA,NUMERIC))
    SET CUSTID = value
ELSE
    DO BADVALUE
ENDIF
```

- The $DATE function formats the current date value contained in the $TODAY function to the format required for ACTDT. The function $TODAY returns the current date as *mmddyy*, thus the $DATE function reformats the date value.

```
SET ACTDT = $DATE('YYMMDD',DATE = $TODAY)
```

For more information about CA Ideal functions, see the *Programming Reference Guide*.

Using $RC in Error Procedures

The CA Ideal system does not set $RC for the current error. If you do not code an error procedure, the default error procedure ensures that $RC is at least a value of 12. A user coded error procedure needs to set $RC to ensure a non-zero return code upon completion of the RUN. Setting $RC is *optional*. Whether it is set in an error procedure or not, the CA Ideal system does not alter it.

$RC typically tests the completion status after a run. Therefore, it is most useful for batch jobs, where the execution of one run depends on the successful execution of a previous one. For more information about using $RC to control batch runs, see the section, Using CA Ideal Commands in Batch, in the *Working in the Environment Guide*.

Handling Runtime Errors

You cannot prevent certain runtime error conditions, but you can detect them during execution. CA Ideal provides two facilities for handling runtime errors:

- The error procedure executes automatically when an error is encountered. You can use the default error procedure that CA Ideal provides or you can code your own error procedure. However, you can include only one error procedure in any program.
- The WHEN ERROR clause in the FOR construct executes automatically when a dataview error ($ERROR-CLASS of DVW) is encountered while processing the FOR construct.

The $ERROR functions are only available in the scope of the error procedure or the WHEN ERROR clause.

Default Error Procedure

By default, CA Ideal provides the following error procedure. It executes when a runtime error is encountered. This procedure includes:

```
<<ERROR>> PROCEDURE
    IF $RC LT 12
        SET $RC EQ 12
    ENDIF
    LIST ERROR
    BACKOUT
    QUIT RUN
ENDPROC
```

In other words:

SET $RC EQ 12 Sets the return code to specify a fatal error.

LIST ERROR Transmits the type of error to output file.

BACKOUT Ensures that possibly incomplete updates due to the error are not applied.

QUIT RUN Terminates the application.

The LIST ERROR statement writes the values of the functions $ERROR-TYPE, $ERROR-CLASS, $ERROR-DVW-STATUS, and $ERROR-DVW-INTERNAL-STATUS and the statement number to an external file. The various $ERROR functions return the value to help to identify the specific error.

By default, a message displays at the screen when a run is terminated. This message states:

```
RUN completed, RC = nn
```

The value of *nn* specifies the return code value.

Coding an Error Procedure

Errors are not always fatal. They can provide information about the error. In certain instances, the program can recover from the error and continue processing. To handle these situations, you can code an error procedure in the program.

You must name the coded error procedure ERROR. The DO statement can invoke it; however, it cannot be the object of a QUIT or PROCESS NEXT statement. The coded error procedure automatically executes when an error occurs. If the coded error procedure itself contains an error, the default error procedure executes. If a fatal error occurs, the coded procedure is ignored and the default error procedure executes.

A coded error procedure can take advantage of PDL statements to inform you of an error and maintain control. The NOTIFY statement can override the default termination message and provide a more informative message. The NOTIFY message displays with the next panel that is transmitted or at the termination of the run. Rather than terminate an entire run, the coded procedure can terminate just the current program.

For example, an error procedure can send an informative message to the screen and to the output file:

```
<<ERROR>> PROCEDURE
     LIST ERROR
     BACKOUT
     SET $RC = 12
     SET WOR-NUM = $RC
     NOTIFY 'PROGRAM ABENDING - ' WOR-NUM
     QUIT PROGRAM
 ENDPROC
```

You can create a panel to provide more detailed error information. The panel can include explanatory text, remedial information, and the $RC value.

When a user-defined error procedure is invoked, you can set the value of the return code in that error procedure. The system does not reset the value.

At the end of the run, the value of $RC displays. This way, even a run that terminates normally can indicate an error during processing.

Categorizing Errors

Functions are available to determine the category of the error that was encountered. For example, $ERROR-CLASS returns the classification of the error, such as NUM for numeric, DVW for dataview, PGM for program, SYS for system, and FTL for fatal. SYS (system) and FTL (fatal) errors always execute the default error procedure and terminate the run.

The following example tests $ERROR-CLASS for NUM. NUM is set when a numeric field contains an invalid numeric value. If $ERROR-CLASS is NUM, the return code is set to 8. Since recovery is possible, the program continues with the next iteration of a processing loop.

```
<<ERROR>> PROCEDURE
   SELECT $ERROR-CLASS
     WHEN 'NUM'
        SET $RC = 8
        PROCESS NEXT MAIN-LOOP
     WHEN OTHER
        SET $RC = 12
   ENDSELECT
   LIST ERROR
   BACKOUT
   QUIT RUN
ENDPROC
```

Common Error Subroutines

Most CA Ideal programmers are familiar with the <<ERROR>> procedure as a means of producing information for the end-user when a processing error occurs. A large number of sites have attempted to standardize this processing by using a common subprogram to display and record the error information.

There are two problems that generally arise from the use of a common subprogram:

- First, by simply being a common subprogram, it participates in a very high number of PGM-PGM-CALL relationships in CA-DataDictionary. Changes to the common program result in a very large number of dictionary calls when you issue a CA Ideal DUPLICATE or MARK STATUS command. This can make it necessary to run overnight batch jobs to do the DUPLICATE or MARK when you change the common subprogram.
- Second, the subprogram is usually called directly from the <<ERROR>> procedure, which can cause anomalies in its processing. When the <<ERROR>> procedure is entered, CA Ideal processes all subsequent statements in an error mode until the QUIT or PROCESS NEXT that leaves the scope of the procedure. A DO or CALL to a subordinate procedure or program is still in error mode. This prevents error recursion, where errors in the <<ERROR>> procedure could make the application loop endlessly. The effects of being in this error mode are often subtle, but an obvious one is that a called subprogram cannot handle errors in its own code. This means, for example, that a common error subprogram that reads message text from a database table cannot handle database errors.

There is a single solution to both problems: A horizontal calling structure, where a dispatcher program calls all the processing programs according to the value of a control variable. The same dispatcher can invoke the common error subprogram with the following advantages:

- The <<ERROR>> procedure of the failing program sets return variables to describe the error and quits the program. This leaves error mode and allows the error handler full recovery of its own errors.
- The error subprogram is called only by dispatcher programs, which are much fewer in number and eliminate the overworking of the relationship file in the dictionary. The nesting depth of the application is reduced by one, which saves storage and reduces the complexity of the calling structure.

Detecting the Severity of the Error Using $RC

In most cases, the execution of CA Ideal commands sets $RC to a specific value. The CA Ideal system sets $RC, except when a RUN command is invoked. When the CA Ideal system determines it, $RC reflects the highest severity of errors in a session as follows:

Error Type	Return Code Value
No error	0
Advisory or warning	4
Error	8
Fatal error	12 or more

The value of $RC is transmitted to the output file at the end of every run. You can evaluate this value to determine the outcome of the run. The default error procedure sets the value of $RC to 12. A coded error procedure can set the value of $RC as appropriate. To determine what value to assign to $RC, several functions are available. You can use these functions to define category, type, and severity of the error.

During online execution, the value of $RC should be initialized to 0 when an application is invoked.

Using $RC in Batch

During online execution, the value of $RC should be initialized to 0 when an application is invoked. Batch execution is different. Frequently, the execution of one program is based on the successful execution of another. In that case, the value of $RC should be retained from one run to the next. The SET command controls whether the value of $RC is retained. This command must execute before invoking the application.

```
SET RUN $RC ZERO   return code reset to 0 at start of a run
SET RUN $RC KEEP   return code retained
```

When executing batch programs, you can test the return code in the job stream using CA Ideal IF, ELSE, and ENDIF commands.

Coding for Multiple Errors

Although you can code only one error procedure in the program, that single procedure can evaluate and handle several types of errors. The following code segment uses a SELECT construct to distinguish error handling for several errors:

```
<<ERROR>> PROCEDURE
     SELECT $ERROR-CLASS
        WHEN 'DVW'
           SET $RC = 1660
           LIST $ERROR-DVW-DBID
           LIST $ERROR-DVW-STATUS
           LIST $ERROR-DVW-INTERNAL-STATUS
        WHEN 'NUM'
           SET $RC = 8
           PROCESS NEXT MAIN-LOOP
        WHEN OTHER
           SET $RC = 12
     ENDSELECT
     SET WOR-NUM = $RC
     NOTIFY 'Program Abending - ' WOR-NUM
     LIST ERROR
     BACKOUT
     QUIT RUN
 ENDPROC
```

In this example, if $ERROR-CLASS is NUM, the program can recover; otherwise, the program terminates. The value of the $RC is either 1660, indicating a dataview error, or 12, indicating some other error. By testing for other $ERROR-CLASS values, you can set the $RC to reflect the error condition encountered.

Evaluating Specific Errors

You can evaluate errors even further. For example, $ERROR-CLASS of DVW results when a dataview access error occurs. The $ERROR-TYPE function returns more specific information. For example, a $ERROR-TYPE value of DVW reflects an error in accessing CA Datacom/DB.

The $ERROR-DVW-STATUS function returns a value only when the $ERROR-TYPE is DVW; offering more information on the type of error in a CA Datacom/DB environment. For example, I3 indicates that a row integrity error occurred. Another user modified the row being processed.

Coding for Dataview Errors

You can detect dataview errors and perform troubleshooting error procefures to detect it.

Handling Errors in the FOR Construct

You can detect dataview errors and perform error processing in the FOR construct by coding a WHEN ERROR clause. Statements specified in the WHEN ERROR clause can access $ERROR functions and should resolve the error with a PROCESS NEXT or DO ERROR statement. If processing falls through to the ENDFOR, the $ERROR functions are no longer available.

In the following example, the WHEN ERROR statement evaluates the error condition when a dataview error occurs.

```
FOR FIRST CUSTOMER
  WHERE CUSTID = PNL-CUST
    DELETE CUSTOMER
WHEN NONE
  NOTIFY 'NO CUSTOMERS FOUND'
WHEN ERROR
  SELECT FIRST ACTION
    WHEN $ERROR-DVW-STATUS = 94 AND
         $ERROR-INTERNAL DVW-STATUS = 31
       LIST 'Constraint Error: ' $ERROR-CONSTRAINT-NAME
       NOTIFY 'Customer ' CUSTID 'has open orders and cannot be deleted'
    WHEN $ERROR-DVW-STATUS = 36
        NOTIFY 'Contact Database Administrator with error information'
    WHEN OTHER
        DO ERROR
  ENDSEL
ENDFOR
```

The values of the $ERROR functions contain different values, depending on the type of error and the type of dataview. For more information about $ERROR functions, see the *Programming Reference Guide*.

Handling Numeric Errors

PDL error handling functions return data identifying the cause of a runtime error in an error procedure. If you use an error handling function other than in an error procedure, N/A is returned.

When an application moves non-numeric data into a numeric field, a numeric error occurs. $ERROR-CLASS and $ERROR-TYPE return the value NUM. $ERROR-NAME returns the name of the sending field in the MOVE. There are several ways to determine the value of the field.

$ERROR-VALUE returns a string showing the value of the field. A question mark (?) marks each byte containing non-numeric data.

For example, if the string 12A45B was moved into a numeric field and the error procedure was invoked, then $ERROR-VALUE contains 12?45?, indicating the positions of the letters A and B.

For a program to see what the actual data was, you can use the $CHAR-TO-HEX function on the sending field in error. For example:

```
<<ERROR>> PROCEDURE
IF $ERROR-TYPE = 'NUM'
   SELECT $ERROR-NAME
        WHEN 'field-name-1'
                SET VAR1 = $CHAR-TO-HEX(field-name-1)
        WHEN 'field-name-2'
                SET VAR2 = $CHAR-TO-HEX(field-name-2)
   ENDSEL
ENDIF
ENDPROC
```

You must include a WHEN clause for every possible sending field. You can directly list the contents without using an intermediate field, for example:

