SAS/ACCESS® 9.1 Interface to IMS
Reference

Contents

vii

What's New

Overview

The SAS/ACCESS Interface to IMS now supports the ENCODING= data set option,
which enables you to override the encoding for processing a specific input or output file.
See "Understanding Character Set Encoding" on page 132.

Note:

□ This section describes the only feature of the SAS/ACCESS Interface to IMS that
changed since SAS 8.2.

□ z/OS is the successor to the OS/390 operating system. SAS/ACCESS 9.1 for IMS is
supported on both OS/390 and z/OS operating systems and, throughout this
document, any reference to z/OS also applies to OS/390, unless otherwise stated.
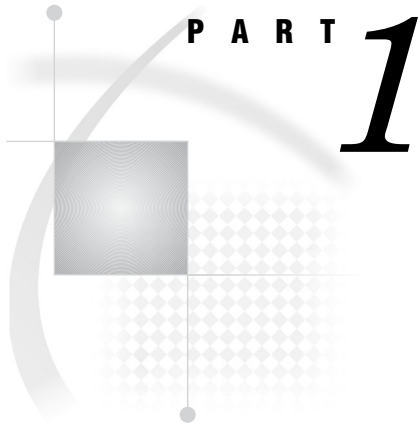
△

P A R T *1*

SAS/ACCESS Interface to IMS: Introduction

Overview of the SAS/ACCESS Interface to IMS

Introduction to the SAS/ACCESS Interface to IMS

This section introduces you to SAS/ACCESS software and briefly describes how to use the interface. This section also introduces the sample IMS data and SAS data files used in this document.

Purpose of the SAS/ACCESS Interface to IMS

SAS/ACCESS software provides an interface between SAS and the IMS database management system (DBMS). You can perform the following tasks with this SAS/ACCESS interface:

- □ create SAS/ACCESS descriptor files using the ACCESS procedure
- □ directly access data in IMS databases from a SAS program using the view descriptor files created with the ACCESS procedure
- □ extract data from IMS databases and place it in a SAS data file using the ACCESS procedure, the DATA step, or other SAS procedures
- □ update, insert, or delete data in IMS databases using the SQL procedure, SAS/FSP software, the APPEND procedure, or the MODIFY statement. The MODIFY

statement can be used with the IMS interface view engine, and supports REPLACE, DELETE, and INSERT calls.

□ issue DL/I calls to update, insert, and delete data in IMS databases using the DATA step interface's INFILE, INPUT, FILE, and PUT statements.

Using the SAS/ACCESS Interface to IMS

The SAS/ACCESS interface to IMS consists of three parts:

□ the ACCESS procedure, which you use to define the SAS/ACCESS descriptor files

□ the IMS interface view engine, which enables you to use IMS descriptor files in SAS programs in much the same way you use SAS data files

□ the DATA step interface, which enables you to access information in IMS databases using SAS programming statements.

The ACCESS procedure enables you to describe an IMS database to SAS in an access descriptor file. You can then create view descriptor files from the access descriptor file, which you can use in SAS programs in much the same way as you would use SAS data files. You can print, plot, and chart the data described by the view descriptor files, use it to create other SAS data sets, and so on. Chapter 3, "Defining SAS/ACCESS Descriptor Files," on page 41 describes how to create and edit SAS/ACCESS descriptor files. Chapter 4, " IMS Data in SAS Programs," on page 49 presents examples of using IMS data in SAS programs, and Chapter 5, "Browsing and Updating IMS Data," on page 71 shows how to use the view descriptor files to update IMS data from within a SAS program.

The interface view engine is an integral part of the SAS/ACCESS interface, but the interface's design is transparent, so you are seldom aware of the engine. SAS interacts automatically with the engine when you use SAS/ACCESS view descriptors in your SAS programs, so you can use IMS data in your programs in much the same way as you use SAS data.

The DATA step interface provides special extensions of standard SAS INFILE and FILE statements to access IMS resources. Chapter 9, "How to Use the IMS DATA Step Interface," on page 197 describes these statement extensions in detail.

You might need to combine data from several sources, including IMS databases, Version 6 SAS data sets, Version 7 SAS data sets, and other databases. With the SAS/ACCESS interface, such combinations are not only possible, but easy to do. SAS can differentiate among SAS data sets, SAS/ACCESS view descriptor files, and other types of SAS files, and it can use the appropriate access method.

How the IMS Engine and DATA Step Interfaces Differ

When comparing the two interfaces, you can identify some obvious differences:

□ The IMS interface view engine requires you to create descriptor files. The engine uses information from the descriptor files to successfully attach IMS and retrieve or update the data being requested by the application. The DATA step interface requires no such files since it is a programming interface.

□ The engine access method provides transparent access to IMS data. To access data, you simply make reference to a descriptor file and you have access to the data defined by the view. Coding DATA step programs requires in-depth knowledge of the database that is being accessed, and the ability to code host level calls to retrieve or update IMS data. You can, however, create SAS DATA step

views from the DATA step programs to provide users who are unfamiliar with the DL/I language transparent access to the data.

When to Use the IMS Engine Interface

Use the IMS engine interface for the following situations:

□ transparent access to IMS data.

□ access to data that lies in a single database path. Performance is enhanced when segment search arguments (SSAs) can be generated from WHERE statements.

□ assignment of READ, WRITE and ALTER levels of protection with passwords.

When to Use the IMS DATA Step Interface

Use the IMS DATA step interface for the following situations:

□ programs that need full control over DBMS access. The DATA step interface provides total CHKP control in an update program as well as control over DBMS calls and SSAs.

□ transaction-style programs that need the capability of dynamically generating SSAs from SAS variable values in transaction files.

□ multi-path processing or accessing data from multiple databases in the same application. Joining data can be more efficient when performed in the DATA step as opposed to the engine interface.

□ access to message queues in a BMP region.

□ access to Fast Path databases.

Features Not Supported by the IMS Engine Interface

The engine does not support Fast Path, message queue access, or HSSR. The SLI region type is also no longer supported; however, you can use the DBCTL feature of IMS/ESA and CICS/ESA for those functions.

Features Not Supported by the IMS DATA Step Interface

The DATA step does not support the DLITEST procedure and HSSR. The SLI region type is also no longer supported; however, you can use the DBCTL feature of IMS/ESA and CICS/ESA for those functions.

SAS/ACCESS Descriptor Files for IMS

SAS/ACCESS descriptor files are the tools that the SAS/ACCESS interface view engine uses to establish a connection between SAS and IMS. To create these files, you run the ACCESS procedure using one of three methods:

□ batch mode

□ interactive line mode

□ noninteractive mode.

There are two types of descriptor files: access descriptors and view descriptors. They are discussed in the next two sections. The following figure illustrates the relationships among an IMS database, an access descriptor, and view descriptors. Chapter 3,

"Defining SAS/ACCESS Descriptor Files," on page 41 shows you how to create, browse, and edit these files.

Figure 1.1 Relationships between an IMS Database, an Access Descriptor, and View Descriptors
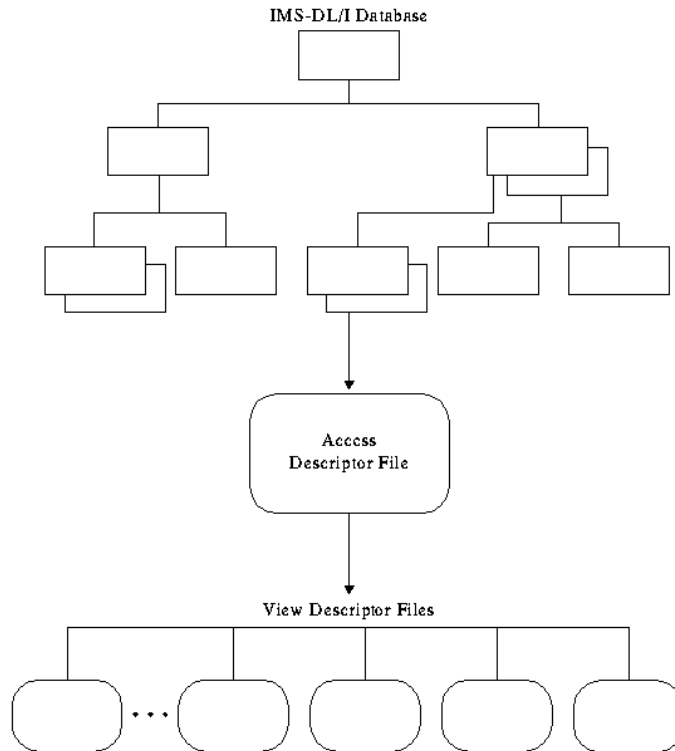
Access Descriptor Files

Access descriptor files are of member type ACCESS. Each access descriptor holds essential information about the IMS database that you want to access, for example, the name of the database; the names of segments, search fields, and key fields; segment lengths; names of non-search fields in the database; and data types. An access descriptor might also contain the corresponding SAS information, such as the SAS variable names and formats. Typically, you have only one access descriptor for each IMS database.

Since an access descriptor describes only one IMS database, you cannot create a single access descriptor that accesses multiple *physical* IMS databases; however, you can create a single access descriptor for a *logical* IMS database that accesses multiple physical databases.

View Descriptor Files

View descriptor files are sometimes called *views* because their member type is VIEW. This document uses the term *view descriptors* to distinguish view descriptor files from views created by the SAS SQL procedure or a DATA step.

Each view descriptor can define all the data in one path of the database, or a particular subset of the data in one path of the database (see Chapter 2, "IMS Essentials," on page 11 for a discussion of paths and other IMS concepts). View

descriptors enable you to define all the data in one path of the database by selecting the items that you want to use and specifying selection criteria for only the records that you want. For example, you might want only records with a transaction date of July 3, 1995, and for customers who live in Richmond. You might have several view descriptors, each selecting different paths of data in an access descriptor that you have defined. You might also have view descriptors that select different subsets of data in one path of an access descriptor.

You can join data by using SAS's SQL procedure. With the SQL procedure, you can create a view that joins and summarizes data from multiple view descriptors (based on IMS databases), SAS data files, DATA step views, or other PROC SQL views. See Chapter 4, " IMS Data in SAS Programs," on page 49 for a discussion and examples that use the SQL procedure.

Executing SAS/ACCESS Programs in Batch Mode

The JCL (job control language) that is required to execute programs using the SAS/ACCESS interface to IMS in OS/390 batch mode is similar to that of other SAS jobs in OS/390 batch mode. Refer to the *SAS Companion for z/OS* for general information about SAS jobs in OS/390 batch environments.