```
LIST $CHAR-TO-HEX(field-name-1)
```

You can also issue a LIST ERROR statement in the scope of the error procedure. The resulting listing displays the contents of the field in error in hex, although the results are not of use in the error procedure.

You can help prevent numeric runtime errors by using the $VERIFY function to make sure that each sending field is numeric before doing a move.

Executing the Error Procedure for User-Determined Errors

You can execute the error procedure when program code determines that an error condition exists, even if the condition is not one that would cause a fatal error. To execute the error procedure, enter the statement DO ERROR.

The following example shows an error procedure being invoked:

```
<<COLOR_CHECK>> PROCEDURE
   SELECT WOR-COLOR
   WHEN 'BLUE'
      DO PROCESS-BLUE
   WHEN 'RED'
      DO PROCESS-RED
   WHEN 'YELLOW'
      DO PROCESS-YELLOW
   WHEN OTHER
      DO ERROR
  ENDSEL
ENDPROC
```

Using SQLCA for SQL

When the FOR construct accesses SQL objects, the CA Ideal error procedure processing is invoked. If the $ERROR-TYPE value is DB2 or SQL, you can evaluate the SQLCODE from the SQLCA with the CA Ideal error procedure. For example, an error procedure for DB2 can contain:

```
SELECT $ERROR-TYPE
    WHEN 'DB2'
        SET $RC = $SQLCODE
        LIST ERROR
        LIST $SQL-LAST-STMT
        BACKOUT
        QUIT RUN
    . . .
```

$SQL-LAST-STMT returns the statement number and program name of the last statement executed.

When using SQL to access the database, be aware that errors encountered in SQL do not invoke execution of the CA Ideal error procedures. Code the necessary SQL, as in:

```
EXEC SQL
   WHENEVER SQLERROR DO SQL-ERROR
END-EXEC
```

You can code the error procedure named SQL-ERROR as:

```
<<SQL-ERROR>> PROCEDURE
    SET $RC = $SQLCODE
    LIST $SQL-LAST-STMT
    BACKOUT
    QUIT RUN
ENDPROC
```

Locating the Error in the Code

The following $ERROR functions are particularly useful in an error procedure designed to process all of the errors in an application. These $ERROR functions pinpoint the location of the error:

- **$ERROR-NAME** Name of error field
- **$ERROR-PGM** Name of program containing error
- **$ERROR-PROC** Name of procedure containing error
- **$ERROR-STMT** Sequence number of statement containing error

Chapter 7: Processing Programs

This chapter describes the facilities of CA Ideal available for processing a program. The PROCESS option of the Main Menu takes you directly to the Process Program menu, where six options-COMPILE, RUN, DEBUG, SUBMIT, EXECUTE, and PRODUCE-are available.

```
 =>
--------------------------------------------------------------------------------
 IDEAL: PROCESS PROGRAM  (001) TEST                      SYS: DOC  MENU

 Enter desired option number ===>        There are    6  options in this menu:

 1.  COMPILE             - Compile a program
 2.  RUN                 - Run a program online
 3.  DEBUG               - Debug a program online
 4.  SUBMIT              - Submit a member containing a batch jobstream
 5.  EXECUTE             - Execute a member containing IDEAL commands
 6.  PRODUCE             - Produce a report facsimile
```

Compiling a Program

Compiling a program results in a form of the program ready for execution (object form) and a compilation listing.

You can compile a program by issuing a COMPILE command in the command area during an online session or by submitting a member containing a batch job stream. This job stream can contain a COMPILE command. You can submit it during an online session.

You can direct the output of an online compilation, the compilation listing, to the output library for browsing online or to a system printer. You can also direct an online compilation to a network printer. This chapter describes how to compile a program online.

Note: When you recompile a program, only the working data, parameters, and panels that were changed since the program was last compiled are recompiled.

Using the COMPILE Command

Use the CA Ideal COMPILE command to compile a program. To access the COMPILE prompter, select option 1 on the Process Program menu or enter an incomplete COMPILE command. COMPILE * compiles the current program.

You can compile only test programs in the current system. If you enter the COMPILE command for a production program, PROD status, the program is not recompiled, but a compilation listing is generated.

When you enter a COMPILE command from CA Ideal, a message COMPILATION INITIATED appears. You can proceed with other activities at the terminal while the program is compiling. CA Ideal allows a maximum of 16 compilations to be in progress simultaneously for a user. For your system, an installation parameter determines the number of compilations that can execute simultaneously. If the limit is reached, new compilation requests are entered into a queue and then compiled. For more information, see the *Installation Guide*.

Issuing a COMPILE Command in CICS

A compile initiated in CICS can produce a compilation listing directed to the output library or to a system or network printer. The default destination for output from an online compilation is the output library, where you can browse the output before you print it. Since online compilation of large applications (those with many report or panel definitions or a large procedure) could tie up much of the available resources, consider submitting compilations in batch with the compilation listing directed to the output library. For more information, see Batch Compilation.

One of the following messages informs you when compilation ends:

```
COMPILATION SUCCESSFUL
COMPILATION UNSUCCESSFUL, PROGRAM HAS ERRORS
COMPILE HAS FAILED
```

If the compilation was successful, the compilation produces a form of the program ready for execution, known as the object program, and a compilation listing. At this point, the program can run. For more information, see the Executing a Program.

If the compilation is unsuccessful, you can display either the compilation errors listing, including the errors listed at the end, or the procedure definition, with highlighted errors as an optional feature. Keep in mind, however, that there could be errors in any component of an application program, not just in the procedure. It is also possible to view both the compilation listing and the procedure definition by using a split screen. For more information on displaying output and splitting the screen, see the *Working in the Environment Guide*.

Example

To compile program DEMO1, enter:

```
COMPILE DEMO1
```

Assume that the compilation was unsuccessful and the message indicating that the program has errors appears. You can display the output and edit the corresponding section of the procedure definition by entering the following CA Ideal commands:

```
SPLIT
1 EDIT PROGRAM DEMO1 PROCEDURE
2 DISPLAY OUTPUT DEMO1
```

Then scroll to the errors given at the bottom of the compilation listing.

In the second region of the screen, the errors noted in the compilation listing are shown, while in the first region, the corresponding procedure statements are shown. Splitting the screen this way lets you proceed with:

```
1 EDIT *
.
.
.
1 COMPILE *
1 RUN *
```

Batch Compilation

In batch, compilation occurs when a COMPILE command is invoked. In CA Ideal, job streams are stored. You can submit them from a member containing the JCL for batch CA Ideal. Submitting a batch job stream is described in the *Working in the Environment Guide*. The procedure name IDBATCH can change from site to site. See your CA Ideal administrator for procedure names at your site.

The following are sample job streams for compiling a program in Z/OS and VSE:

z/OS Job Stream

```
//COMP1 JOB . . .
//BATCH EXEC IDLBATCH,PARM='NOPRINT'
//IDEAL.COMPLIST DD SYSOUT=A
//IDEAL.SYSIN DD *
PERSON userid PASSWORD password
SELECT SYSTEM DOC
                                    {  LIB     }
COMPILE COMP1 VERSION 1 DESTINATION { SYS name }
                                    {          }
OFF
```

VSE Job Stream

```
* $$ JOB JNM=IDBATCH,PRI=n,USER='username',DISP=D
* $$ LST DISP=x,CLASS=x,LST=cuu...
// JOB IDBATCH
// OPTION LOG,NODUMP
// EXEC PROC=IDLPROC,PARM='NOPRINT'
*
// EXEC IDBATCH,SIZE=80K
PERSON userid PASSWORD password
SELECT SYSTEM DOC
                                    {  LIB     }
COMPILE COMP1 VERSION 1 DESTINATION { SYS name }
OFF                                 {          }
/*
// EXEC LISTLOG
/*
/&
* $$ EOJ
```

Note: JCL statements for work files and sort work files are necessary for compiles executed with the XREF=FULL or SHORT options.

In the job stream examples above, the DESTINATION clause of the COMPILE command is the output library or a system printer. You can monitor the status of a compilation directed to the output library by displaying the output library. As soon as the compile starts to execute, its status in the library is CRTIN. When the status is READY, the compile has finished executing.

How to Read a Compilation Listing

The information in the compilation listing is output in the sequence listed below. For more information and a complete example, see the Sample List appendix.

Compile Options in Effect

Displays available compile options for processing and listings. For each option, Yes signifies that the option is on, No signifies that the option is off. For more information, see the description of the SET COMPILE command in the *Command Reference Guide*.

Program Identification Display

Displays descriptive information about the program's creation, last edit access, and last compilation. The run status and a short description of the program also display. For more information, see Creating a New Program Definition.

Resource Listing

Lists all external components accessed by the procedure, namely dataviews, panels, reports, and other called programs. For more information, see Defining Program Resources.

Dataview Listing

Displays the components of each dataview accessed by the procedure. For more information, see the *Creating Dataviews Guide*.

Panel Identification

Displays descriptive information about the panel's creation and last edit access. A short description of the panel also displays. For more information, see the *Creating Panel Definitions Guide*.

Panel Layout

Displays a panel representation. For more information, see the *Creating Panel Definitions Guide*.

Panel Facsimile

Displays a panel as it appears when a program is run. The symbols described in Panel Layout do not appear on this screen. For more information, see the *Creating Panel Definitions Guide*.

Field Summary Table

Displays a table that describes each field identified on the panel. For more information, see the *Creating Panel Definitions Guide*.

Panel Parameters

Displays input and output fill characters, panels used as help, prefix or suffix panels, and other general options for panel definition. For more information, see the *Creating Panel Definitions Guide*.

Input Rules

Displays a table that describes the input each field accepts. This information is also found in the Extended Field Definition. For more information, see the *Creating Panel Definitions Guide*.

Output Rules

Displays a table that describes the output each field displays. This information is also found in the Extended Field Definition. For more information, see the *Creating Panel Definitions Guide*.

Working Data

Displays data that is local to the program. For more information, see Defining Working Data.

Parameters

Displays a listing of names and descriptions of data items that are passed to the program from the calling program. You can also define a parameter in a main (calling) program if it is issued in a RUN command for that program. For more information, see Defining Parameters Used as Input.

Report Identification

Displays descriptive information about the report's creation and last edit access. A short description of the report also displays. For more information, see the *Generating Reports Guide*.

Report Parameter

Displays report layout options such as a report's length and width on a page, the spacing between lines and columns, column headings and how they are highlighted, control breaks, group continuation indication, heading definitions, summary information, and date and page specification. For more information, see the *Generating Reports Guide*.

Report Page Heading Definition

Displays the page heading used in the report. The position or location of the page heading is specified. You can use field names, literals, functions, and arithmetic expressions as part of a page heading. For more information, refer the *Generating Reports Guide*.

Report Detail Definition

Displays the fields to appear in each detail line of the body of the report and specifies sorting, control breaks, summary functions, and so on for these fields. For more information, see the *Generating Reports Guide*.

Procedure

Displays a listing of program logic for the application. For complete syntax of all the PDL statements, see the *Programming Reference Guide*.

SQL Generated Listing

Displays SQL generated to support FOR constructs for the current compile. For more information on the SET COMPILE LSQL command, see the *Command Reference Guide*.

Cross Reference Listing (Batch)

Displays a listing in ascending sequence of the symbols from components the procedure accesses. This listing appears when it is the site default or when the SET COMPILE REF command equals FULL (this is the default). For more information, see the next section.

Compile Cross Reference (Batch)

The compile cross reference listing displays the symbols accessed in the procedure. It is included in batch compilation output when it is the site default or when the command SET COMPILE REF is FULL or SHORT. For an example, see appendix Sample Lists.

The following information displays in the compile cross reference listing:

- **symbol** Any name that is identified in the procedure or resource fill-in of the program or defined in working data, parameter data, or panels the program accesses.
- **For CA Datacom SQL access**, the name of a table or view. They can be used explicitly in embedded SQL or generated from a FOR construct.
- **For DB2 SQL access**, the name of a table, view, or column, or a correlation name. They can be used explicitly in embedded SQL or generated from a FOR construct.
- **qualifier** Any group, panel, or dataview name that qualifies the symbol if there is a qualifier. This is left blank for level-1 items, report names, subprogram names, dataview names, and panel names.

  For DB2 columns, any table or view name that qualifies a column name. For DB2 correlation names, the DB2 qualification (*auth-id.table-name*).
- **(ENT) symbol entity** The entity type of a symbol. There are eight symbol entity types:
  - **COR** SQL correlation name
  - **DVW** Dataview
  - **PAR** Parameters
  - **PGM** Program
  - **PNL** Panel
  - **PRC** Procedure
  - **RPT** Report
  - **WOR** Working data

  The entity type PRC is always given to labels.

- **(T) Type** The data type of the symbol where applicable. Valid types are:
  - **C** Condition
  - **D** Date field
  - **F** Flag (T or F)
  - **G** Group
  - **L** Label or procedure name
  - **N** Signed numeric
  - **U** Unsigned numeric
  - **V** Variable length field
  - **X** Alphanumeric

  This column is blank for subprogram names, report names, pseudo functions, and fields in error.
- **(I) Internal Representation** Valid internal numeric representation types are as follows:
  - **Z** Zoned decimal
  - **P** Packed decimal
  - **B** Binary

  This column contains a value only when type is N, U, or D. For non-numeric data types, this column is blank.
- **(CH/DG) Characters or Digits** The number of characters or digits that the symbol contains including decimal places. Digits are denoted in format *dd.nn*, where *dd* shows the integer positions and *nn* shows the decimal positions.

  This column is blank for subprogram names, report names, pseudo functions, and fields in error.
- **(DEFN) Definition** The sequence number of the line in working data or parameter data where the symbol name is defined. This column is blank for program names, report names, literals, figurative constants, pseudo functions, and correlation names.

- **References** The statement numbers in the PDL procedure or report where the symbol name is referenced. A -U appears after the statement number if the symbol is updated in that statement. For example, the following statement number means that the symbol is a sending field on line 200 and a receiving field on line 300:

  ```
  200, 300-U
  ```

  You can reference identifiers, literals, pseudo-functions, and figurative constants in reports and in the procedure. You can reference condition names and labels only in procedures. Symbol names that are referenced in a report have an indicator RPT in front of all references in that entity.

  References in the procedure appear first, followed by references in a report. For example:

Symbol	References
FIELDX	200 300-U 400 RPT RPTX 200

  The value of FIELDX is referenced at statement numbers 200, 300, and 400 in the procedure. FIELDX is also referenced at sequence number 200 in report RPTX.

  The program lists:

  - For SQL tables and views, the statement number of each FOR construct and each SQL statement referenced.
  - For DB2 columns, each explicit reference to the column name with the appropriate update indicator, and separately after all other entity types correlation names.

Executing a Program

The RUN command initiates execution of a program by a user or by an application developer during the testing of an application. A program cannot run after an EDIT change until it is successfully compiled. For more information, see Compiling a Program.

You can run a program by issuing a RUN command in the command area during a session. See Running a CA Ideal Application Online. You can also submit a member containing a RUN command in a batch job stream. See the Batch CA Ideal and Running Batch Applications.

You can direct the output of a run-any generated reports-to the output library (for browsing online), to a system or network printer, or to a CA-eMail ID.

Determine the destination of a report in one of the following ways:

- Issue the RUN command without the destination clause. All output generated during the run are directed to the destination determined by the default destination clause values for the current session.
- Use the destination clause of the RUN command to specify where all reports generated by the application are directed. The destination clause information supplied in a RUN command overrides the default destination clause values for the current session.
- The ASSIGN REPORT statement can change the destination of a report during a run.

Using the RUN Command

To execute a program, enter the RUN command or select option 2 from the Program Maintenance Menu. If you enter the RUN command with no operands or select the RUN option from the menu, a prompter that provides the complete syntax displays. Fill in the required values and press the Enter key to enter the RUN command.

RUN * executes the current program.

Passing DATA to Programs through a RUN Statement

A parameter string in the RUN command can supply data for the first level-1 data item of the program's parameter section. For more information about RUN command, see the *Command Reference Guide*.

Passing DATA to an Application through Transparent Signon

In a CICS environment, you can call a non-Ideal subprogram to read the terminal input/output area (TIOA). You can pass the data entered on the screen after the CA Ideal Transparent Signon transaction ID to the CA Ideal calling program before any TRANSMITs execute. The CA Ideal *Administration Guide* describes how to define and use a transparent signon.

AItering the Runtime Environment

CA Ideal allows the resources an application uses to change based on the environment where the application is run. You can set these changes to apply to all users and applications running at a particular site, for a particular user for applications run during a session, or they can be determined dynamically by the application based on the environment where the application runs. You can make these changes without changing the application, recompiling the program, or changing the status of the application.

The modifications that you can make are:

- You can run CA Datacom/DB native dataviews against different databases than those assigned when the dataviews were cataloged.
- You can assign sequential dataviews in VSE a different device, block size, and file name.
- You can assign the reports an application generates to different printers than the printers established in the RUN command or as the default session or site printers. You can also give reports a different page size.
- You can substitute subprograms in test status for production status subprograms without recompiling the calling program.
- You can change the authorization ID for SQL objects a program accesses at runtime.

In most instances, you can accomplish these changes in three ways:

- The CA Ideal administrator can modify the object code for the application to reflect the change. This is done with ALTER commands, which are described in the *Command Reference Guide*.
- You can specify ASSIGN commands to modify the run environment for the current session. This section describes the ASSIGN commands. You can display all ASSIGN commands in effect by issuing the DISPLAY SESSION OPTIONS command.
- You can select some environment options dynamically using the PDL statements ASSIGN REPORT and ASSIGN DATAVIEW. The PDL statements are fully described in the *Programming Reference Guide*.

Assigning Dataviews to a Different Database

Initially, a dataview acquires the DBID of the corresponding DATABASE entity occurrence (identified in the dataview display as DBID *nnn*). A program using this dataview is then associated with the same database. Once a program is compiled with a dataview specified as a resource, this association with a DBID becomes part of the object code for the application.

An ASSIGN statement in PDL lets an application dynamically select a DBID for a specified dataview to run against. This statement is described in the *Programming Reference Guide*.

The ASSIGN command selects a DBID for a specified dataview for the current session. This lets the dataview run against a different DBID than the one associated with the dataview when the application was compiled.

Specify this command before the application runs. It is in effect for the duration of a session. This affects all applications using that dataview run during the session.

See the RESET command in the *Command Reference Guide* to revert a dataview back to the database it referenced when the application was compiled.

The ASSIGN DVW command takes precedence over the ASSIGN DBID command. The DBID is changed first, then the dataview.

For example, if your application programs were compiled using a test database (DBID 024) for all dataviews and you want to test those programs using a production database (DBID 025) for all dataviews except the PAYROLL dataview, you could enter the following commands:

```
ASSIGN DVW PAYROLL DBID 024
ASSIGN DBID 024 DBID 025
```

If several ASSIGN commands are made for the same dataview, only the last command in the sequence is in effect. For example:

```
ASSIGN DVW PAYROLL DBID 035
RUN PAY1
.
.
.
ASSIGN DVW PAYROLL DBID 036
RUN PAY1
.
.
.
```

Assigning Dataviews to a Different Table Partition

An ASSIGN statement in PDL lets an application dynamically select ANY or an individual child table of a CA-Datacom/DB partitioned table. For more information about the ASSIGN statement, see the *Programming Reference Guide*.

The ASSIGN command selects ANY or an individual child table of a CA-Datacom/DB partitioned table for a specified dataview for the current session.

Specify this command before the application runs. It is in effect for the duration of a session. This affects all applications using that dataview run during the session.

For more information about reverting a dataview back to the parent URT, see the RESET command in the *Command Reference Guide*.

Assigning a Global Substitute for a Database

Using the ASSIGN DBID command, you can substitute one database for another database during a session. This means you can direct any dataviews cataloged to run against a specific database or assigned to run against a specific database to a different database during the current session without making any changes to the dataviews.

This command remains in effect from the time you specify it until the end of a session or until a RESET command or another ASSIGN command is issued. It affects all references to the specified database ID. See the RESET command in the *Programming Reference Guide* to revert a database back to the database originally referenced by the applications.

The ASSIGN DVW command takes precedence over the ASSIGN DBID command. Regardless of the order in which the commands are entered, the DBID is changed first, then the dataview.

The ASSIGN command acts on the DBID identified in the program object code. Therefore, you cannot chain the assignment of databases with multiple ASSIGN commands. For example, the following commands are not equal to the command ASSIGN DBID 024 DBID 524, since only the DBID 024 is found in the program object code:

```
ASSIGN DBID 024 DBID 324
ASSIGN DBID 324 DBID 524
```

Substituting Subprograms for a Run

You can substitute a test version of a CA Ideal subprogram for the production or another test-status version of that program during a run using the ASSIGN PROGRAM command. This lets you test a subprogram with the production version application without replacing the production version subprogram or creating a test version of the entire application. This command only affects applications for the current CA Ideal session.

Changing the Authorization ID for SQL Access

For programs that access SQL objects, you can change the authorization ID at runtime. Using the ASSIGN AUTHORIZATION command, you can override the authorization ID specified in the plan or package resource table for a DB2 object or in the environment fill-in of a program that accesses an CA Datacom SQL object.

For CA Datacom SQL access, the ASSIGN AUTHORIZATION command is entered before running a program to select an alternate plan at runtime. As a result, the statements in the new plan execute instead of the statements in the default plan.

For DB2 access, the ASSIGN AUTHORIZATION command is entered before running a program in dynamic mode or before generating the plan for a program that runs in static mode. This command changes the authorization ID that qualifies the DB2 objects without changing the program.

For more information on the ASSIGN AUTHORIZATION command, see the *Command Reference Guide*.

Directing the Outputs of a Run

You can specify the destination of each report with the ASSIGN REPORT command. This command is issued before a run and is in effect for the duration of the current session. As many ASSIGN REPORT commands as there are reports generated can be active, to a maximum of 20 per user during one run.

The parameters specified in the ASSIGN REPORT command override the corresponding parameters specified in the RUN command. Parameters specified in an ASSIGN REPORT command override any corresponding parameters specified in the previous ASSIGN REPORT command.

The following table shows the default destinations (the batch file names) for some of the standard reports:

Default Destination	Report
AUXPRINT	Reports that are not otherwise assigned to external files.
COMPLIST	All batch compiler listings.
RUNLIST	All LIST statements produced by an application RUN (including LIST ERROR). You can only alter the RUNLIST destination.

For more information on the ASSIGN REPORT command, see the *Command Reference Guide*.

Resetting the Elements of a Run Environment

You can reset the assignment of a report, dataview, or program to the original or default assignment using the RESET command. However, if the original assignment was changed with the ALTER command, the RESET command cannot reset the assignment to the original or default assignment.

For more information on the RESET and RUN commands, see the *Command Reference Guide*.

Running a CA Ideal Application Online

You can run an application by issuing a RUN command in the command area during an online session or by submitting a member that contains a batch job stream. This job stream can contain a RUN command and can be submitted during an online session.

You must run applications that transmit panels online. You cannot run applications that contain sorted reports under CICS. All other applications can run online or in batch. Applications that produce reports can direct the reports to the output library, to a system printer, to a network printer (online only), or a CA-eMail ID.

Batch CA Ideal and Running a Batch Application

Batch CA Ideal can perform any CA Ideal service that is initiated by a command and that does not require interaction with the user. They include:

- Setting options. SET commands are described in the *Working in the Environment Guide*.
- Running utilities. See the *Working in the Environment Guide*.
- Managing entities, such as deleting, marking status, printing, and so on.
- Cataloging dataviews. See the *Creating Dataviews Guide*.
- Compiling programs. See Compiling a Program.
- Running batch programs. See the *Working in the Environment Guide*.
- Creating sorted reports. See the *Generating Reports Guide*.

Services that require interaction with a user cannot run in batch. For example, the following command has insufficient syntax to complete the DUPLICATE command and returns a prompter:

```
DUPLICATE
```

In batch, the DUPLICATE command terminates at this point since information cannot be entered into the prompter and the next CA Ideal command in the job stream executes. However, the following command works successfully in any environment, provided the entity exists:

```
DUPLICATE PANEL ORDFRM VERSION 2 NEXT VERSION
```

In addition, a program with any panel-processing PDL statements, such as TRANSMIT, cannot run in batch and terminates the run. If a DISPLAY command is used for index, session, or dataview option display, the command is treated like a PRINT command.

Note: Printing to a network printer (DESTINATION NET) is not allowed in batch.

Terminating a RUN

The RUN command, issued online or in batch, terminates upon successful completion of the executed program, upon encountering abnormal conditions, or by encountering online interruptions.

Successful completion of a run

A run terminates when program execution is successfully completed (when a QUIT RUN statement in any program or subprogram or a QUIT PROGRAM statement in the main program is encountered, or when the main program falls through to the end without an explicit QUIT RUN).

Abnormal termination of a run

A run terminates when CA Ideal does not give control to the ERROR PROCEDURE, such as when an environmental or system error occurs (for example, MAXLINES are exceeded). An error message is issued.

A run terminates when an execution error occurs and there is no ERROR PROCEDURE in the program. A default ERROR PROCEDURE that lists the error, performs a BACKOUT, issues a message, and quits the program is used. See the section Error Procedure in the *Programming Reference Guide*.

Online interruption of a run

A run terminates when a panel is on the screen and a command or function key initiates a new activity, such as Clear, Return, Edit, Display, or Delete.

At the end of a run, the message RUN completed, RC=*nn* appears. The RC=*nn* is the value of the return code at the end of the run. Each system message has a message level with an associated return code. The program can also explicitly set the return code to any value.

When an internal system error is detected in the run, $RC is set to 12. See also the $RC function in the *Programming Reference Guide*.

Message Level	Return-Code
A - Advisory	4
C - Conditional	16
D - Disaster	16
E - Error	8
F - Fatal error	12 or greater
I - Information	0
T - Terminal	16
W - Warning	4

How to Debug a Program

The DEBUG command executes a program under the control of the debugger. You can interrupt program execution, display data values, and modify data without altering the source program and without recompiling. Before you can execute a program with the DEBUG command, it must be successfully compiled. If you edited the program since the last successful compile, you must recompile the program before you can execute it with the DEBUG command.

To debug a program during an online session, you can issue a DEBUG command in the command area or select option 3 of the Process Program Menu. If you enter the DEBUG command with no operands or select option 3 from the Process Program Menu, a prompter displays for the missing information. To debug a program in batch, you can submit a member that contains a DEBUG command in a batch job stream.

You can direct reports generated during a debug run to the output library (for browsing online), to a system or network printer, or to a CA-eMail ID, just as they can during a normal run. The destination of a report is determined in the same way for a debug run as for a normal run. See the *Working in the Environment Guide* for details on redirecting output.

You can alter the destination of DBUGLIST, which is the file that contains the output from all LIST commands during a debug session, in the same way as RUNLIST.

To terminate a debug session, you can allow the run to end in the same way that it normally ends or you can enter a QUIT RUN debug command. The return code for the debug session is the same as it would be if you ran the program with a RUN command.

For more information on debugging programs, see the chapter "Symbolic Debugger." For complete details on the DEBUG command, see the *Programming Reference Guide*.

Chapter 8: Symbolic Debugger

This chapter describes the CA Ideal Symbolic Debugger. It begins with basic concepts and includes information on:

- Setting breakpoints
- Examining and changing data values
- Attaching commands to a breakpoint
- Controlling breakpoints
- Using command members, batch considerations
- Using the debug command with DB2, VSAM, or the CA-Datacom SQL

Debug Concepts

The CA Ideal debugger is a flexible tool for debugging programs. It lets you follow program logic and display the values of data items at crucial locations in the program. You can specify debugging commands, like CA Ideal commands, online or in batch.

The traditional method for debugging programs requires changing source code and recompiling the program. The CA Ideal debugger lets you perform debugging tasks separately from the source program. This means you can interrupt program execution, display data values, and modify data without altering the source program and without recompiling.

Breakpoints

A debug run is interrupted at breakpoints in the program where you can perform debugging tasks. There are four types of breakpoints:

BREAK

Assigned by the user at a statement in the program.

INIT

Assigned automatically at the start of the RUN.

QUIT

Assigned automatically at the end of the RUN.

ERROR

Assigned automatically when a fatal error occurs.

Breakpoints always occur at the beginning and end of a run and when an error that the error procedure cannot correct occurs. You can also set breakpoints at the beginning of as many program statements as required.

At a breakpoint, you can perform the following tasks:

- Display, print or modify data
- Display the source program
- Add, display, disable, or enable breakpoints
- Attach debug commands to execute at breakpoints
- Resume or quit the debug session

Commands

The following command starts a CA Ideal debug session:

DEBUG *program*

The DEBUG command lets you specify the same options that run the program with a RUN command.

Note: You must have DEBUG-TEST authorization to start the debugger. When you use DEBUG on a production program or subprogram, you must also have DEBUG-PROD authorization.

Online, you can issue DEBUG commands interactively or you can attach commands to breakpoints. When a particular breakpoint is reached, the DEBUG commands associated with the breakpoint execute. This is described fully in the section titled Attaching Commands to a Breakpoint later in this chapter.

In batch, all DEBUG commands are available that are typical for a batch environment.

You can issue the DEBUG commands in the following table at any breakpoint during a DEBUG run-unit.

Command	Meaning
AT	Specifies the location of a breakpoint.
COMMANDS	Switches to a fill-in screen containing the current debug commands or the debug commands contained in a specified member.
DATA	Displays data for the current debug breakpoint.
DELETE	Deletes breakpoints.

Command	Meaning
DISABLE	Temporarily disables breakpoints.
DISPLAY	Produces a formatted data display.
ECHO	Controls whether to write debug commands, output from DISPLAY and LIST commands, and breakpoint headers to the debug print file.
ENABLE	Enables breakpoints.
EQUATE	Defines abbreviations for the names of groups, fields, or parameters.
GO	Resumes processing of an application after a breakpoint.
LIST	Writes data displays to the debug print file.
MOVE	Modifies data.
PROC	Displays the procedure for a program.
QUIT	Turns off debugging.

Note: Use DEBUG commands only during a debugging session and only at breakpoints-not when application panels are transmitted.

In addition, you can issue the following CA Ideal commands at a breakpoint during a debug run.

- @I$TRACE
- END
- HELP/RETURN
- INCLUDE/EXCLUDE
- PRINT OUTPUT
- PRINT SCREEN
- SCROLL/POSITION
- FIND/RENUMBER/CHANGE/INPUT

- SET COMMAND
- SET EDIT
- SPLIT/OFF/COMBINE/REFORMAT

CA Ideal saves debug commands in a CA Ideal member. Online, this member is saved between debug sessions and can be used again with another session. The default name for this member is *uuu*.DEBUG, where *uuu* is your CA Ideal user ID. In batch, a unique name is generated for each run. For information on members for debugging batch programs, see the Using Command Members section in this chapter.

Debug Components

A debug session is a controlled run of a program; that is, the entire run is under the control of the debugger. At each breakpoint, you are shown one of the debug components:

- The commands fill-in
- The data display
- The procedure display

The following table shows what you can do in each component, what command accesses the component, and what kind of breakpoint automatically places you there.

Component	Function	Command	When Accessed Automatically
Commands	Display/edit breakpoints	CMD	At INIT breakpoint and attached commands
Data	Display data values	DATA	At breakpoint with attached commands
Procedure	Display procedure and set breakpoints	PROC	At breakpoint with no attached commands

You can perform the functions available in a component, issue commands to switch to another component, or resume or quit the debug session. These components are illustrated in the following sample session.

Sample Debug Session

The following session uses the default breakpoints to find what is wrong with program DBSAMP. It uses the GO command to proceed from breakpoint to breakpoint, the PROC command to display the program procedure, and the DISPLAY WORK command to display working data.

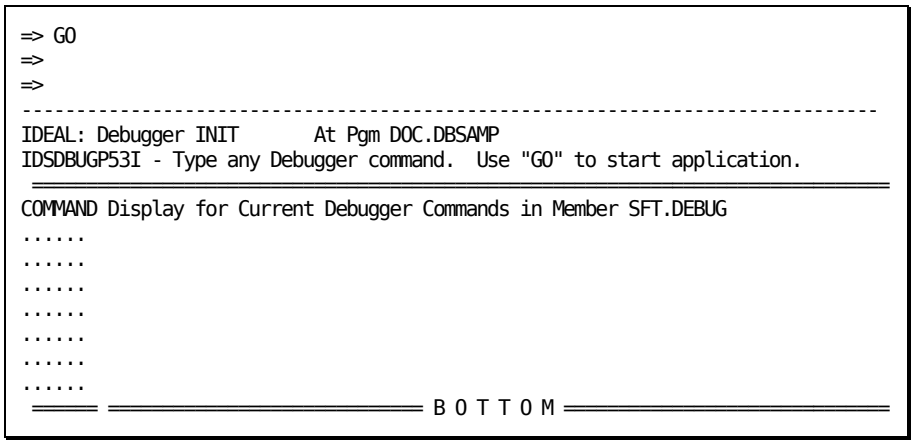
1. The DEBUG command starts a CA Ideal debugging session. So to debug program DBSAMP, enter:

   ```
   DEBUG DBSAMP
   ```

2. CA Ideal responds with the next display, the Initial breakpoint.

   The Initial breakpoint shows the contents of the DEBUG command member. Since this is the first debug run in the sample session, the command member is empty.

   The session automatically provides at least one command line.



   You can customize a DEBUG command member by using the CMD operand of the DEBUG command. In the member DBSAMP1, a previous DEBUG session's commands were saved at the end of the run. The following command lets you begin editing commands saved from that session at the INIT breakpoint:

   ```
   DEBUG DBSAMP CMD DBSAMP1
   ```

   When you complete your tasks at a breakpoint, type the command GO to proceed to the next breakpoint. You can include the GO command as one of the commands saved in the DEBUG command member for any breakpoint except the INIT breakpoint. There is no way to attach commands to the INIT breakpoint.

3. Because the program has an error, CA Ideal initiates the ERROR breakpoint. This screen indicates that the value of the subscript WT-LINE is incorrect. To determine which line of code caused the error, note the procedure name and statement number provided on the status line in the following figure.

```
=>  PROC
=>
=>

--------------------------------------------------------------------------------
IDEAL: Debugger ERROR      At Pgm DOC.DBSAMP   Proc MAIN           Stmt 000800
DESCRIPTION:  1-IDAETERR13E - Invalid subscript
================================================================================
IDSDBUGP59I - Additional Error Information follows:
 FATAL ERROR OCCURRED
CLASS=SUB   TYPE=SUB                                RETURN CODE=12
DESCRIPTION:  1-IDAETERR13E - Invalid subscript
NAME:          WT-WORK-TEXT-DATA.WT-LINE
VALUE:                                    TYPE=P, HEX=00000C
SUBSCRIPT           1
Level Field Name           Value                (Offset) Typ,Len (Occ)
 ===== ============================ B O T T O M ============================
```

You can display the source line that caused the error by issuing the PROC command. If the error occurred in a subprogram, you would have to enter the following command:

PROC *pgmname* VER *version* SYS *sys*

Note: You cannot edit the procedure during a debug run.

4. The result is the Procedure screen that follows. To display the value of WT-LINE and other working data at the time of the error, enter:

```
=> DISPLAY WOR
=>
=>
--------------------------------------------------------------------------------
IDEAL: Debugger ERROR      At Pgm DOC.DBSAMP   Proc MAIN           Stmt 000800
DESCRIPTION:  1-IDAETERR13E - Invalid subscript
================================================================================
PROCEDURE Display for Program DOC.DBSAMP   Version 001
<<MAIN>>   PROCEDURE

<<TEXT_REC>>
      FOR   FIRST TXT-REC
            LOOP
            VARYING WT-LINE
                  FROM WT-MAX   BY WT-STEP   DOWN THRU WT-MIN
>ERROR                  IF    TEXT-LINE(WT-LINE) NOT EQ $SPACES
                        SET   NUM-LINE-IN-REC =
                              $EDIT (WT-LINE,PIC='99')
                        PROCESS NEXT TEXT_REC
                  ENDIF
            ENDLOOP
      ENDFOR
ENDPROC
```

5. The display of working data in the following screen shows that the value of WT-LINE is zero, which is invalid for a subscript. End the error breakpoint with GO. This also terminates the DEBUG run-unit.