Executing a Cataloged Procedure

The JCL for a batch job that accesses IMS data requires that you specify your site's designated cataloged procedure in the EXEC statement. So, instead of specifying your site's default SAS cataloged procedure (such as **// EXEC SAS**), you use the following EXEC statement:

```
// EXEC your-cataloged-procedure
```

The name of the cataloged procedure that invokes SAS and supports the use of the SAS/ACCESS interface to IMS differs at each installation, particularly if your installation uses separate cataloged procedures for accessing test databases and production DL/I databases. Be sure to check with your site's SAS support consultant for the correct procedure name. (SAS no longer supplies the Version 5 SASDLI cataloged procedure.) The installation notes that are shipped with the SAS/ACCESS interface to IMS explain to your database administrator how to create a cataloged procedure for your site.

The cataloged procedure for accessing IMS data contains all of the JCL statements and parameters that the SAS cataloged procedure contains, plus JCL statements and parameters necessary for the run-time execution of the IMS engine interface and IMS DATA step programs. PROC ACCESS can use the standard SAS catalog if the only task you are performing is creating descriptor files.

DD Statements

If you execute DL/I calls through a batch DL/I region (DLI or DBB), DD statements for all the database and index data sets that are accessed must be included in the job step JCL. DDnames and DS names (names of database data sets) must be obtained from the DBA staff at your site. Data sets that support the index must be allocated for the HIDAM database type.

If you execute DL/I calls through an online DL/I access region (BMP), database data sets are allocated to the DL/I control region or to the CICS control region. Therefore, you do not need to include DDnames for them in the job step JCL.

When you execute a batch DL/I region and want to log updates, you need a DD statement for a log data set. For information about using the IMSLOG option, on pointing to the IMS resident libraries, and information about pointing to the DBD, PSB, and ACBLIB, check with your DBA or refer to the installation instructions for the SAS/ACCESS interface to IMS.

All other JCL considerations that are outlined in the *SAS Companion for z/OS* apply to the IMS engine interface and IMS DATA step execution.

Executing SAS/ACCESS Programs under TSO

The SAS/ACCESS Interface to IMS can run interactively if your site has installed SAS under TSO. The TSO commands needed for the run-time execution of the IMS engine interface and IMS DATA step programs are similar to those for other TSO SAS jobs. (For general information about using SAS under TSO, see the *SAS Companion for z/OS* and the installation instructions for the SAS/ACCESS interface to IMS). PROC ACCESS can use the standard SAS CLIST if the only task that you are performing is creating descriptor files.

Allocating Database Data Sets

If you access DL/I databases through a batch DL/I region (DLI or DBB), you must first allocate the database data sets. You can allocate these database data sets from within or outside of SAS.

From within a SAS session, you use the SAS FILENAME statement to associate DDnames with database data sets and other OS/390 files that might be accessed by the interface view engine or a DATA step program. You can specify the FILENAME statements in the SAS Program Editor or in an AUTOEXEC file.

From outside of a SAS session, you can use a TSO ALLOCATE command. You can obtain the appropriate filerefs and data set names from the database administrator (DBA) staff at your site.

If you execute DL/I calls through an online DL/I access region (BMP), the database data sets are allocated to the respective DL/I control regions. Therefore, you do not need to allocate them with the TSO ALLOCATE command (or other means).

All other commands and TSO environment considerations are described in the *SAS Companion for z/OS*.

About the Example Data in the Document

This document uses two HDAM IMS databases, the ACCTDBD database and the WIRETRN database. These databases were created for a bank, and they contain data about the bank's customers and their checking and savings account transactions. The seven ACCTDBD database segments are named CUSTOMER, CHCKACCT, CHCKDEBT, CHCKCRDT, SAVEACCT, SAVEDEBT, and SAVECRDT. The WIRETRN database has one segment, WIRETRAN, and includes only data pertaining to wire transfers of money. All the data in the document is fictitious.

The document also uses one HIDAM database, EMPLINF2, in the examples.

Note: These databases are designed to show how the interface treats IMS-DL/I data. They are not meant as an example for you to follow in designing databases for any purpose. △

Appendix 2 gives more information about the ACCTDBD database and the data it contains. Appendix 2 also includes definitions of all the view descriptors referenced in this document and all the SAS data files and statements used to create them. Chapter 3, "Defining SAS/ACCESS Descriptor Files," on page 41 provides information about the WIRETRN database and definitions of the view descriptors used in the examples.

Running the Examples in This Document

To run the examples based on the ACCTDBD and WIRETRN databases, you must first load the database files and define the access and view descriptors shown in Appendix 2. Use the sample library files described here.

IMSLD
  contains the source programs for loading the ACCTDBD, EMPLINF2, and WIRETRN database files for both the engine interface and DATA step. It includes the JCL used to allocate the IMS databases, to create DBDs and PSBs, and to create needed flat files.

IMSEX
  contains the example SAS programs that use the engine interface, as shown in Chapter 4, " IMS Data in SAS Programs," on page 49 and Chapter 5, "Browsing and Updating IMS Data," on page 71.

IMSDS
  contains the example SAS programs that use the DATA step interface, as shown in Chapter 1, "Overview of the SAS/ACCESS Interface to IMS," on page 3 and Chapter 9, "How to Use the IMS DATA Step Interface," on page 197.
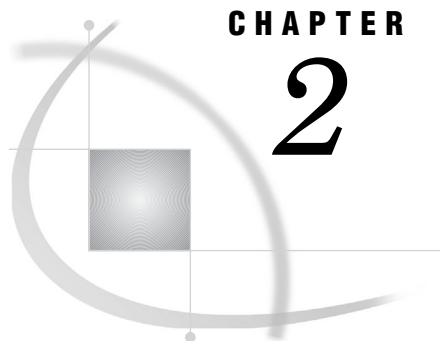
C H A P T E R

2

IMS Essentials

Introduction to IMS Essentials

　　This section introduces SAS users to IMS, a hierarchical database management system by IBM. It focuses on the terms and concepts that will help you use the SAS/ACCESS interface to IMS. It includes descriptions of the following:

□ hierarchical database structure and elements

□ IMS databases

□ physical databases

□ the elements of DL/I calls

□ execution modes

□ resource sharing.

The IMS DBMS

IMS (Information Management System) is a program licensed by IBM. It is a database management and data communication system that is used to manage intricate databases and terminal networks. IMS enables you to define, reorganize, and load data structures, and to relate data structures to an application.

With IMS, you can use the high-level language DL/I (Data Language/I) to operate on the data that is controlled by the DBMS. DL/I calls are invoked from application programs written in languages such as PL/I, COBOL, and C, or by subroutine calls from assembler language programs.

Overview of IMS Databases

An *IMS database* is a large, centralized collection of information comprising one or more physical files that can be accessed by the SAS/ACCESS interface to IMS. An IMS database is a *hierarchical database*. Information is structured in *records* that are subdivided into a hierarchy of related *segments*.

A record is a root segment and all of its dependent segments. Segments are further subdivided into *fields*. The data in any record relates to one entity. Ideally, information in the database records is subdivided into segments and fields on some logical basis, either by the inherent structure of the data or by consideration of the uses to which the data will be put.

The term *hierarchical* implies that there are levels of data. You can think of a hierarchical database as one that starts at the top with general information about the item, individual, or case. As you progress from level to level down through the hierarchy, more and more information related to the general information at the top level is given. Each level in the hierarchy has one or more segments.

In some ways the structures of IMS databases and tabular files (such as SAS data sets) are comparable, but in other ways they differ. For example, database fields and data set variables are similar, and database records are like data set observations because both contain data about one entity. At the same time, however, database records differ from data set observations because subsets of records can be accessed while you cannot access a subset of a data set observation. The observation is stored and accessed as a unit.

A tabular file has nothing comparable to a segment. The concept of data segments is one of the things that makes a hierarchical database different from a SAS data set and other tabular files.

Consider banking data as an example. Customer information maintained by a bank might include the following:

name	checking account debits
Social Security number	checking account credits
address	savings account number
home phone	savings account balance
work phone	savings account date of last statement
checking account number	
checking account balance	savings account balance at last statement
checking account date of last statement	savings account debits
checking account balance at last statement	savings account credits

If this information is stored in a tabular file, each item of information is a *variable,* and all of the variables for any given person comprise one observation. You can visualize the layout of banking data in a tabular file as shown in the following figure.

Figure 2.1 Tabular File Structure

The rows in the table represent observations (customers), and the columns represent variables. The structure of the file is such that a maximum number of variables for debits and credits must be defined when the file is created. In the previous figure there are variables for up to only three debits and three credits per customer, which presents a problem if a customer has more than three debit or credit transactions.

The same data can also be stored in an IMS database but would be structured very differently. For example, the following figure shows one way the banking information could be structured in IMS. The sample database, called ACCTDBD, is used in this document and described in "About the Example Data in the Document" on page 8 in Chapter 1, "Introducing the SAS/ACCESS Interface to IMS."

Figure 2.2   Hierarchical File Structure
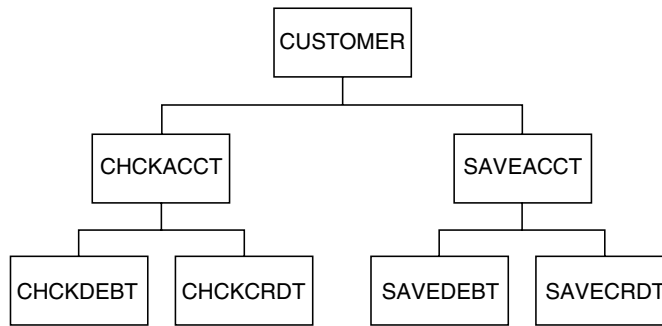
Each block in the figure represents a *segment type*, which is a grouping of related *fields* of data. There are three levels in the ACCTDBD database hierarchy and seven segment types. For each database record, the top or first level has only one segment, called the *root segment*. The root segment in the ACCTDBD database is called CUSTOMER; it contains fields with these data: Social Security number, customer name, address, city, state, country, ZIP code, home phone, and work phone. The segments under the root segment are *dependent segments* called CHCKACCT, CHCKDEBT, CHCKCRDT, SAVEACCT, SAVEDEBT, and SAVECRDT. Each of the dependent segments contains fields of data, as shown in the following table.

Table 2.1   Dependent Segments and Corresponding Fields

Dependent Segment	Fields
CHCKACCT	checking account number, current balance, last statement date, last statement balance
CHCKDEBT	checking account debit date and time, amount, description
CHCKCRDT	checking account credit date and time, amount, description
SAVEACCT	savings account number, current balance, last statement date, last statement balance
SAVEDEBT	savings account debit date and time, amount, description
SAVECRDT	savings account credit date and time, amount, description

Segment Occurrences

The hierarchical database structure is useful for storing multiple occurrences of any given element of information, especially if there are varying numbers of occurrences of the data for each record.

Consider the dependent segment SAVEACCT, which contains the following fields:

☐ savings account number

☐ savings account balance

  □ date of last statement

  □ savings account balance at last statement.