```
=>  GO
=>
=>
--------------------------------------------------------------------------------
IDEAL: Debugger ERROR      At Pgm DOC.DBSAMP   Proc MAIN           Stmt 000800
DESCRIPTION:  1-IDAETERR13E - Invalid subscript
================================================================================
=> DIS WOR
1     WT-WORK-TEXT-DATA
 2    WT-MAX              0                            NP,3
 2    WT-MIN              0                            NP,3
 2    WT-STEP             -1                           NP,1
 2    WT-LINE             0                            NP,5
====== ========================= B O T T O M ==============================
```

As shown in step 4, the value of WT-LINE is set by the LOOP varying statement using WT-MAX and WT-MIN as the upper and lower bounds of the loop. The value of both WT-MAX and WT-MIN is zero. Either could cause WT-LINE to contain an invalid value.

To investigate the cause of the incorrect WT-MAX and WT-MIN values further, you can display these values before the error is detected and at the first iteration of the loop. Do this by inserting a breakpoint at line 800, the location in the program where the error occurs.

Setting Breakpoints

From any CA Ideal program, you can set breakpoints for that program or for any other CA Ideal program for which you have debug authority. By default, breakpoints are set for the current program or procedure, but you can specify another procedure in the current program or another program. By setting breakpoints, you can interrupt the debugging run at points other than the default breakpoints. You can specify a breakpoint at the statement number of any executable statement or procedure header, and you can specify a breakpoint in an error procedure.

The breakpoint takes effect immediately before the statement executes (before the first verb in the statement executes if there is more than one verb).

Specifying Breakpoints

Use any of the following DEBUG commands to specify the locations of breakpoints:

- AT primary command (at any breakpoint)
- <label> line command (in PROC mode)
- AT line command (in PROC mode)

Use the AT primary command in the command area of any debug component or in the display area of the commands fill-in.

Use the AT and <label> line commands in the sequence number field of the procedure display. This AT line command specifies a break at line 300:

```
000200                  ...

AT0300                  IF TEXT-LINE(WT-LINE) NOT EQ $SPACES
```

This provides a breakpoint labeled <$*nn*>.

Subprograms

To set a breakpoint in a CA Ideal subprogram, use the procedures explained above, but specify the name of the subprogram. For example, to set a breakpoint at statement 100 of the subprogram B in system DYB, enter the following command while you are running the calling program in DEBUG mode:

```
AT STMT 100 IN DYB.B
```

Another way to do this is to display the Procedure Section by issuing the following command from the calling program:

```
PROC DYB.B
```

Then, when the Procedure Section of DYB.B displays, you can determine which statements require breakpoints.

Note: You *cannot* set a breakpoint for a non-Ideal subprogram.

Labels

Each user-defined breakpoint has a label associated with it. The label identifies the breakpoint and lets you specify commands to execute at the break. Each label in a run must be unique.

The format of a label is:

<xyz>

The value of *xyz* is any set of one to three letters, numbers, or national characters, except ALL.

For example:

```
<BK1>  <AA>  <$BK>  <$01>
```

You can specify labels when you set breakpoints, using meaningful tags and making certain that each label is unique. Or you can leave it to CA Ideal to assign a unique label to each breakpoint in the form <$01>, <$02>, <$03>, and so on.

Restrictions

Breakpoints entered at the following points are ignored:

- Non-executable statements, such as a comment line or ENDPROC.
- Non-existent statement numbers, such as 101, when statements are numbered 100, 200, and so on.

Breakpoints entered at the following statements cause the following actions:

Statement	Action
LOOP	Stop execution only once at the start of the first iteration.
ENDLOOP	Stop execution only once at the end of the last iteration.
WHILE or UNTIL	Stop execution at each iteration when the respective test is made.
FOR	Stop execution only once, at the start, before data is accessible.
ENDFOR	Stop execution only once, at the end, after all rows and records are processed. The last record is accessible, if any.
WHEN NONE	Stop execution if the clause is executed (that is, now rows are found).
IF	Stop execution before the IF condition is evaluated.

Statement	Action
ELSE	Stop execution at the beginning of the ELSE path. The IF condition is false.
ENDIF	Stop execution at the end of the construct. The IF condition can be true or false.

WHEN	Stop execution if the WHEN clause has an expression or a condition that performs a test, and the test is performed. That is, if execution of a SELECT construct reaches a WHEN clause, the breakpoint takes effect whether the WHEN tests true or false.
	For example, in the following SELECT FIRST construct, a breakpoint entered at statement 30 stops execution when AMT equals 500, 1,500, or 2,000. It does not stop execution if AMT=1,000 because after WHEN AMT=1000 control passes immediately to WHEN ANY and ENDSEL during SELECT EVERY logic.
	`10 SELECT FIRST ACTION`
	`20 WHEN AMT = 1000`
	`30 WHEN AMT = 2000 ...`
	If statement 10 is replaced by SELECT EVERY ACTION, a breakpoint set at statement 30 stops in EVERY case.

WHEN NONE, WHEN ALL, WHEN ANY	Stop execution only if the clause is, in fact, executed. In the following example, a breakpoint entered on the WHEN ANY stops execution only if statement 20 or 30 is executed.
	`10 SELECT FIRST ACTION`
	`20 WHEN AMT =1000`
	`30 WHEN AMT = 2000 ...`
	`40 WHEN ANY`

Stopping After a Statement

Breakpoints stop execution immediately before the selected statement. To stop execution immediately after a statement and before the next statement, you might have to introduce a dummy statement that has no effect on execution.

For example, a breakpoint entered at an ELSE stops execution at the beginning of the ELSE path. To stop execution immediately before the ELSE, you can enter a statement before the ELSE and assign it a breakpoint. To stop execution after the statement at line 20 in the following example, assign a breakpoint to the statement at line 30:

```
10  IF A = B
20      statement
30      MOVE W TO W
40  ELSE...
```

Sample Session

This sample runs the debugger with the same program from the previous sample, DBSAMP, setting a breakpoint.

1. Type the following command to debug the current program:

   ```
   ==> DEBUG *
   ```

2. CA Ideal displays the commands fill-in. You can also display the commands fill-in by entering COMMAND at any breakpoint.

   To create a breakpoint labeled <BR1> at line 800 in program DBSAMP, enter the following AT command in the commands fill-in:

   ```
   <BR1> AT 800
   ```

   The full range of editing commands, including scrolling commands, are available in the commands fill-in.

   You can see the breakpoint in a display of the program procedure by typing the command PROC. For a program other than the current program, type PROC *program-name*:

```
=> PROC
=>
=>

 ------------------------------------------------------------------------------
 IDEAL: Debugger INIT       At Pgm DOC.DBSAMP
 IDSDBUGP53I - Type any Debugger command.  Use "GO" to start application.
 ==============================================================================
       COMMAND Display for Current Debugger Commands in Member SFT.DEBUG
 ...... <BR1> AT 800
 ......
 ......
 ================================= B O T T O M ================================
```

3. In the program procedure in the following screen, note the breakpoint labeled <BR1>. If the AT command in the previous step did not specify a label, CA Ideal assigns the label <$01>.

   You can edit the Procedure display using all editing commands available when you edit a program definition display.

   To continue execution, issue the GO command.

```
=> GO
=>
=>
 ------------------------------------------------------------------------------
 IDEAL: Debugger INIT       At Pgm DOC.DBSAMP
 IDSDBUGP53I - Type any Debugger command.  Use "GO" to start application.
 ==============================================================================
       PROCEDURE Display for Program DOC.DBSAMP   Version 001
 000100 <<MAIN>>   PROCEDURE
 000200
 000300 <<TEXT_REC>>
 000400       FOR   FIRST TXT-REC
 000500             LOOP
 000600             VARYING WT-LINE
 000700                   FROM WT-MAX   BY WT-STEP   DOWN THRU WT-MIN
 <BR1>                     IF    TEXT-LINE(WT-LINE) NOT EQ $SPACES
 000900                          SET   NUM-LINE-IN-REC =
 001000                                $EDIT (WT-LINE,PIC='99')
 001100                          PROCESS NEXT TEXT_REC
 001200                    ENDIF
 001300             ENDLOOP
 001400       ENDFOR
 001500 ENDPROC
```

4. Execution stops at the breakpoint at line 800.

   To proceed, enter the GO command.

```
=> GO
=>
=>
 ------------------------------------------------------------------------------
 IDEAL: Debugger BREAK      At Pgm DOC.DBSAMP   Proc MAIN           Stmt 000800
 IDSDBUGP54I - Breakpoint cmd = <BR1> AT STMT 000800 IN DOC.DBSAMP
 ==============================================================================
        PROCEDURE Display for Program DOC.DBSAMP   Version 001
 000100 <<MAIN>>   PROCEDURE
 000200
 000300 <<TEXT_REC>>
 000400       FOR   FIRST TXT-REC
 000500             LOOP
 000600             VARYING WT-LINE
 000700                   FROM WT-MAX   BY WT-STEP   DOWN THRU WT-MIN
 <BR1>                     IF    TEXT-LINE(WT-LINE) NOT EQ $SPACES
 000900                          SET   NUM-LINE-IN-REC =
 001000                                $EDIT (WT-LINE,PIC='99')
 001100                          PROCESS NEXT TEXT_REC
 001200                    ENDIF
 001300             ENDLOOP
 001400       ENDFOR
 001500 ENDPROC
```

   Since the error was not corrected, the error breakpoint displays.

5. To end the session, enter QUIT DEBUG.

```
=> QUIT DEBUG
=>
=>

 ------------------------------------------------------------------------------
 IDEAL: Debugger ERROR      At Pgm DOC.DBSAMP   Proc MAIN           Stmt 000800
 DESCRIPTION:  1-IDAETERR13E - Invalid subscript
 ==============================================================================
        IDSDBUGP59I - Additional Error Information follows:
 000001 FATAL ERROR OCCURRED
 000002 CLASS=SUB   TYPE=SUB                              RETURN CODE=12
 000003 DESCRIPTION:  1-IDAETERR13E - Invalid subscript
 000004 NAME:          WT-WORK-TEXT-DATA.WT-LINE
 000005 VALUE:                                  TYPE=P, HEX=00000C
 000006 SUBSCRIPT           1
 000007 Level Field Name           Value                (Offset) Typ,Len (Occ)
 ================================= B O T T O M =================================
```

It is important to be aware of the fact that a breakpoint issues a checkpoint. Do not place your breakpoints anywhere you do not want a checkpoint issued. A few general rules are:

- If your program updates multiple DB records and you want all your changes backed out if an error occurs, do not place your breakpoint in the FOR construct.
- Do not place breakpoints in DB2 FOR constructs. The transmit of the debug screen causes a checkpoint and drops the DB2 cursor. A -501 results.
- Do not place a breakpoint between a Delete statement and an ENDFOR. An 'I3' error results.
- Do not code a breakpoint between two ESI calls.

Examining Data Values

At any breakpoint, you can examine the values of data items defined in working data, parameter data, panels, or dataviews that are defined in the current run-unit or that are resources of any program in the current run-unit.

To display values on the terminal, use DISPLAY. To list them in the debug print file, use LIST. In either case, the output is a formatted display that includes the debug DISPLAY and LIST commands used.

Online, you can issue the DISPLAY and LIST commands at any breakpoint or specify that they execute at any breakpoints.

Display

Online, a DISPLAY command produces a formatted display for the breakpoint. This screen shows when the DISPLAY is issued. You can access it by typing DATA. You can scroll the display, but not modify it. A sample is included in Changing Data Values later in this chapter. The DISPLAY command causes a syncpoint online.

In batch, DISPLAY is treated as a LIST.

List

The LIST command produces a formatted display for the breakpoint in the report DBUGLIST. When CA Ideal is delivered, the destination for DBUGLIST is the output library. The output number displays when debug terminates. You can display it by using the DIS OUT STATUS command. After debug terminates, use DISPLAY OUT or PRINT OUT to look at listing output.

You can also use the CA Ideal command ASSIGN REPORT DBUGLIST to assign the ddname or logical unit of a printer to the output to print the listing directly.

Note: DBUGLIST is subject to the restrictions imposed on print files, such as the limit of 16 simultaneous print files (15 unsorted reports and RUNLIST).

Echo

To list all debug commands issued and all breakpoints reached and to display output in the listing, issue the ECHO ON command at a breakpoint. From that point until an ECHO OFF executes, CA Ideal writes a complete record of the session to the DBUGLIST print file.

Changing Data Values

You can use the debug MOVE command to assign values to elementary data items during the debug session.

Sample Session

This sample runs the debugger with the current program, DBSAMP. It initializes WT-MAX and WT-MIN and, at the first break, displays the dataview TXT-REC and working data.

1. Type the following command to debug the current program:

   ```
   ==> DEBUG *
   ```

2. As the following commands fill-in shows, the debugger retains the breakpoint set in the previous session (see the section titled Setting Breakpoints earlier in this chapter), which is saved in the member DEBUG. The DEBUG INIT breakpoint sets initial values for WT-MAX and WT-MIN. The primary commands typed in this session are inserted at the bottom of the current DEBUG command member.

```
=> MOVE 2 TO WT-MAX
=> MOVE 1 TO WT-MIN
=> GO
--------------------------------------------------------------------------------
IDEAL: Debugger INIT       At Pgm DOC.DBSAMP
IDSDBUGP53I - Type any Debugger command.  Use "GO" to start application.
================================================================================
       COMMAND Display for Current Debugger Commands in Member SFT.DEBUG
000100 <BR1> AT STMT 000800 IN DOC.DBSAMP
================================== B O T T O M ================================
```

3. CA Ideal stops at breakpoint <BR1> and displays the program procedure. You can also display the procedure by entering PROC at any breakpoint.

   Enter the following debug command to display the values in the dataview TXT-REC:

   ```
   DISPLAY TXT-REC
   ```

```
=> DISPLAY TXT-REC
=>
=>

 ------------------------------------------------------------------------------
IDEAL: Debugger BREAK      At Pgm DOC.DBSAMP   Proc MAIN           Stmt 000800
IDSDBUGP54I - Breakpoint cmd = <BR1> AT STMT 000800 IN DOC.DBSAMP
================================================================================
       PROCEDURE Display for Program DOC.DBSAMP   Version 001
000100 <<MAIN>>   PROCEDURE
000200
000300 <<TEXT_REC>>
000400       FOR   FIRST TXT-REC
000500             LOOP
000600             VARYING WT-LINE
000700                   FROM WT-MAX   BY WT-STEP   DOWN THRU WT-MIN
<BR1>              IF    TEXT-LINE(WT-LINE) NOT EQ $SPACES
000900                   SET   NUM-LINE-IN-REC =
001000                         $EDIT (WT-LINE,PIC='99')
001100                   PROCESS NEXT TEXT_REC
001200             ENDIF
001300             ENDLOOP
001400       ENDFOR
001500 ENDPROC
```

   The command is logged in DBUGLIST and the dataview displays.

4. To verify that the values of WT-MAX and WT-MIN are correct, display the working data at the breakpoint by entering the debug command DISPLAY WOR.

```
=> DISPLAY WOR
=>
=>

 ------------------------------------------------------------------------------
IDEAL: Debugger BREAK      At Pgm DOC.DBSAMP   Proc MAIN           Stmt 000800
IDSDBUGP54I - Breakpoint cmd = <BR1> AT STMT 000800 IN DOC.DBSAMP
═══════════════════════════════════════════════════════════════════════════════
       Level Field Name          Value                (Offset) Typ,Len (Occ)
000001 => DISPLAY TXT-REC
000002 1     TXT-REC
000003  2    USE-CODE            W                             X,1
000004  2    WO-NUM
000005   3   PRODUCT-CODE        E                             X,1
000006   3   ORDER-CLASS         D                             X,1
000007   3   META-NUM
000008    4  META-YR            00                            X,2
000009    4  META-NO            001                           X,3
000010   3   ORD-SUB-CODE                                     X,1
000011  2    REC-SEQ-NUM                                      X,3
000012  2    NUM-LINE-IN-REC      3                           X,2
000013  2    TEXT-LINE                                        X,72  (25)
000014         (1)               This will allow the user to se
000015  2    EXTRA-SPACE                                      X,30
```

This reaffirms that the WT-MAX and WT-MIN have correct initial values. You could also display individual fields or parameter data with the DISPLAY command.

5. Complete the run with a QUIT RUN command.

```
=> QUIT RUN
=>
=>

 ------------------------------------------------------------------------------
IDEAL: Debugger BREAK      At Pgm DOC.DBSAMP   Proc MAIN           Stmt 000800
IDSDBUGP54I - Breakpoint cmd = <BR1> AT STMT 000800 IN DOC.DBSAMP
═══════════════════════════════════════════════════════════════════════════════
0000016 => DISPLAY WOR
000017 1     WT-WORK-TEXT-DATA
000018  2    WT-MAX              2                             NP,3
000019  2    WT-MIN              1                             NP,3
000020  2    WT-STEP             -1                            NP,1
000021  2    WT-LINE             2                             NP,5
═══════════════════════════ B O T T O M ═══════════════════════════════════════
```

Attaching Commands to a Breakpoint

You must retype the debug commands you enter interactively at a breakpoint each time that breakpoint occurs. To display the same data values, for example, you must retype the same DISPLAY commands. This section describes how to specify debug commands that execute automatically each time the program reaches a particular breakpoint.

Attaching to a User-Defined Breakpoint

To specify a command to execute at a user-defined breakpoint, first define the breakpoint:

```
==>  <BK1> AT 800
```

Then you can attach debug commands to the breakpoint. There are two ways to do this.

- From the command line on any debug component, enter:

  <*label*> *command*

  <*label*> is the label of the breakpoint. For example, the following specifies two commands for breakpoint <BR1>:

  ```
  ==> <BR1> MOVE TRUE TO FLAG
  ==> <BR1> DISPLAY WOR
  ```

- In the commands fill-in, you can type the commands on the lines following the breakpoint assignment.

  ```
  <BR1> AT100
  MOVE TRUE TO FLAG
  DISPLAY WOR
  ```

Attaching to ERROR or QUIT Breakpoint

To specify a command to execute at an ERROR or QUIT breakpoint, you first have to associate a label with the breakpoint. For example, the following specifies the label <QUI> for the QUIT breakpoint. You can you use any legal label-QUI is simply an example.

```
==>  <QUI> AT QUIT
```

Then you can associate debug commands with the breakpoint's label as described above. For example:

```
==> <QUI> DIS WOR
```

Commands You Can Use

You cannot attach all debug commands to breakpoints. For example, you cannot set a breakpoint from another breakpoint. The commands you can specify are:

- DELETE
- DISABLE
- DISPLAY
- ENABLE
- GO
- LIST
- MOVE
- QUIT

Deleting, enabling, and disabling are described later.

Sample Session

The following debug session attaches commands to the previously set breakpoint and to a new breakpoint at the top of the program.

1. Type the following to debug the current program:

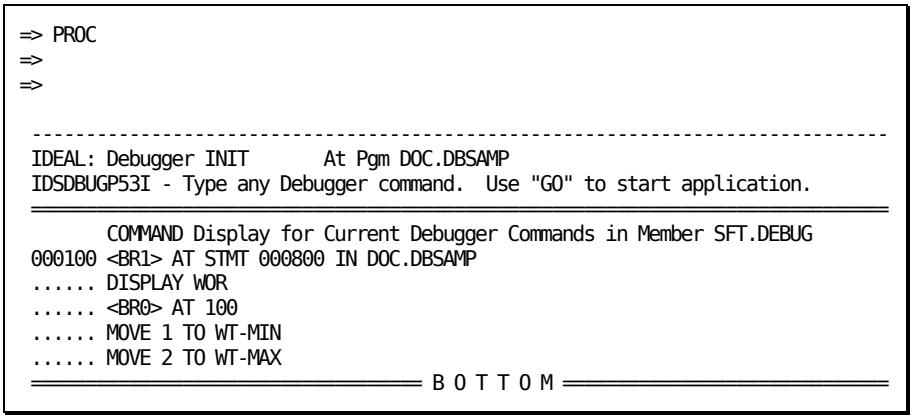
   ```
   ==> DEBUG *
   ```

2. The commands fill-in at the Init breakpoint shows the <BR1> breakpoint from the previous session.

   Edit the fill-in, inserting a DISPLAY WOR statement at <BR1> and new breakpoint <BR0> at line 100 with attached commands:

   ```
   MOVE 1 TO WT-MIN
   MOVE 2 TO WT-MAX
   ```

   Use the PROC command to see that these breakpoints were inserted where you intended as follows:



3. Proceed with the run by typing the command GO.

```
=> GO
=>
=>

 ------------------------------------------------------------------------------
IDEAL: Debugger INIT       At Pgm DOC.DBSAMP
IDSDBUGP53I - Type any Debugger command.  Use "GO" to start application.
═══════════════════════════════════════════════════════════════════════════════
       PROCEDURE Display for Program DOC.DBSAMP   Version 001
<BR0>  <<MAIN>>   PROCEDURE
000200
000300 <<TEXT_REC>>
000400       FOR   FIRST TXT-REC
000500             LOOP
000600             VARYING WT-LINE
000700                   FROM WT-MAX   BY WT-STEP   DOWN THRU WT-MIN
<BR1>                  IF    TEXT-LINE(WT-LINE) NOT EQ $SPACES
000900                       SET   NUM-LINE-IN-REC =
001000                             $EDIT (WT-LINE,PIC='99')
001100                       PROCESS NEXT TEXT_REC
001200                 ENDIF
001300             ENDLOOP
001400       ENDFOR
001500 ENDPROC
```

4. At the breakpoint <BR0>, the Data display shows the attached commands. Proceed with the run by typing the command GO.

```
=> GO
=>
=>
 ------------------------------------------------------------------------------
IDEAL: Debugger BREAK      At Pgm DOC.DBSAMP   Proc MAIN            Stmt 000100
IDSDBUGP54I - Breakpoint cmd = <BR0> AT STMT 000100 IN DOC.DBSAMP
═══════════════════════════════════════════════════════════════════════════════
       Level Field Name          Value                (Offset) Typ,Len (Occ)
000001 => MOVE 1 TO WT-MIN
000002 => MOVE 2 TO WT-MAX
══════ ═══════════════════════════ B O T T O M ═══════════════════════════════
```

5. At the breakpoint <BR1>, the Data display shows the working data fields. Proceed with the run by typing the command GO.

```
=> GO
=>
=>
 ------------------------------------------------------------------------------
IDEAL: Debugger BREAK      At Pgm DOC.DBSAMP   Proc MAIN           Stmt 000800
IDSDBUGP54I - Breakpoint cmd = <BR1> AT STMT 000800 IN DOC.DBSAMP
═══════════════════════════════════════════════════════════════════════════════
       Level Field Name          Value                (Offset) Typ,Len (Occ)
000001 => DISPLAY WOR
000002 1     WT-WORK-TEXT-DATA
000003  2    WT-MAX              2                           NP,3
000004  2    WT-MIN              1                           NP,3
000005  2    WT-STEP             -1                          NP,1
000006  2    WT-LINE             2                           NP,5
═══════════════════════════════ B O T T O M ═══════════════════════════════
```

6. Proceed with the run by typing the command GO. The run completes successfully without an ERROR stop. You should make appropriate changes to working data so the program proceeds properly under RUN instead of DEBUG.

Processing Without Terminal Interaction

To have debug commands execute at a breakpoint without stopping for terminal display, specify a GO command as the last command of the breakpoint.

The following sample session sets two breakpoints. At each breakpoint, any output data is listed to the print file without stopping for terminal interaction. The DEBUG session produces two outputs: The output of the DEBUG LIST commands, identified as DBUGLIST in the output library; and the output of the program's LIST statements, identified by the program name in the output library.

1. Begin the debug run. Type:

   ```
   ==> DEBUG DBGDEMO2
   ```

2. At the Init breakpoint in the commands fill-in, enter the AT, LIST, and GO commands for each breakpoint. Proceed by typing the GO command.