Different customers can have different numbers of savings accounts; some might have none, others might have two or three. If the data is not segmented, there must be space in each customer's record for the maximum number of savings accounts per customer. With the segmented structure, however, it is possible to have one SAVEACCT segment occurrence for each savings account a customer has. Any segment type can have an unlimited number of segment occurrences. Although the segment types are predefined, the number of segment occurrences is not predefined. Note that each occurrence of a root segment represents a separate record.

Here is an example of how a segment type can have an unlimited number of segment occurrences. A certain customer has two savings accounts. In one month, the customer has two deposits for account number 111 and one deposit and two withdrawals for account number 222. The following figure shows the customer's record.

Figure 2.3 Sample Record
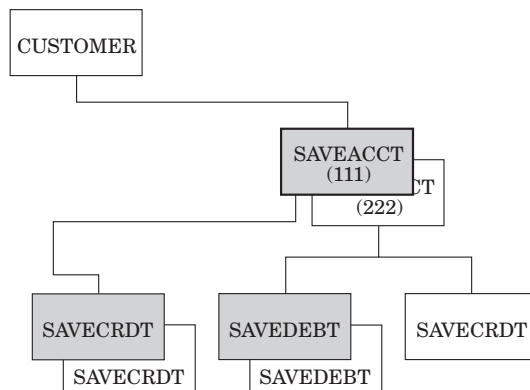
The figure shows seven segment occurrences within three (shaded) segment types.

Segment Relationships

The information in a hierarchical database is subdivided or segmented according to a logical scheme. Moving from top to bottom through the database, there is a relationship between the segments. A segment that is hierarchically dependent on a segment one level up in the hierarchy is said to be the *child*. The segment on which it is dependent is the *parent*. CUSTOMER is a parent segment with two children, CHCKACCT and SAVEACCT. CHCKACCT, in turn, is the parent of CHCKDEBT and CHCKCRDT, and SAVEACCT is the parent of SAVEDEBT and SAVECRDT. Segments that share a parent are called *siblings*; for example, CHCKACCT and SAVEACCT are siblings. Multiple segment occurrences of one segment type with the same parent occurrence are called *twins*. For example, SAVEACCT 111 and SAVEACCT 222 are twins.

All dependent segments are children but are not necessarily parents. The root segment (CUSTOMER), on the other hand, is a parent if any dependent segments exist, but it is never a child. (It is possible to have a database with no dependent segments, that is, with only one level, the root segment.) In a hierarchical structure, there can be only one parent segment for a child segment.

Segments can also be grouped by *paths*. Two segments belong to the same path if one is a dependent of the other. You can access multiple segments in a path at the same time. These relationships are shown in the following figure.

Figure 2.4   Segment Relationships
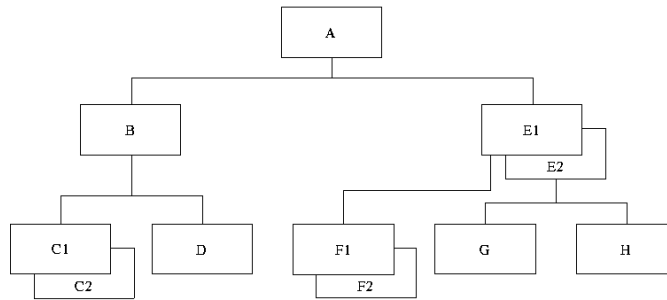
Parents:
    A, B, E1, E2

Children:
    B, E1, E2, C1, C2, D, F1, F2, G, H

Twins:
    C1 and C2, E1 and E2, F1 and F2

Siblings:
    B, E1, and E2; C1, C2, and D; G and H

Paths:
    A, B, and C1; A, B, and C2; A, B, and D; A, E1, and F1; A, E1, and F2; A, E2, and
    G; A, E2, and H

Path Navigation

You can navigate one path of an IMS database at a time with the interface view engine in Version 7 and later of SAS. That is, you can select items in one path of the database when creating a view descriptor. Consider Figure 2.5 on page 17, which shows one path of data shaded. The SAS/ACCESS interface processes each record occurrence from top to bottom and from left to right following these rules:

1 The first occurrence of a root segment is processed first.
2 Then, the first child of a root segment in the defined path is processed before twins of the root segment.
3 Twins are processed after a child (if any) down that path. Twins are processed in order of occurrence. Any child of a twin is processed according to this rule.
4 After the child and twins are processed for that one path, the next eligible root segment in the path is processed.

Note: No siblings are processed. △

The following figure illustrates a path of data in a particular program view. The numbering indicates the order of processing.

Figure 2.5 A Database Path
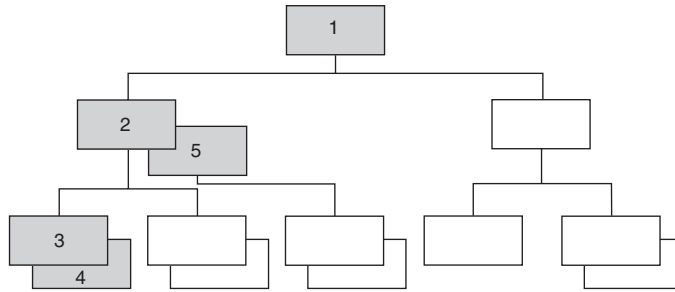
Fields

There are three types of fields in the segments of an IMS database:

☐ A *sequence field* (or *key field)* is a field that identifies and provides access to segments in a database. A sequence field is defined to IMS in the *database description* (*DBD*), which specifies characteristics of a database. In some cases, a sequence field sequences twin segment occurrences in ascending order, according to their sequence field values. For example, if the sequence field of the CHCKACCT segment is ACNUMBER, twin CHCKACCT occurrences in a given customer's record are ordered from the lowest to highest account number. Root segments usually have a sequence field, but dependent segments do not necessarily have them.

In a root segment, the sequence field also uniquely identifies the record. In dependent segments, the sequence field can provide unique identification, but this is not required. Root segments might or might not be sequenced by the sequence field, depending on the IMS access method used to store the database.

☐ A *search field* is defined to IMS in the DBD and is used to search through the database for particular values. For example, CUSTZIP is defined as a search field in the CUSTOMER segment, permitting the SAS/ACCESS interface to IMS to search the database for records containing a specific ZIP code.

☐ An *undefined field* is not defined to IMS. All fields other than sequence fields and search fields do not have to be defined in the DBD. IMS does not know the format of an undefined field and cannot search for segments based on values in an undefined field. The format of an undefined field is determined by the program that loads the database initially.

The CUSTOMER segment of the ACCTDBD database contains examples of two kinds of fields. There are fields for Social Security number, name, city, state, country, ZIP code, address, home phone, and work phone. The Social Security number field is defined as the sequence field, meaning that it uniquely identifies the record. The name field of CUSTOMER does not uniquely identify a record because customer names might be duplicated. However, because names can be used to search through the database, the name field is defined as a search field, as are the address, city, state, country, ZIP code, home phone, and work phone fields.

The sequence field of a root segment allows direct access to the root segment. The sequence field of a dependent segment does not allow direct access to the record, but IMS finds segments faster when searching on sequence fields rather than search fields.

Physical Databases and Program Views

A *physical database* is defined to IMS in one DBD, which is described later in this section. A physical database is limited to 15 hierarchical levels and 255 segment types (up to 254 dependent segments organized over 14 levels, plus the root segment). There is no limit to the number of segment occurrences, however. Chapter 9, "How to Use the IMS DATA Step Interface," on page 197, illustrates the physical database ACCTDBD.

A *program view* of a database consists of the hierarchically structured segments used in a program or application. A particular program view can be composed of all segments in a database or a subset of the segments, depending on the program's requirements. Program views are defined to IMS in *program communication blocks* (*PCBs),* which are contained in *program specification blocks* (*PSBs).* (See the next several sections for more information about DBDs, PCBs, and PSBs.)

The following figure illustrates a program view that consists of some segments from the ACCTDBD database. This program view might be used by a program that prints monthly checking account statements. However, a SAS/ACCESS view descriptor can access data in only one path in the database. Therefore, in one invocation, the view descriptor can retrieve data in either the CHCKDEBT segment or the CHCKCRDT segment.*

Figure 2.6   Sample Program View
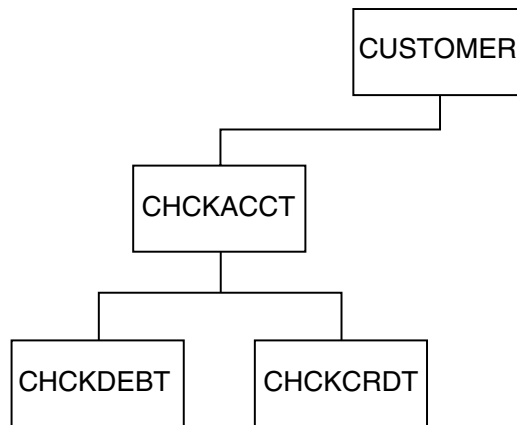
In order for the SAS/ACCESS interface to access an IMS database, certain information about the database must be defined. These definitions are contained in DBDs and PSBs.

What You Need to Know to Create Descriptors

Typically, DBDs and PSBs are generated by the database administration staff, not by application programmers or users. Users do not need to know how to create DBDs, PSBs, or PCBs in order to use the SAS/ACCESS interface to IMS. If you will be creating access descriptors and view descriptors, you need to know the following information:

□ the name of the database you want to use (DBD name).

---

* With the SAS/ACCESS DATA step interface, the view shown can be processed in a single DATA step execution. See Chapter 8, "Overview of the IMS DATA Step Interface," on page 153 for more information.

☐ the DDnames and names of database data sets (and for HIDAM, the index data sets).

☐ the name of the PSB that contains the PCBs to be used for your database.

☐ which fields are sequence or search fields (as defined in the DBD).

☐ standard segment descriptions, such as field name, field length, and data type. You might need a copybook, segment layout, or some other detailed description of the database if this information is not in the DBD.

☐ what type of access is permitted (as defined in the PCBs in the PSB).

☐ the sensitive segments (as defined in the PCBs in the PSB) and their names and lengths. A *sensitive segment* is a segment in the database that can be accessed only by using a PCB that permits read or update access.

☐ the order of the PCBs in the PSB, and which PCBs your program needs to access.

☐ the order in which fields are defined in the segment.

You can get all of this information from your installation's database administrator (DBA).

The descriptions of DBDs and PSBs that follow do not need to be understood in detail. They are included here for readers who want this additional information.

Database Description

The database description (DBD) is usually created by the DBA at an installation. The DBD specifies characteristics of a database, including the following:

❶ the name of the DBD, which is also used as a shorthand name for the IMS database (1-8 characters).

❷ the type and access method for the database (DEDB, MSDB, HDAM, HIDAM, HSAM, HISAM, GSAM, SHISAM, or SHSAM). These database types are defined in the next section.

❸ the randomizing method to assign an address to each record's key (HDAM only).

❹ the DDname for the database.

❺ the device type.

❻ the block size.

❼ the name, parent, and length of each segment type in the database. The parent information enables IMS to determine the segment's position in the hierarchy.

❽ the name, length, starting position, and type of data for each sequence and search field in each segment. (In the following example, the code that specifies these characteristics is highlighted.)

Note: It is not necessary to specify every field in a segment in the DBD. Only those fields to be used as sequence and search fields are specified in the DBD. △

DBD for the ACCTDBD Database

The following is the DBD for the ACCTDBD database.

```
DBD ❶ NAME=ACCTDBD, ❷ ACCESS=(HDAM,OSAM),        X
     ❸ RMNAME=(DFSHDC40,3,71)
DATASET ❹ DD1=ACCTDD, ❺ DEVICE=3380,             X
```

```
❻ BLOCK=2400
❼ SEGM   NAME=CUSTOMER,PARENT=0,BYTES=225
❽ FIELD NAME=(SSNUMBER,SEQ,U),BYTES=11,START=1,   X
  TYPE=C
  FIELD NAME=CUSTNAME,BYTES=40,START=12,TYPE=C
  FIELD NAME=CUSTADD1,BYTES=30,START=52,TYPE=C
  FIELD NAME=CUSTADD2,BYTES=30,START=82,TYPE=C
  FIELD NAME=CUSTCITY,BYTES=28,START=112,TYPE=C
  FIELD NAME=CUSTSTAT,BYTES=2,START=140,TYPE=C
  FIELD NAME=CUSTLAND,BYTES=20,START=142,TYPE=C
  FIELD NAME=CUSTZIP,BYTES=10,START=162,TYPE=C
  FIELD NAME=CUSTHPHN,BYTES=12,START=172,TYPE=C
  FIELD NAME=CUSTOPHN,BYTES=12,START=184,TYPE=C
 ❼ SEGM   NAME=CHCKACCT,BYTES=40,PARENT=CUSTOMER
 ❽ FIELD NAME=(ACNUMBER,SEQ,U),BYTES=12,START=1, X
  TYPE=X
  FIELD NAME=STMTAMT,BYTES=5,START=13,TYPE=P
  FIELD NAME=STMTDATE,BYTES=6,START=18,TYPE=X
  FIELD NAME=STMTBAL,BYTES=5,START=26,TYPE=P
 ❼ SEGM   NAME=CHCKDEBT,BYTES=80,              X
     PARENT=((CHCKACCT,DBLE)),RULES=(,LAST)
 ❽ FIELD NAME=DEBTAMT,BYTES=5,START=1,TYPE=P
  FIELD NAME=DEBTDATE,BYTES=6,START=6,TYPE=X
  FIELD NAME=DEBTBLNK,BYTES=2,START=12,TYPE=X
  FIELD NAME=DEBTTIME,BYTES=8,START=14,TYPE=C
  FIELD NAME=DEBTDESC,BYTES=59,START=22,TYPE=C
 ❼ SEGM   NAME=CHCKCRDT,BYTES=80,              X
     PARENT=((CHCKACCT,DBLE)),RULES=(,LAST)
 ❽ FIELD NAME=CRDTAMT,BYTES=5,START=1,TYPE=P
  FIELD NAME=CRDTDATE,BYTES=6,START=6,TYPE=X
  FIELD NAME=CRDTBLNK,BYTES=2,START=12,TYPE=X
  FIELD NAME=CRDTTIME,BYTES=8,START=14,TYPE=C
  FIELD NAME=CRDTDESC,BYTES=59,START=22,TYPE=C
 ❼ SEGM   NAME=SAVEACCT,BYTES=40,PARENT=CUSTOMER
 ❽ FIELD NAME=(ACNUMBER,SEQ,U),BYTES=12,START=1, X
  TYPE=X
  FIELD NAME=STMTAMT,BYTES=5,START=13,TYPE=P
  FIELD NAME=STMTDATE,BYTES=6,START=18,TYPE=X
  FIELD NAME=STMTBAL,BYTES=5,START=26,TYPE=P
 ❼ SEGM   NAME=SAVEDEBT,BYTES=80,              X
     PARENT=((SAVEACCT,DBLE)),RULES=(,LAST)
 ❽ FIELD NAME=DEBTAMT,BYTES=5,START=1,TYPE=P
  FIELD NAME=DEBTDATE,BYTES=6,START=6,TYPE=X
  FIELD NAME=DEBTBLNK,BYTES=2,START=12,TYPE=X
  FIELD NAME=DEBTTIME,BYTES=8,START=14,TYPE=C
  FIELD NAME=DEBTDESC,BYTES=59,START=22,TYPE=C
 ❼ SEGM   NAME=SAVECRDT,BYTES=80,              X
     PARENT=((SAVEACCT,DBLE)),RULES=(,LAST)
 ❽ FIELD NAME=CRDTAMT,BYTES=5,START=1,TYPE=P
  FIELD NAME=CRDTDATE,BYTES=6,START=6,TYPE=X
  FIELD NAME=CRDTBLNK,BYTES=2,START=12,TYPE=X
  FIELD NAME=CRDTTIME,BYTES=8,START=14,TYPE=C
  FIELD NAME=CRDTDESC,BYTES=59,START=22,TYPE=C
  DBDGEN
```

DBD for the WIRETRAN Segment

The following is the DBD for the WIRETRAN segment of the WIRETRN database.

```
DBD    NAME=WIRETRN,ACCESS=(HDAM,OSAM),          X
        RMNAME=(DFSHDC40,3,71)
  DATASET DD1=WIREDD,DEVICE=3380, BLOCK=2400
  SEGM  NAME=WIRETRAN,PARENT=0,BYTES=100
  FIELD NAME=(SSNACCT,SEQ,M),BYTES=23,START=1,  X
   TYPE=C
  FIELD NAME=ACCTTYPE,BYTES=1,START=24,TYPE=C
  FIELD NAME=WIREDATE,BYTES=8,START=25,TYPE=C
  FIELD NAME=WIRETIME,BYTES=8,START=33,TYPE=C
  FIELD NAME=WIREAMMT,BYTES=5,START=41,TYPE=X
  FIELD NAME=WIREDESC,BYTES=40,START=46,TYPE=C
  DBDGEN
```

IMS Database Types

During installation, the database administrator (DBA) chooses the type of database to use for the IMS databases. The DBA decides which type of database to use based on how most of the programs that use an IMS database will access the data in the database. The following is a list of database types that the DBA can use to define an IMS database that is supported by the SAS/ACCESS interface to IMS in Version 7 and later of SAS:

Data Entry Database (DEDB)
: is a direct-access database that consists of one or more areas, with each area containing both root segments and dependent segments. The database is accessed using VSAM improved control interval processing (ICIP). This database type can only be used with the SAS/ACCESS DATA step interface.

Main Storage Database (MSDB)
: is a root-segment database, residing in main storage, which can be accessed to a field level. This database type can only be used with the SAS/ACCESS DATA step interface.

Hierarchical Direct Access Method (HDAM)
: is one of the DL/I language's two direct-access methods. A direct-access method enables DL/I to locate any database record, regardless of the record sequence in the database, by using a randomizing routine or an index. HDAM provides direct access to data through a randomizing routine. Sequentially accessing an HDAM database, DL/I retrieves data in the order that the data is physically stored in the database.

Hierarchical Indexed Direct Access Method (HIDAM)
: is one of DL/I language's two direct-access methods. HIDAM provides direct access to data through an index.

Hierarchical Sequential Access Method (HSAM)
: is one of DL/I language's sequential-access methods. In a sequential-access database, segments are stored in a hierarchical sequence, one segment after another. HSAM provides sequential access to root segments and dependent segments. You can access data in HSAM databases, but you cannot update any of the data.

Hierarchical Indexed Sequential Access Method (HISAM)
: processes data sequentially, but has an index that enables you to directly access records in the database.

Generalized Sequential Access Method (GSAM)
> enables IMS/ESA batch application programs to access a sequential OS/390 data set record that is defined as a database record. This database record is handled as one unit, with no segments, fields, or hierarchical structure. Any records to be added are inserted at the end of the database. GSAM does not enable you to update or delete records in the database.

Simple Hierarchical Sequential Access Method (SHSAM)
> is an HSAM database that contains only one segment type, a root segment. Only two types of calls are valid with SHSAM databases: *Get calls* to read a database and *Insert calls* to load a database. You must reload a database in order to update it.

Simple Hierarchical Indexed Sequential Access Method (SHISAM)
> is a HISAM database with only one segment type, a root segment.

IMS Data Types

When specifying the characteristics of the physical database in the DBD, the DBA identifies for each segment in the database the fields that IMS can use to search or sequence a segment. The DBA can define each individual field, define the entire segment as one field and assign a generic data type, or define some fields individually and other fields as a group. The DBA can define fields in an IMS database segment using the following data types:

Table 2.2   Data Type Codes and Corresponding Data Types

Data Type Code	Data Type
X	hexadecimal
P	packed decimal
C	alphanumeric character
F	binary fullword
H	binary halfword
Z	zoned decimal
E	short floating point
D	long floating point
L	extended floating point

Note: All COBOL and PL/I data types are supported as hexadecimal data types. △

IMS Data Types in SAS/ACCESS Descriptors

To create access and view descriptors to be used by the SAS/ACCESS interface to IMS, you need to know how the DBA has defined the database fields. You also need to know how the fields are initialized and the order of all the fields in each segment to be accessed. You can get this information from a layout of the database or a COBOL copybook.

The following table shows the DBFORMAT= value that you specify in an access descriptor for some common COBOL and PL/I data types. This table also shows the SAS variable formats that the SAS/ACCESS interface generates from the IMS DBFORMAT= value.

Table 2.3   Recommended DBFORMAT= Values to Use for Common COBOL and PL/I Data Types

IMS Type	COBOL	PL/I	Description	Standard Length in Bytes	Recommended DBFORMAT=	SAS Format Generated
C	PIC A	Pic 'A'	Alphabetic	<=200	$*w*.	$*w*.
				>200	$200.	$200.
C	PIC X	Char or Pic 'X'	Alphanumeric	<=200	$*w*.	$*w*.
				>200	$200.	$200.
Z	PIC 9	Pic '9'	Numeric Edited			*w.d*
Z	PIC S9	Pic '99T'	Zoned-Decimal		ZD*w.d*	*w.d*
H	PIC 9(4) COMP	Fixed Bin (15)	Fixed-Point Binary	2	IB2.	7.0
F	PIC 9(8) COMP	Fixed Bin (31)	Fixed-Point Binary	4	IB4.	10.0
E	COMP-1	Float Bin (21)	Floating-Point	4	Rb4.	E13.0
D	COMP-2	Float Bin(53)	Floating-Point	8	RB8.	E22.0
P	COMP-3	Fixed Decimal	Packed-Decimal	<=16	PD*w.d*	*w.d*

When you create an access descriptor, you use the ITEM= statement to describe the IMS DBD. When you need to specify a SAS informat that corresponds to a COBOL data description, refer to PICTURE and USAGE. If the USAGE is COMP-1 or COMP-2, there is no PICTURE. If no USAGE is specified, it defaults to DISPLAY.