```
=> GO
=>
=>
 ------------------------------------------------------------------------------
IDEAL: Debugger INIT       At Pgm DOC.DBGDEMO2
IDSDBUGP53I - Type any Debugger command.  Use "GO" to start application.
═══════════════════════════════════════════════════════════════════════════════
       COMMAND Display for Current Debugger Commands in Member TOI.DEMOLIST
000100  <L2> AT STMT 000400 IN DOC.DBGDEMO2
000200         LIST WORK
000300         LIST LONG-NAME LONG
000400         GO
000500  <L1> AT STMT 000300 IN DOC.DBGDEMO2
000600         LIST SHORT-NAME GOT-THERE
000700         GO
═══════════════════════════════ B O T T O M ═══════════════════════════════
```

3. At the Quit breakpoint, the procedure displays. End the session with a GO command.

```
=> GO
=>
=>
 ------------------------------------------------------------------------------
 IDEAL: Debugger QUIT       At Pgm DOC.DBGDEMO2
 IDSDBUGP55I - At normal end of application. Use "GO" to complete application.
 ==============================================================================
       PROCEDURE Display for Program DOC.DBGDEMO2 Version 001
 000100 <<MAIN>> PROCEDURE
 000200   SET SHORT-NAME = 'SHORT'
  <L1>    SET LONG-NAME = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789'
  <L2>    SET SUM = 0
 000500   LOOP VARYING I FROM 1 THRU 3
 000600     ADD 1 TO SUM
 000700     IF  I EQ 2
 000800       SET GOT-THERE = TRUE
 000900     ENDIF
 001000   ENDLOOP
 001100   LIST SUM GOT-THERE SHORT-NAME LONG-NAME
 001200 ENDPROC
 ================================== B O T T O M ===============================
```

4. At the end of the run, CA Ideal displays a return code and the numbers of the outputs from the run. In this case, the outputs are 499 (the DBUGLIST) and 500 (the LIST statement output). To view a listing, type the DISPLAY OUTPUT command.

```
=> DISPLAY OUTPUT 499
=>
=>

 1-IDADRUNP04I - Run completed, RC = 0, OUTPUT = 499,500
 ------------------------------------------------------------------------------
 IDEAL: MAIN MENU            PGM DBGDEMO2 (001) TEST         SYS: DOC      MENU
  Enter desired option number ===>          There are  11 options in this menu:
  1. PROGRAM              Define and maintain programs
  2. DATAVIEW             Display dataview definitions
  3. PANEL                Panel Definition Facility
  4. REPORT               Report Definition Facility
  5. PLAN                 Application Plan Maintenance
  6. PROCESS              Compile, Run, Submit
  7. DISPLAY              Display Entities
  8. PRINT                Print Entities
  9. ADMINISTRATION       Administration functions
 10. HELP                 Overview of HELP facilities
 11. OFF                  End IDEAL Session
```

5. The following screen shows output 499, the DBUGLIST, which is the listing produced by the debug LIST command at each breakpoint. DBUGLIST contains a log of debugger activity since DEBUG was first issued. Each breakpoint is labeled and data LIST and DISPLAY images are recorded. DBUGLIST output is produced only in DEBUG.

```
DIS OUT 500
=>
=>
 ------------------------------------------------------------------------------ >>>
IDEAL  DISPLAY OUTPUT         OUT DBUGLIST (00499)                      DISPLAY
 ============================== T O P ===================================
IDEAL 11.0     DEBUGGER PRINTOUT                 March 1, 2006        11:32:11
 ------------------------------------------------------------------------------
IDEAL: Debugger BREAK      At Pgm DOC.DBGDEMO2 Proc MAIN           Stmt 000300
IDSDBUGP54I - Breakpoint cmd =  <L1> AT STMT 000300 IN DOC.DBGDEMO2
 ==============================================================================
Level Field Name           Value                 (Offset) Typ,Len (Occ)
=> LIST SHORT-NAME GOT-THERE
1     SHORT-NAME           SHORT                          X,5
1     GOT-THERE            F                              F
 ------------------------------------------------------------------------------
IDEAL: Debugger BREAK      At Pgm DOC.DBGDEMO2 Proc MAIN           Stmt 000400
IDSDBUGP54I - Breakpoint cmd =  <L2> AT STMT 000400 IN DOC.DBGDEMO2
 ==============================================================================
Level Field Name           Value                 (Offset) Typ,Len (Occ)
=> LIST WORK
1     SUM                  0                              NP,1
1     I                    0                              NP,1
1     GOT-THERE            F                              F
1     LONG-NAME            ABCDEFGHIJKLMNOPQRSTUVWXYZ0123 X,36
1     SHORT-NAME           SHORT                          X,5
=> LIST LONG-NAME LONG
1     LONG-NAME            ABCDEFGHIJKLMNOPQRST  (0000)   X,36
                           UVWXYZ0123456789      (0020)
 ============================== B O T T O M ===============================
```

6. LIST PDS statements continue to generate RUNLIST output.

```
=>
=>
=>
 ------------------------------------------------------------------------------ >>>
IDEAL  DISPLAY OUTPUT         OUT DBGDEMO2 (00500)                      DISPLAY
 ============================== T O P ===================================
 3 T SHORT ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
 ============================== B O T T O M ===============================
```

Controlling Breakpoints

Having defined breakpoints and attaching commands to them, you can control which breakpoints are active during a run. This section describes:

- Temporarily bypassing breakpoints
- Bypassing all breakpoints (QUIT DEBUG)
- Deleting breakpoints
- Bypassing the initial breakpoint

Temporarily Bypassing Breakpoints

There are two ways to temporarily deactivate breakpoints.

1. You can use the debug DISABLE command to deactivate one or all breakpoints. DISABLE does not remove a breakpoint from the final commands member. Use DELETE to remove breakpoints.

   Breakpoints that were disabled are left in the procedure display and commands fill-in. They are marked with a not-sign (Ø). On some terminals, this is an up-arrow ^. For example, in the commands fill-in:

   ```
   ==>DISABLE <BR1>
   ```

   The result is:

   ```
   <BR1>AT STMT 000800 IN DOC.DBSAMP
          DISPLAY WOR
   <BR0> AT STMT 000100 IN DOC.DBSAMP
          MOVE 1 TO WT-MIN
          MOVE 2 TO WT-MAX
   ```

2. You can produce the same result by entering a not-sign to the right of the breakpoint's label in the procedure display or commands fill-in.

   When you disable a breakpoint, any attached commands are also disabled.

You can use the ENABLE command to reactivate disabled breakpoints or delete the not-signs in the procedure display or commands fill-in.

The DEBUG commands DISABLE and ENABLE are useful both online and in batch. These commands let you increase or decrease the level of detail in terms of the number of breakpoints and the amount of information received.

Bypassing All Breakpoints (QUIT DEBUG)

At any breakpoint, you can terminate the debug session but continue program execution. The program no longer stops at breakpoints or executes attached commands. Enter:

```
==>QUIT DEBUG
```

This has the effect of disabling all breakpoints including ERROR and QUIT.

Deleting Breakpoints

There are several ways to permanently delete breakpoints.

- You can use the debug DELETE command. You can issue this command from the command area at any breakpoint. It is particularly useful for deleting all breakpoints in one command. For example:

  ```
  ==>DELETE ALL
  ```

- You can also attach DELETE to a breakpoint.
- You can delete breakpoints by deleting the appropriate labels in the procedure display or by deleting the appropriate lines in the Commands fill-in.
- You can specify an empty command member (see the section titled Using Command Members later in this chapter) by deleting all breakpoints previously entered.

Bypassing the Initial Breakpoint

At the Initial breakpoint, debug pauses to display the current command member. To bypass this breakpoint, issue the GO command with the DEBUG command. For example:

```
==>DEBUG DEMO1;GO
```

The semicolon (;) is the currently defined command delimiter.

Using Command Members

Debug commands entered in one session are saved in a CA Ideal member and can be used in subsequent sessions. The commands fill-in shows the contents of this member.

You do not have to specify a member name. Online, the default member name is DEBUG, belonging to the current user. In batch, a unique member name is generated for each session. In either case, unless you specify another member, debug uses this member, applying the commands in the member to the session and updating the member during the session.

In a debug session, the following are saved in the current command member:

- ECHO commands
- EQUATE commands
- AT commands
- Commands attached to breakpoints

Any other commands (such as DISPLAY, MOVE, and DELETE) are only saved if they are attached to a breakpoint. Deleted commands are removed at the end of the run.

In a command member used with multiple programs, the breakpoints and any attached commands are associated with their programs. EQUATE and ECHO commands affect all commands in the member.

The DEBUG member is like any other CA Ideal member-you can delete it, duplicate it, display it, and so on. CA Ideal creates it and updates it when you run debug, so you do not have to create or edit it, though you can.

Specifying a Command Member

You can specify separate command members for different programs by specifying the member name with the DEBUG command. For example:

```
==>DEBUG DEMO1 COMMANDS DEMO1
```

This uses member DEMO1 or creates a new member if DEMO1 does not exist. The debug commands from the current session are saved in DEMO1.

If you complete a debug run with the default member, you can still save the settings in another member for another debug run by duplicating the DEBUG member to another name. For example:

```
==>DUPLICATE MEMBER DEBUG NEWNAME DEMO1
```

You can then delete the default DEBUG member to start the next debug run fresh.

Later, whenever you want to debug a program, specify the command member associated with the program. For example:

```
DEBUG DEMO1 COMMANDS DEMO1
```

All of the breakpoints, attached commands, and so on are in place.

You can also switch command members during a debug run using the debug command COMMANDS with a member name. For example:

```
==> COMMANDS DEMO1
```

This command makes DEMO1 the current command member.

Editing a Command Member

You can edit a command member in a CA Ideal edit session or in a debug session using the commands fill-in following these rules:

- Only AT, EQUATE, ECHO, and attached commands are allowed.
- The commands must be in the sequence: ECHO, EQUATEs, ATs.
- Labels are optional on AT commands.
- Attached commands follow their AT commands. The attached commands do not have labels.
- Not-signs (Ø) denote disabled AT commands and their attached commands.
- A single continuation line is permitted on an attached command or an EQUATE. A trailing semicolon (;) indicates that the line is continued.

Note: To use another user's command member requires EDIT-MEMBER-ACROSS-USER authorization. Specifying another user's command member that does not exist requires CREATE-MEMBER-ACROSS-USER authorization.

Batch Considerations

In batch, you must enter any debug commands immediately following the DEBUG command. The last command must be GO.

When a breakpoint occurs, any attached commands execute and processing continues. Since there is no stop for terminal interaction, you do not need to attach a GO to each breakpoint.

Output similar to the online debug screens is written to the print file.

In batch, a DISPLAY command is treated as a LIST.

Batch Sample 1

This sample sets two breakpoints. The first breakpoint, labeled <1>, displays the value of a counter I. The second breakpoint, which has the default label <$01>, sets a breakpoint at line 800. The sample also displays working data when it reaches either the Error or the Quit breakpoint.

Batch Jobstream

```
DEBUG DBGDEMO2 VER 1
<1> AT 700
<1> D I
AT 800
<QUI> AT QUIT
<QUI> DISPLAY WOR
<ERR> AT ERROR
<ERR> DIS WOR
GO
```

RUNLIST Output

```
3 T SHORT ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
```

DEBUGLIST Output