Use the following information to make the conversions:

☐ pictures that include either **A** or **X** represent character values.

☐ pictures that include numbers use **9** to represent digits. They might use an **S** to mean signed and a **V** to show the location of an implied decimal point.

The number of characters or digits is specified either by the number of **A**s, **X**s, or **9**s in the picture or by the number in parentheses immediately after the **A**, **X**, or **9**. For example, **AAAA** is the same as **A(4)**.

The following table shows other conversions.

Table 2.4   COBOL Conversions

USAGE	PICTURE	SAS Informat	Width	Decimal
COMP-1	None	RB4.		
COMP-2	None	RB8.		
DISPLAY	9(int)V9(fract)	ZD*w.d*	(int + fract)	(fract)
COMP-3	9(int)V9(fract)	PD*w.d*	CEIL((int+fract+1)/2)	(fract)
COMP	9(int)V9(fract)	IB*w.d*	*	

*   If the (*int* + *fract*) is 1-4, the width is 2 and decimal is a fraction. If the (*int* + *fract*) is 5-9, the width is 4 and decimal is a fraction. If the (*int* + *fract*) is 10-18, the width is 8 and decimal is a fraction.

Use SAS formats to print the fractional part read with the IB*w.d* and RB*w.d* SAS informats.

Program Specification Block

A Program Specification Block (PSB) is generally created by the DBA at an installation. A PSB consists of one or more program views of one or more databases. A SAS task using the SAS/ACCESS interface to access an IMS database must reference one and only one PSB. Information specified in the PSB includes the following items:

- □ at least one program view for each database that is accessed by the SAS/ACCESS interface in the executing task. A program view is defined in a PCB; in fact, you can use the terms program view and PCB interchangeably. Each PCB provides these specifications:

  - □ the database to be accessed.

  - □ the processing options (read-only or various updating options).

  - □ the maximum length of the concatenated key fields (sequence fields) in any path.

  - □ the database segments that can be accessed. These segments are called *sensitive segments*. The name, parent segment, and access mode for each sensitive segment is given.

- □ the programming language the PSB supports. The SAS/ACCESS interface to IMS uses a PSB regardless of the specified programming language. This flexibility means that no additional PSBs are required for the SAS/ACCESS interface.

- □ the name of the PSB.

Example PSB

Here is a sample of a PSB called ACCTSAM, which contains some database PCBs for the ACCTDBD database and one PCB for the WIRETRN database:

```
PCB    TYPE=DB,DBDNAME=ACCTDBD,PROCOPT=G,        X
       KEYLEN=11
SENSEG NAME=CUSTOMER,PARENT=0,PROCOPT=G
PCB    TYPE=DB,DBDNAME=ACCTDBD,PROCOPT=G,        X
       KEYLEN=23
SENSEG NAME=CUSTOMER,PARENT=0,PROCOPT=GP
SENSEG NAME=CHCKACCT,PARENT=CUSTOMER,PROCOPT=G
SENSEG NAME=SAVEACCT,PARENT=CUSTOMER,PROCOPT=G
PCB    TYPE=DB,DBDNAME=ACCTDBD,PROCOPT=A,        X
       KEYLEN=23
SENSEG NAME=CUSTOMER,PARENT=0,PROCOPT=AP
SENSEG NAME=CHCKACCT,PARENT=CUSTOMER,PROCOPT=AP
SENSEG NAME=CHCKDEBT,PARENT=CHCKACCT,PROCOPT=A
SENSEG NAME=CHCKCRDT,PARENT=CHCKACCT,PROCOPT=A
SENSEG NAME=SAVEACCT,PARENT=CUSTOMER,PROCOPT=AP
SENSEG NAME=SAVEDEBT,PARENT=SAVEACCT,PROCOPT=A
SENSEG NAME=SAVECRDT,PARENT=SAVEACCT,PROCOPT=A
PCB    TYPE=DB,DBDNAME=WIRETRN,PROCOPT=A,        X
       KEYLEN=23
SENSEG NAME=WIRETRAN,PARENT=0,PROCOPT=A
PSBGEN LANG=ASSEM,IOASIZE=500,PSBNAME=ACCTSAM, X
 CMPAT=YES
END
```

Security Options

IMS provides security for databases through data sensitivity, a way of controlling which data the SAS/ACCESS interface to IMS can access. The SAS/ACCESS interface is allowed to access only data to which it is sensitive. There are three levels of data sensitivity:

segment sensitivity
> enables the IMS interface to access only certain segments in a particular hierarchy.

field-level sensitivity
> enables the IMS interface to access only certain fields in a particular segment.

key sensitivity
> enables the IMS interface to access only segments below a particular segment in a hierarchy. It does not enable the IMS interface to access that particular segment, and returns only the segment's key to the interface.

The DBA can specify data sensitivity for an IMS database in each database PCB in the PSB.

DL/I Calls

The SAS/ACCESS interface to IMS accesses IMS database segments by issuing DL/I calls. A *DL/I call* is a request made by the interface to DL/I to access one or more segments of a database or message queue, or to perform some system function. Certain information must be specified in the DL/I call to communicate the SAS/ACCESS interface's request to DL/I. The normal information specified in a call is as follows:

☐ a *call function* specifying the action DL/I is to perform (get, insert, replace, and so on)

☐ a program view (PCB) in the PSB to use when performing the function. The PSB to be used has been specified earlier in your view descriptor or in your DL/I INFILE statement (for an IMS DATA step program).

☐ an I/O area to use for transferring segment data between DL/I and the SAS/ACCESS interface to IMS.

☐ up to 15 *segment search arguments* (SSAs). An *SSA* is a set of formatted search criteria that specifies the segment type or occurrence on which to perform the function.

Normally, a DL/I call accesses one segment at a time. However, by using a special command code in an SSA, the SAS/ACCESS interface to IMS can access multiple segments along a hierarchical path in the database. This type of call is a *path call*.

The following descriptions of the elements of a DL/I call do not need to be understood in detail by most users. They are included here for readers who want this additional information.

DL/I Call Functions

DL/I calls are categorized as Get calls or Update calls. A *Get call* is a call that retrieves (reads) a segment or segments. An *Update call* performs some kind of write function, such as inserting a new segment or replacing or deleting an existing segment.

The basic DL/I database call functions are listed here. Some of the descriptions refer to SSAs (segment search arguments), qualified calls, and unqualified calls. These are described in "Segment Search Arguments" on page 28.

GU   get-unique. If unqualified, this call retrieves (reads) the first segment in the PCB view (program view) of the database. If SSAs are specified, the call retrieves the first segment that satisfies qualifications specified by the SSAs.

GN   get-next. If unqualified, this call retrieves the next segment in the hierarchical sequence of the database. If SSAs are specified, the call retrieves the next segment that satisfies qualifications specified by the SSAs.

GNP   get-next-within-parent. This call is like the GN call but is restricted to the subtree of the current parent. (The parent is described in the PCB.)

GHU   get-hold-unique. This call is like the GU call but also holds the segment for the next update call that uses the same PCB.

GHN   get-hold-next. This call is like the GN call but also holds the retrieved segment for the next update call that uses the same PCB.

GHNP   get-hold-next-within-parent. This call is like the GNP call but also holds the segment for the next update call that uses the same PCB.

DLET   delete. This call deletes the segment retrieved by the last get-hold call using the same PCB.

REPL   replace. This call replaces the segment held from the last get-hold call using the same PCB with an updated segment that you provide. The get-hold call must be the last DL/I call that used the same PCB.

ISRT   insert. This call adds new segments using the PCB specified.

Program Communication Block

The SAS/ACCESS interface to IMS is comprised of two distinct interfaces: the IMS engine interface and the DATA step interface. The IMS DATA step interface can use any PCB. You can use DL/I INFILE statement extensions to specify the PCB. The DATA step interface also offers limited support for TP PCBs and message queue processing.

The IMS engine interface (that is, where you use view descriptors) uses only two types of PCBs: the Database (DB) PCB and the Input/Output (I/O) PCB. The engine interface uses the first DB PCB that matches the database you specify in the access descriptor, unless you specify otherwise in the PCB index field of the view descriptor. For updating, the engine interface to IMS uses the I/O PCB for checkpointing. The I/O PCB is also used if the kind of DL/I processing environment, or region type, is BMP. An I/O PCB is created when you enter CMPAT=YES in the PSBGEN statement.

Using a program view of the database (that is, using the appropriate PCB), a call can selectively access only the segments that are required by the SAS/ACCESS interface. For example, you might need the interface to retrieve savings account data from the ACCTDBD database without retrieving savings debit segments. The PCB defines as sensitive segments only those segments needed by the SAS/ACCESS interface.

A PCB to be used by the SAS/ACCESS interface that accumulates savings account credit information might use the program view shown in the following figure.

Figure 2.7 Program View of Savings Account Segments

```
┌──────────┐
│ CUSTOMER │
└──────────┘
     │
     └──────────┐
          ┌──────────┐
          │ SAVEACCT │
          └──────────┘
               │
               └──────────┐
                    ┌──────────┐
                    │ SAVECRDT │
                    └──────────┘
```
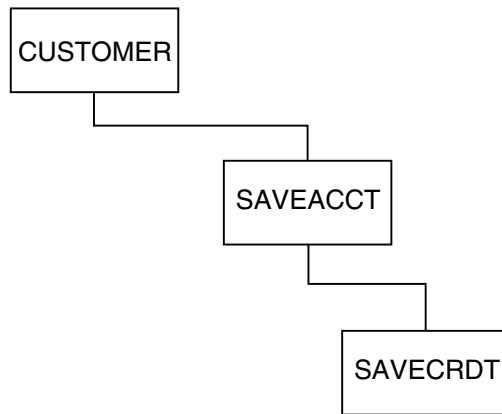
In addition to defining a program view of the database by specifying sensitive segments, the PCB also accumulates information about the results of a call. This information, called the *PCB mask data,* includes the following elements:

segment level
    is the hierarchical level of the last segment successfully retrieved or processed.

DL/I status code
    is a return code that indicates whether the call was successful.

DL/I processing option
    is a code that indicates what kind of access to the database is allowed. The processing option might be one of the following:

G	for Get
D	for Delete, includes G
I	for Insert
R	for Replace, includes G
A	for All of the above
E	for Exclusive use, in conjunction with G,D,I,R,A
L	for Loading database, excludes HISAM
LS	for Loading sequentially, required for HISAM
O	for inhibiting program isolation, must be used with G
P	for Path calls
GS	for getting segments in ascending sequence

segment name
    is the name of the last segment type successfully retrieved or processed by the call.

key feedback data
    is the concatenated key (sequence) field values of all segments in the path between the root segment and the last segment that was successfully retrieved or processed.

The PCB mask data also contains other information not described here. *

Database Position

For each PCB, DL/I maintains a *current position indicator.* The position indicator points to the last segment accessed or to the top of the database, if no DL/I call has been issued or if the last call failed. The position determines which segment should be processed next, that is, by the current DL/I call.

Suppose your DATA step program uses a PCB that defines the CUSTOMER, CHCKACCT, and SAVEACCT segments as sensitive segments. The program is a read-only program and unqualified GN (get-next) calls are issued. Therefore, the program uses sequential processing. The program view is shown in the following figure.

Figure 2.8   Program View of Account Segments
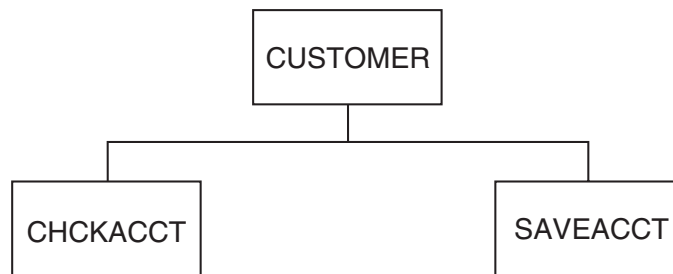
When the first GN call is issued, DL/I is positioned at the front of the database and the call retrieves the first occurrence of a CUSTOMER segment. When the next call is issued, DL/I uses the current position to determine which segment is retrieved next. In this case, CHCKACCT is retrieved before SAVEACCT because the default search sequence for sequential access is top to bottom, left to right.

Note: The database position is influenced by considerations that are not described here, such as the type of call issued and certain command codes. △

Segment Search Arguments

A DL/I call can be qualified or unqualified. A *qualified call* is one that specifies one or more SSAs (segment search arguments). An SSA provides additional information for the DL/I call. The simplest SSA identifies a segment type for the call to access. Other SSAs not only identify the segment type, but they also specify a value or a set of values to select a particular segment occurrence. An *unqualified call* does not have any SSAs and, therefore does not specify a particular segment or set of segments.

If an SSA describes only the segment type to be accessed, it is an unqualified SSA. (The call is still a qualified call, but the SSA itself is unqualified.) In an unqualified SSA, you can also specify an optional *command code,* which might affect how the call function is performed or it might affect the qualification of a segment. See "Command Codes" on page 30 for more information.

An unqualified SSA has the form

segment-name <command code>

where *segment-name* is an 8-byte field specifying the segment type, followed by a blank. A blank follows because the minimum SSA length for a DL/I call is 9 bytes. Command codes consist of an asterisk (*) followed by a letter, such as *U or *D.

---

*   The IMS DATA step programs can return PCB mask data to the user, but the SAS/ACCESS engine interface cannot.

If an SSA provides a field name and specific value for that field, it is a qualified SSA. A qualified SSA has the form

segment-name <command code> (*field-name operator value . . .*)

where *segment-name* is the 8-byte segment type, *field-name* is the 8-byte name of a sequence or search field for that segment as defined in the DBD, and *operator* is a 2-byte field that contains a comparison operator. *Value* is a value that is compared to the specified field in the segment. The values for each of the segment type, field name, operator, and value must be padded to represent the total number of bytes used by the particular field in the DBD. IMS requires the padding.

The first segment occurrence that satisfies the qualification or qualifications is retrieved.

For example, to retrieve a CUSTOMER segment for Hooper J. Walls, the Get (retrieve) call would be qualified with this qualified SSA:

```
CUSTOMER(CUSTNAME =WALLS, HOOPER J.          )
```

The comparison operators that IMS uses in a qualified SSA, along with their alternate forms, are listed in the following table.

Table 2.5 Comparison Operators and Their Equivalents

Operator	Alternate Form
=	= or EQ*
>	> or GT
<	< or LT
>=	=> or GE
<=	=< or LE
¬=	=¬ or NE
&	or AND (dependent AND)
|	+ or OR (logical OR)

* Pad the =, >, and < operators with blanks on the right or left.

The IMSWHST= Option for Qualified SSAs

The SAS/ACCESS interface provides certain system and configuration options to use with IMS. One such configuration option, IMSWHST=, affects qualified SSAs and applies only to the IMS engine interface.

IMSWHST=Y makes sure that any specified WHERE criteria have been incorporated into the SSAs that are generated by the IMS engine. Doing so limits the amount of data that the database returns.

If qualified SSAs are not generated by the view descriptor's or application's WHERE statement (or there is no WHERE data set option), the software issues an error message, no IMS records are retrieved, and processing stops.

The default, IMSWHST=N, specifies that IMS records are retrieved for processing regardless of whether or not qualified SSAs are passed to IMS by a view descriptor's or application's WHERE statement.

Multiple SSAs in the DATA Step Interface

If you use the IMS DATA step interface and specify more than one SSA for a call, you must specify them in hierarchical order. You can specify as many as 15 SSAs. The segment specified in the last SSA, the *target segment,* is the segment accessed.

For example, if you want to issue a GN (get-next) call to retrieve a CHCKDEBT segment with a DEBTDATE of 28 March 1995 for banking customer Mary T. Summers, you would qualify the GN call with these SSAs:

```
CUSTOMER*U-(CUSTNAME =SUMMERS, MARY T.            )
CHCKDEBT(DEBTDATE =032895)
```

The target segment for the call is CHCKDEBT. It is the only segment returned.

To access more than one segment in one call, you must set up a *path call*, as explained in "Command Codes."

Command Codes

Any SSA, qualified or unqualified, can include a *command code.* A command code provides still more information for the call. It might affect how the call function is performed, or it might affect the qualification of a segment. Command codes consist of an asterisk (*) followed by a letter.

One commonly used command code is *D, which signifies a path call. For a DL/I GET call, this means that the segment named in the SSA with the *D code is returned, even if it is not the target segment (the segment named in the last SSA). The segments retrieved with SSAs that specify *D are returned to the I/O area in hierarchical sequence. The target segment is placed in the I/O area behind the segments whose SSAs specified *D. (This has no effect on what is returned in the key feedback area; it affects only the I/O area.)

For example, one PCB defines CUSTOMER, CHCKACCT, and CHCKDEBT as sensitive segments. (Figure 2.5 on page 17 shows this program view.) In the IMS DATA step interface, you specify these SSAs and a GU (get-unique) call function. Two segments are returned to the I/O area: CUSTOMER and CHCKDEBT.

```
CUSTOMER*D-(CUSTNAME =WALLS, HOOPER J.           )
CHCKACCT
CHCKDEBT(DEBTDATE =030594)
```

The PCB mask data will contain CHCKDEBT as the name of the last segment successfully retrieved, and the key feedback data contains the concatenated key fields of the CUSTOMER and CHCKACCT segments. The CHCKDEBT segment has only a search field and no sequence field, and therefore no data for CHCKDEBT is in the key feedback area.

IMS Execution Modes

When the SAS/ACCESS interface to IMS accesses DL/I databases, it executes within a DL/I subsystem. DL/I subsystems are either batch or online:

□ Usually, an *online DL/I subsystem* is used by multiple terminals or programs at the same time, and databases are shared by the users. The terminal users (for example, bank tellers or airline reservation agents) execute preprogrammed applications to access DL/I databases. These users might be executing the same SAS program or different SAS programs.

□ In a *batch DL/I subsystem,* only one program executes at a time, and it has exclusive use of the databases. Batch subsystems are typically used when one or more functions must be executed repetitively (for example, printing customers' monthly bank statements), and the database is not required for concurrent access by another subsystem.

Batch and online DL/I subsystems can execute concurrently. For example, a bank might run an online subsystem to which all the bank teller terminals are connected. As customers make deposits and withdrawals during the day, the tellers use checking and savings application functions to record these transactions and update a database such as the ACCTDBD database. Simultaneously, a batch DL/I subsystem might execute a SAS program to print a report of all loans with overdue payments from a loan database.

It is important to know how the SAS system options for the SAS/ACCESS interface to IMS are set at your site. These SAS system options determine the execution mode of the SAS/ACCESS interface, which must be consistent with your IMS system configuration.

Outline of a Batch DL/I Subsystem

In a batch DL/I subsystem, a *batch region* is a processing environment for running batch mode jobs using a local batch control program. The batch region is initialized by a *region controller*, which is the primary entry point for all DL/I executions. The region controller initializes the batch region according to JCL (job control language) specifications made when the user's SAS program is submitted.

When the program runs in the batch region, the SAS/ACCESS interface to IMS communicates with DL/I to access DL/I databases and to issue calls against databases. DL/I also handles the DL/I log, which contains information that is needed to recover changes to the database if the program terminates abnormally.

The batch region in which the SAS/ACCESS interface executes is either a DLI region or DBB region. The use of these regions depends on the operating system. Under OS/390, a *DBB region* uses the Application Control Block library (ACBLIB) to get the DBDs and PSBs. A *DLI region* uses the Database Description library (DBDLIB) and Program Specification Block library (PSBLIB) to get the DBDs and PSBs. The following figure shows the typical batch DL/I subsystem.