```
IDEAL 11.0     DEBUGGER PRINTOUT                 March 1, 2006        12:53:26
--------------------------------------------------------------------------------
IDEAL: Debugger INIT       At Pgm WET.DBGDEMO2
IDSDBUGP53I - Type any Debugger command.  Use "GO" to start application.
 ==============================================================================
=> <1> AT 700
=> <1> D I
=> AT 800
=> <QUI> AT QUIT
=> <QUI> DISPLAY WOR
=> <ERR> AT ERROR
=> <ERR> DIS WOR
--------------------------------------------------------------------------------
IDEAL: Debugger BREAK      At Pgm WET.DBGDEMO2 Proc MAIN            Stmt 000700
IDSDBUGP54I - Breakpoint cmd =   <1> AT STMT 000700 IN WET.DBGDEMO2
================================================================================
Level Field Name           Value                 (Offset) Typ,Len (Occ)
 => D I
1     I                    1                              NP,1
--------------------------------------------------------------------------------
IDEAL: Debugger BREAK      At Pgm WET.DBGDEMO2 Proc MAIN            Stmt 000700
IDSDBUGP54I - Breakpoint cmd =   <1> AT STMT 000700 IN WET.DBGDEMO2
================================================================================
Level Field Name           Value                 (Offset) Typ,Len (Occ)
 => D I
1     I                    2                              NP,1
--------------------------------------------------------------------------------
IDEAL: Debugger BREAK      At Pgm WET.DBGDEMO2 Proc MAIN            Stmt 000800
IDSDBUGP54I - Breakpoint cmd = <$01> AT STMT 000800 IN WET.DBGDEMO2
================================================================================
--------------------------------------------------------------------------------
IDEAL: Debugger BREAK      At Pgm WET.DBGDEMO2 Proc MAIN            Stmt 000700
IDSDBUGP54I - Breakpoint cmd =   <1> AT STMT 000700 IN WET.DBGDEMO2
================================================================================
Level Field Name           Value                 (Offset) Typ,Len (Occ)
 => D I
1     I                    3                              NP,1
--------------------------------------------------------------------------------
IDEAL: Debugger QUIT       At Pgm WET.DBGDEMO2
IDSDBUGP55I - At normal end of application. Use "GO" to complete application.
 ==============================================================================
Level Field Name           Value                 (Offset) Typ,Len (Occ)
 => DISPLAY WOR
1     SUM                  3                              NP,1
1     I                    4                              NP,1
1     GOT-THERE            T                              F
1     LONG-NAME            ABCDEFGHIJKLMNOPQRSTUVWXYZ0123 X,36
1     SHORT-NAME           SHORT                          X,5
```

Batch Sample 2

The following sample sets two breakpoints. The first breakpoint, labeled <1>, displays the value of a counter I. It is immediately disabled and does not begin displaying the counter until control reaches the second breakpoint, which enables breakpoint <1>. The sample also displays working data when it reaches either the error or the QUIT breakpoint.

Batch Jobstream

```
DEBUG DBGDEMO2 VER 1
<1> AT 700
<1> D I
DISABLE 1
<ON> AT 800
<ON> ENABLE 1
<QUI> AT QUIT
<QUI> DISPLAY WOR
<ERR> AT ERROR
<ERR> DIS WOR
GO
```

RUNLIST Output

```
3 T SHORT ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
```

DBUGLIST Output

```
IDEAL 11.0     DEBUGGER PRINTOUT                  March 1, 2006       13:04:43
--------------------------------------------------------------------------------
IDEAL: Debugger INIT       At Pgm WET.DBGDEMO2
IDSDBUGP53I - Type any Debugger command.  Use "GO" to start application.
 ================================================================================
=> <1> AT 700
=> <1> D I
=> DISABLE 1
=> <ON> AT 800
=> <ON> ENABLE 1
=> <QUI> AT QUIT
=> <QUI> DISPLAY WOR
=> <ERR> AT ERROR
=> <ERR> DIS WOR
--------------------------------------------------------------------------------
IDEAL: Debugger BREAK      At Pgm WET.DBGDEMO2 Proc MAIN            Stmt 000800
IDSDBUGP54I - Breakpoint cmd =  <ON> AT STMT 000800 IN WET.DBGDEMO2
================================================================================
Level Field Name           Value                (Offset) Typ,Len (Occ)
 => ENABLE 1
--------------------------------------------------------------------------------
IDEAL: Debugger BREAK      At Pgm WET.DBGDEMO2 Proc MAIN            Stmt 000700
IDSDBUGP54I - Breakpoint cmd =   <1> AT STMT 000700 IN WET.DBGDEMO2
================================================================================
Level Field Name           Value                (Offset) Typ,Len (Occ)
 => D I
1     I                    3                             NP,1
--------------------------------------------------------------------------------
IDEAL: Debugger QUIT       At Pgm WET.DBGDEMO2
IDSDBUGP55I - At normal end of application. Use "GO" to complete application.
 ================================================================================
Level Field Name           Value                (Offset) Typ,Len (Occ)
 => DISPLAY WOR
1     SUM                  3                             NP,1
1     I                    4                             NP,1
1     GOT-THERE            T                             F
1     LONG-NAME            ABCDEFGHIJKLMNOPQRSTUVWXYZ0123 X,36
1     SHORT-NAME           SHORT                         X,5
```

Debug with DB2, VSAM, or SQL

You can adopt the following methods to debug with the dataviews.

Suppressing Terminal Interaction

Dataviews for SQL and VSAM SQL have restrictions on retaining sets or record positioning across transaction boundaries. In the case of the debug session, this is when a run stops to interact with the terminal.

For SQL dataviews, this is equivalent to releasing a set. Refer to the notes for the FOR statement for SQL dataviews in the *Programming Reference Guide*.

For VSAM dataviews, this applies to the use of non-unique alternate indexes. For information on the notes for the FOR statement for VSAM, see the *Programming Reference Guide*.

In any case, the problem does not arise if you suppress the interaction with the terminal by not attaching any DISPLAY commands and attaching a GO command to any breakpoints. Data can still be captured in the DBUGLIST print file using the LIST command.

Updateable Dataviews

A debug MOVE command in a FOR construct for an updateable dataview can update fields in that dataview. However, if a FOR construct for a DB2 dataview did not perform any updates when the program was compiled, a debug MOVE command causes a runtime error.

Program Function Key Assignments

The following program function key assignments are in effect while using the debug facility.

PF1/13 HELP

PF2/14 RETURN

PF3/15 PRINT SCREEN

PF4/16 PROCEDURE

PF5/17 DATA

PF6/18 COMMANDS

PF7/19 SCROLL BACKWARD

PF8/20 SCROLL FORWARD

PF9/21 GO

PF10/22 SCROLL TOP

PF11/23 SCROLL BOTTOM

PF12/24 INPUT

HELP

Displays a panel or series of panels that contain information to explain how to complete the current function.

RETURN

Returns from a help panel to the component display or from the function to the menu that selects the function.

PRINT SCREEN

Generates a hardcopy printout of the current screen contents.

PROCEDURE

Positions to the program procedure where the error occurred for an error breakpoint; otherwise, positions to the main program.

DATA

Positions to the Debug Data display.

COMMANDS

Positions to the debug commands fill-in.

SCROLL BACKWARD

Displays the previous frame in the current component.

SCROLL FORWARD

Displays the next frame in the current component.

GO

Proceeds to the next breakpoint.

SCROLL TOP

Positions to the first line of the component.

SCROLL BOTTOM

Positions to the bottom of the component.

INPUT

Opens a window of null lines preceding the first line of the component or at the current cursor position. Unused null lines in the window are deleted when you press the Enter key after INPUT.

Appendix A: Database Dependent Facilities

This appendix presents information about facilities that are database dependent. You can use several of these facilities in a way that is acceptable to both CA Datacom/DB native access and SQL. You can use others only for a specific database access.

Adaptable Facilities

To ensure maximum portability in adaptable facilities, you must satisfy the conventions for both environments. For example, a convention in CA Ideal for CA Datacom/DB native access requires that the left operand of the WHERE clause specify a column in the dataview and the right operand represent a valid value for the data type. CA Ideal SQL access lets you specify the values on either side of the relational operator, but does require that column names be qualified. If portability is important, always specify the fully qualified column name on the left, satisfying the requirements of both databases.

Take care when specifying arithmetic functions on the WHERE clause since CA Datacom/DB native access and SQL evaluate differently. CA Datacom/DB native access receives the value as CA Ideal evaluates it. SQL performs the evaluation. In some instances, the value CA Ideal evaluated can be truncated during processing to conform to the format CA Ideal specifies.

Dates are handled differently. CA Datacom/DB native access stores dates as a numeric value, while SQL dates are defined as alphanumeric.

IS NULL evaluates whether a nullable column contains a value. If the column is null, the row is selected, as in:

```
FOR EACH CUSTOMER
   WHERE SALESMAN IS NULL
  . . .
```

This locates all customers that currently are not assigned to a salesman because the column contains null. IS NOT NULL is true when the column does contain a value.

Specific Facilities

There are facilities specific to CA Datacom/DB native access and to SQL that you should use when a program is applied to a specific database only. The following pages list those facilities.

Naming Conventions

CA Datacom/DB native access

The naming conventions allow the use of the hyphen as in:

```
act-dt and open-$
```

SQL

Naming conventions let you use the underscore as in:

```
act_dt and open_$
```

Comparing Multiple Values

CA Datacom/DB native access

Operands can be implied as in the following to access every customer in Texas, Louisiana, or New Mexico:

```
FOR EACH CUSTOMER
WHERE STATE EQ 'TX' OR 'LA' OR 'NM'
. . .
```

SQL

You can use IN to designate more than one valid value for a specific column as in the following to access every customer in Texas, Louisiana, or New Mexico:

```
FOR EACH CUSTOMER
   WHERE STATE IN ('TX', 'LA', 'NM')
. . .
```

Comparing Masked Values

CA Datacom/DB native access

The CONTAINS/NOT CONTAINS relational operator is allowed for CA Datacom/DB native access requests as in the following to access the customers in all states whose two-character code begins with an A:

```
FOR EACH CUSTOMER
  WHERE STATE
     CONTAINS 'A*'
. . .
```

The asterisk represents any single character.

SQL

LIKE compares the alphanumeric value of a column with a string containing one or more mask characters as in the following to locate the customers in all the states whose two-character code begins with A:

```
FOR EACH CUSTOMER
   WHERE STATE LIKE ('A_')
. . .
```

You can specify multiple characters as in the following to locate all customers whose name begins with A, contains B as the third character, and Z as the last character:

```
FOR EACH CUSTOMER
   WHERE CUSTNAME
      LIKE   ('A_B%Z')
. . .
```

The underscore (_) represents any single character. The percent sign (%) represents zero or more characters.

WHERE Clause

CA Datacom/DB native access

CA Datacom/DB native access allows the right operand of the WHERE clause to be any valid expression. This includes using CA Ideal/PDL functions. For example, the following accesses the customers in the state indicated by the two characters returned through the $SUBSTR function:

```
FOR EACH CUSTOMER
   WHERE STATE EQ
     $SUBSTR(variable,
     START=1,LENGTH=2)
. . .
```

For more information on using functions, see the section on evaluating user input. For details on each function, see the *Programming Reference Guide*.

CA Datacom/DB native access lets you specify subscripted alphanumeric columns in the WHERE clause. This is done when an alphanumeric group is defined in the dataview. Since using group occurrences is not a recommended practice in a relational database and reflects poor design, an example is not provided here.

SQL

You can use BETWEEN to specify a valid range for a specific column as in the following that accesses all customers with an outstanding balance between $100 and $500, inclusive:

```
FOR EACH CUSTOMER
   WHERE OPEN$
      BETWEEN 100 AND 500
. . .
```

FOR Construct

CA Datacom/DB native access

You can use the FOR ANY form of the FOR construct. In addition, you can replace the FOR construct with SQL queries to the database. A single program can contain both FOR constructs and SQL queries. You can nest constructs and queries interchangeably, but SQL subqueries are not permitted on a FOR construct.

SQL

You can replace the FOR construct with SQL queries to the database. A single program can contain both FOR constructs and SQL queries. You can nest constructs and queries interchangeably, but SQL subqueries are not permitted on a FOR construct.