Figure 2.9 Typical Batch DL/I Subsystem
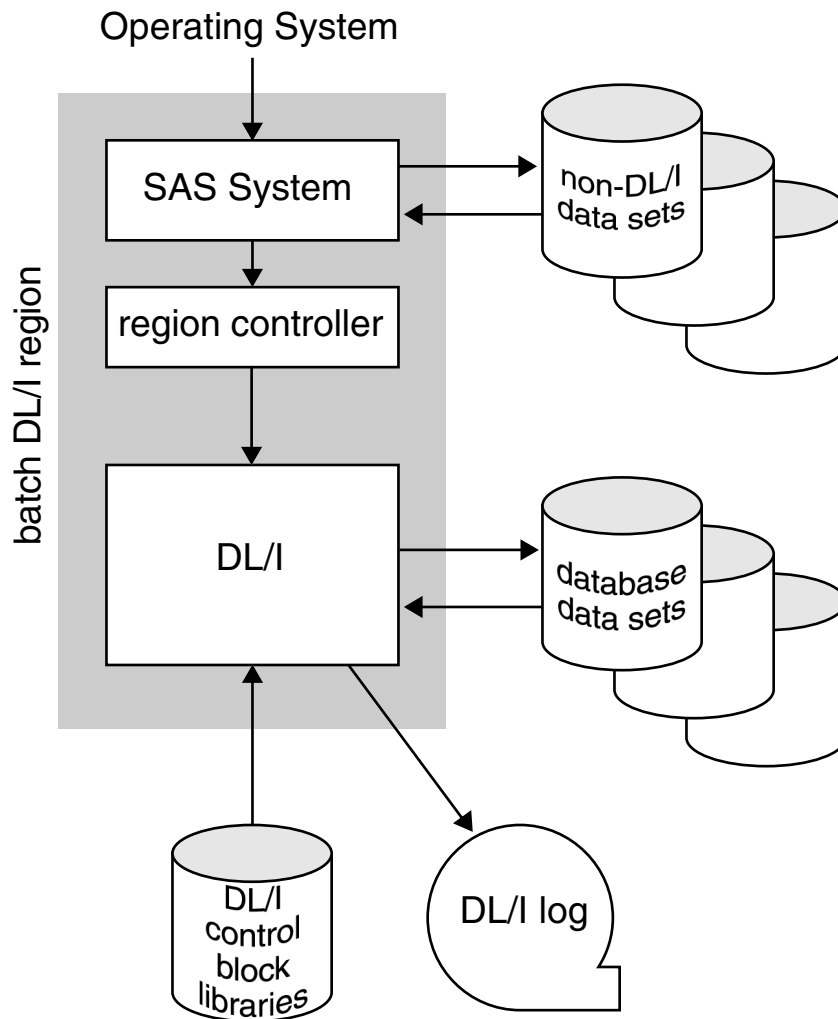
The following steps are performed when a SAS program is executed in the batch DL/I subsystem:

1 The operating system passes control to SAS. When the SAS/ACCESS interface initializes, it attaches a subtask to execute the DL/I region controller. Parameters that are passed to the region controller specify the type of batch region to execute (DLI or DBB), the name of the program (IMSEXEC under OS/390), PSB to use, and other execution options.

2 The region controller establishes the DL/I region environment and passes control to the SAS/ACCESS interface.

3 The SAS/ACCESS interface receives pointers to the PCBs in the PSB. It uses these PCBs in DL/I calls.

4 The SAS/ACCESS interface formats a DL/I call and passes control to DL/I to access DL/I databases.

5 DL/I accesses the database data sets, performs the requested call function, and logs any information required for recovery in the DL/I log.

6 A return code and other information for the PCB mask are placed in the PCB, and control returns to the SAS/ACCESS interface.

7 Steps 4 through 6 are repeated until the SAS procedure or SAS DATA step is completed. The region controller subtask is detached and SAS continues to process the other SAS PROC or DATA steps.

Outline of an Online DL/I Subsystem

In an online DL/I subsystem, an *online control region* is initialized and uses JCL specifications to set up the environment in which user programs execute. Under OS/390, types of online control regions include IMS/ESA DB/DC regions or CICS regions.

An online control region also allocates and controls access to DL/I database data sets for multiple-user programs, ensuring the integrity of the databases being used by many programs. Normally, the online control region obtains exclusive control of the database data sets so that other DL/I subsystems do not update the database data sets concurrently. This preserves database integrity within the overall system.

When the online control region allocates a database, it is referred to as an *online database*. The ACCTDBD database is an online database when it is allocated to an online subsystem. When the online control region is terminated, any associated databases can be used in a batch processing region. Databases can be freed to allow access by a batch program concurrent with online control region execution. Alternatively, batch and online processing can concurrently share access to databases by using IMS/ESA data sharing support.

The SAS/ACCESS interface to IMS interacts with an online control region through a DL/I online access region. The *online access region* is used when a batch program requires access to a database allocated by the online control region, that is, to an online database. There are two types of online access regions under OS/390:

- □ a BMP region is used to access an IMS/ESA DB/DC online control region
- □ a BMP region is used to access a CICS region (the DBCTL facility of IMS/ESA provides this functionality).

For example, the ACCTDBD database must be updated periodically with another database, which contains information about transactions using automated teller machines (ATMs). There is a batch program to read this transactions database and update the ACCTDBD database. However, because the ACCTDBD and transaction database data sets are allocated exclusively to the online subsystem for the tellers, a batch subsystem cannot allocate the data sets. This kind of conflict is resolved by an online access region, in which a batch program executes but issues the DL/I calls under the control of the online control region. This method preserves the integrity of the online databases.

The typical online access region and online control region interaction is depicted in the following figure.

Figure 2.10 Typical Online DL/I Subsystem
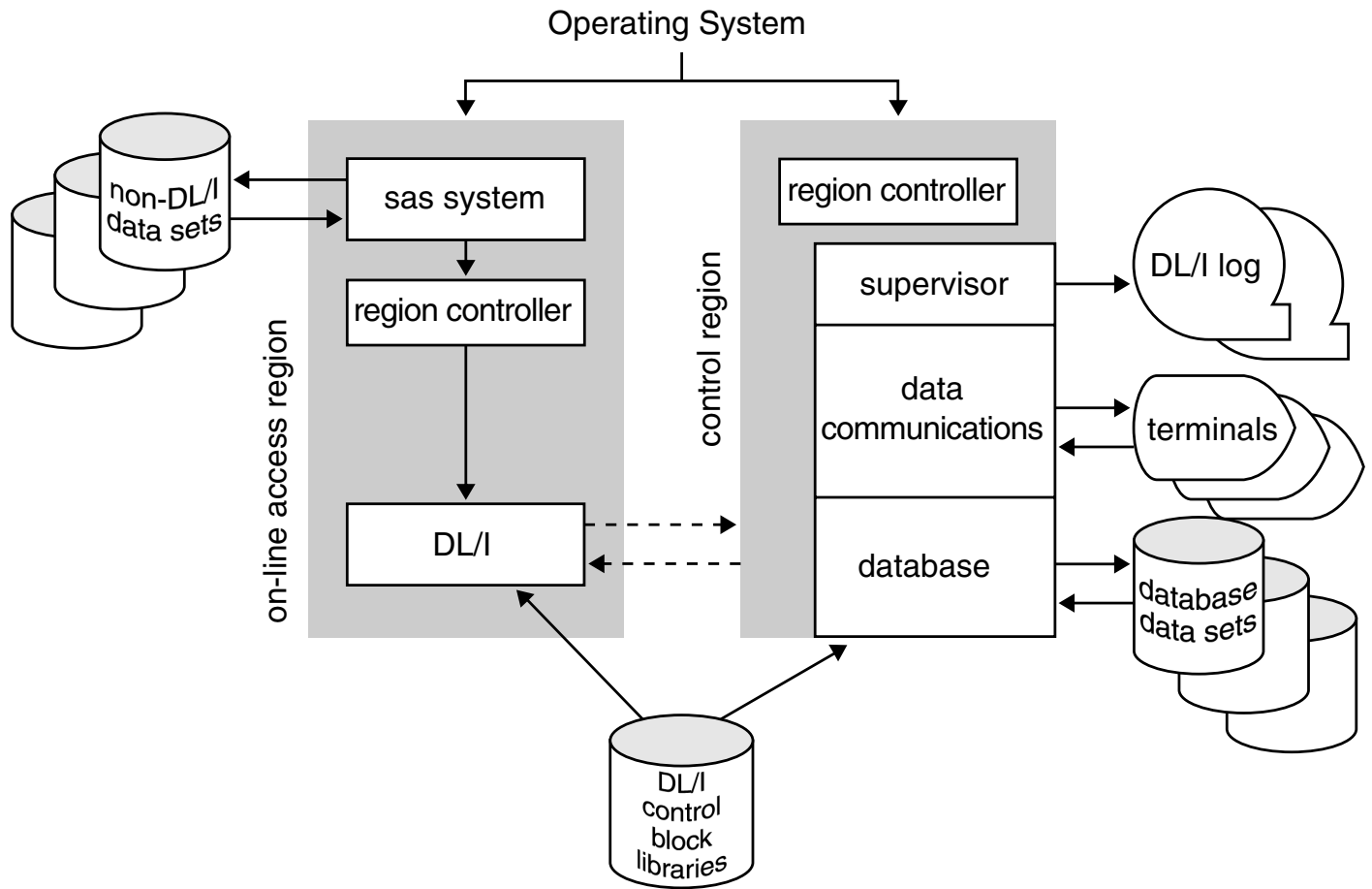
Summary of Region Types

The following table summarizes the region types that are used in the various DL/I subsystems discussed in this section.

Table 2.6 Summary of Region Types

IBM Product	Type of Subsystem	Database Controlled by	Batch Region	Online Access Region
IMS/ESA DB	batch	region controller	DLI or DBB	
IMS/ESA DB/DC	online	control region		BMP
CICS	online	control region		BMP

Shared IMS Database Access

Each of the IBM IMS products provides some capability for sharing IMS resources. Two general categories of sharing exist:

□ sharing resources within one IMS subsystem

    ☐ sharing resources between multiple IMS subsystems.

The concepts of read integrity and update integrity are important in a description of resource sharing. *Read integrity* means that two programs cannot access a record simultaneously if one has update intent. Read integrity guarantees that the data is current when reading a record. *Update integrity* means that two programs cannot access a record simultaneously if both have update intent. Update integrity guarantees that data accessed for update is current, but it does not guarantee that data accessed for reading is current.

Resource sharing within one subsystem is the most common form of resource sharing and is available with an online IMS subsystem. In the online IMS subsystem, the online IMS control region allocates the database data sets and controls concurrent access to the databases by multiple programs. Read integrity is guaranteed when sharing within an online subsystem unless the processing option GO has been specified in the PCB. For more information about the GO option, see Chapter 2, "Program Specification Block (PSB) Generation," in the *IMS/ESA Utilities Reference Manual*, or *DL/I Resource Definition and Utilities*. Update integrity is always guaranteed in an online subsystem.

In the second form of sharing, sharing resources between multiple IMS subsystems, there are two subcategories:

    ☐ *Database-level sharing* enables multiple IMS subsystems to access a database concurrently. Both online and batch regions can be used.

        One subsystem can update a database while other subsystems access the same database in read-only mode. When sharing takes this form, update integrity is guaranteed, but read integrity is not guaranteed. Read integrity is guaranteed only if all subsystems use read-only access.

        Database-level sharing is available in IMS/ESA DB and IMS/ESA DB/DC systems.

    ☐ *Block-level sharing* enables multiple IMS subsystems to have concurrent update access to a database.

When sharing resources, IMS preserves both read and update integrity.

Note: GSAM databases cannot be shared. △

General Considerations for Sharing Resources

When resources are shared, whether within a subsystem or between subsystems, many users can access a given database at the same time. Consequently, one invocation of the SAS/ACCESS interface to IMS can have an impact on the performance of several users' programs.

When read integrity is guaranteed, the SAS/ACCESS interface has read-only access and *owns* (has exclusive access to) the last database record it accessed. Even under these circumstances, the SAS/ACCESS interface with read-only access does not normally affect the performance of other programs. However, if the SAS/ACCESS interface is positioned on one database record for a long time, it affects other programs by preventing them from accessing that record. If read integrity is not guaranteed, the SAS/ACCESS interface does not own records and, therefore, does not affect other programs.

The SAS/ACCESS interface is more likely to affect the performance of other programs if it updates database records. When the SAS/ACCESS interface updates records, it owns any record that has been updated since the interface's last synchronization point. A *synchronization point* occurs when the SAS/ACCESS interface issues a CHKP (checkpoint) call. This synchronization point saves the changes the SAS/ACCESS interface has made since the last CHKP call it issued to the database. By default, the SAS/ACCESS interface issues CHKP calls at the beginning and end of

processing. With SAS/FSP software, use the AUTOSAVE option to increase the frequency of issuing CHKP calls.

Synchronization points are important because they cause IMS to release some resources allocated to the SAS/ACCESS interface. These resources include the database records owned by the interface, the IMS enqueue table entries that mark this ownership, and the dynamic log records required to back out (cancel) updates since the prior synchronization point. When IMS releases the SAS/ACCESS interface's ownership of updated database records, other programs can access the record with the updated information.

Database-Level Shared Access

In database-level shared access, multiple IMS subsystems (batch or online or both) allocate the database data sets concurrently. Concurrent allocation is possible in a single operating system with shared disposition allocation. It might be possible between multiple operating systems, regardless of the allocation disposition, if the database data sets reside on shared Direct Access Storage Device (DASD).

CAUTION:
> **If the IMS requirements for database-level sharing are not followed closely,** IMS database integrity can be compromised by multiple allocations. Be sure that database-level sharing or block-level sharing is implemented for a database before you allocate a database data set with shared disposition. △

In database-level sharing, one subsystem can have update access to a database while other subsystems have read access to the same database. In this case, update integrity is guaranteed, but read integrity is not guaranteed. Alternatively, all subsystems can be restricted to read access, in which case read integrity is guaranteed because there is no danger of a record being updated. The remainder of this section on database-level sharing discusses sharing when one subsystem has update access and other systems have read access.

When one subsystem has update access and the others have read access, it is possible for a read-access invocation of the SAS/ACCESS interface to obtain uncommitted update data from a program that later backs out the updates.

If the subsystem with update access is a batch subsystem, only one program or invocation of the SAS/ACCESS interface has update access to the database (since only one program executes in a batch subsystem). No other program or invocation of the interface with update intent (indicated in the PCB) can execute until the first subsystem completes, so there is no contention for the database records. (Remember that read integrity is not guaranteed in this situation and programs with read access do not own records.) Since other executing programs are not waiting for records, you do not have to be concerned about releasing records for other programs to use.

If the subsystem with update access is an online subsystem, other subsystems (whether batch or online) are still restricted to read access. However, unlike a batch subsystem, multiple programs in the update-access online subsystem can update the database. In other words, two forms of sharing occur at once:

□ database-level sharing between subsystems, with one updating and others reading

□ sharing within one online subsystem, with multiple programs sharing the databases.

Database-level sharing is specified by completing the following tasks:

□ registering the database with Database Recovery Control (DBRC) for database-level sharing

□ ensuring that DBRC is used in the IMS/ESA IMS region

□ specifying a share option of (2,3) or (3,3) when the VSAM data set is defined.

Under OS/390, if DBRC is not used, database integrity is compromised. DBRC is active in SAS executions of application regions as long as the value of the SAS system option IMSDLDBR= is not N.

Block-Level Shared Access

In block-level shared access, multiple IMS subsystems allocate the database data sets concurrently. This shared allocation is possible in a single operating system with shared disposition allocation. Block-level shared access is possible between multiple operating systems regardless of the allocation disposition if the database data sets reside on shared DASD.

If the IMS requirements for block-level sharing are not followed completely, the IMS database integrity might be compromised by this multiple allocation. Be sure that you implement block-level sharing for a database before you allocate a database data set with shared disposition.

Block-level shared access differs from database-level shared access in that it guarantees both read and update integrity for the shared database. It is not possible for the SAS/ACCESS interface to IMS to obtain uncommitted update data that is later backed out.

A disadvantage of block-level sharing is that different subsystems must contend for database records. Therefore, synchronization-point processing becomes essential when updating a database that is shared at the block level with other IMS subsystems.

An advantage of block-level sharing over database-level sharing is that the SAS/ACCESS interface that updates does not have to wait to obtain exclusive update control of the database.

Block-level sharing is specified by completing the following tasks:

□ registering the database with DBRC for block-level sharing

□ ensuring that DBRC is used in the application region

□ establishing communication with an IMS/ESA Resource Lock Manager (IRLM), which is executing under the same operating system as the IMS region

□ specifying (for VSAM data sets) a share option of (3,3) when the VSAM data set is defined.

If DBRC is not active, database integrity is compromised. If DBRC was included in IMS/ESA during operating system generation, DBRC is active in SAS executions of application regions as long as the SAS system option IMSDLDBR= does not have a value of N.

Similarly, if communication with the IRLM is not established, database integrity is compromised. The IMS region establishes communication with the IRLM specified by the SAS system option IMSDLIRN= as long as the IRLM is active and the SAS system option IMSDLIRL= does not have a value of N.

P A R T *2*
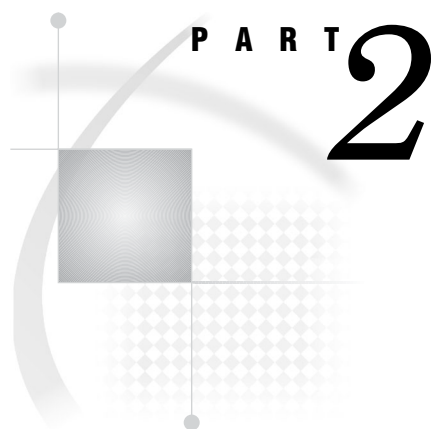
The IMS Engine Interface: Usage

CHAPTER

3
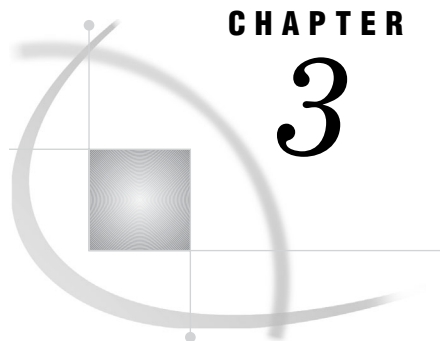
Defining SAS/ACCESS Descriptor Files

Introduction to Defining SAS/ACCESS Descriptor Files

To use the SAS/ACCESS to IMS interface view engine, you must define special files that describe IMS databases and data to SAS. These files are called SAS/ACCESS descriptor files. This section uses examples to show you how to create and edit descriptor files.

The examples in this section are based on the IMS database WIRETRN. Complete information about the WIRETRN database is provided later in this section. From this database, the examples create an access descriptor. Then, the examples create a view descriptor based on the access descriptor. For complete reference information about the ACCESS procedure, see Chapter 6, "ACCESS Procedure Reference," on page 95.

SAS/ACCESS Descriptor Files Essentials

One way that SAS interacts with IMS databases is through an interface view engine that makes use of SAS/ACCESS descriptor files created with the ACCESS procedure. There are two types of descriptor files:

- access descriptors
- view descriptors.

An *access descriptor* contains information about the IMS database that you want to use. The information includes the IMS database name, the IMS field names and their default SAS formats, database formats, segment names and lengths, and key fields. An access descriptor also contains any special handling considerations for a field and indicates if an item occurs multiple times in a database segment. You use the access descriptor to create view descriptors. An access descriptor is like a master descriptor file for a single IMS database because it contains a complete description of that database (if you choose to enter all the data). Because IMS does not store descriptive information about a database, you must enter the database definition in the access descriptor.

A *view descriptor* defines a subset of the data that is described by an access descriptor.

Note: This subset must contain data from only one path in the database on which the access descriptor is based. You choose this subset by selecting particular items and specifying criteria that the data must meet. △

For example, you might want to select two items, CUSTOMER_NAME and STATE, and specify that the value stored in item STATE must equal **NC**.

A view descriptor is a SAS data set of member type VIEW. After you create your view descriptors, you can use them in a SAS program to read or write the data directly from and to an IMS database, or you can extract IMS data and place it in a SAS data file. Typically, you have several view descriptors (each selecting a different path of data in the database) for each access descriptor that you have defined.

Creating and Using Descriptor Files

You can use the ACCESS procedure in batch mode to create the access descriptor MYLIB.WIRETRN and the view descriptor VLIB.WIREDATA. Because IMS does not have a dictionary or store descriptive information about IMS databases, you must provide the database definition in the SAS statements following the procedure statement. You can also create view descriptors in the same PROC ACCESS execution after the access descriptor statements are entered. (See Chapter 6, "ACCESS Procedure Reference," on page 95 for a list of valid options that you can use with PROC ACCESS.) Here is the general format for creating descriptors:

```
proc access options;
    statements;
run;
```

Creating Access and View Descriptors in One PROC Step

Perhaps the most common way to use the ACCESS procedure is to create an access descriptor and one or more view descriptors during a single PROC ACCESS execution.

The following example shows how to create the access descriptor MYLIB.WIRETRN based on the IMS database WIRETRN. The view descriptor VLIB.WIREDATA is based on this access descriptor. After the following example, each SAS/ACCESS statement is explained in the order of appearance in the program:

```
JCL statements;

libname mylib 'access-descriptor libref';
libname vlib 'view-descriptor libref';

  proc access dbms=ims;
    create mylib.wiretrn.access;
      database=wiretrn dbtype=hdam;
      record='wire transaction' segment=wiretran
                                  seglng=100;
        item='ssn - account' level=2 dbformat=$23.
                              search=ssnacc key=y;
        item='account type'  level=2 dbformat=$1.
                              search=accttype;
```

```
        item='wire date'      level=2 dbformat=$8.
                                  search=wiredate;
        item='wire time'      level=2 dbformat=$8.
                                  search=wiretime;
        item='wire amount'    level=2 dbformat=pd5.2
                                  search=wireammt
                                  dbcontent=l;
        item='wire descript' level=2 dbformat=$40.
                                  search=wiredesc;
    an=y;
    list all;

    create vlib.wiredata.view psbname=acctsam
      pcbindex=5;
      select 'wire transaction';
    list view;
  run;
```

Here is an explanation of the statements in this example. See "Invoking the ACCESS Procedure" on page 100 for complete reference information about these statements.

JCL statements;
   included for batch and noninteractive line modes.

libname mylib='access-descriptor libref';
libname vlib='view-descriptor libref';
   reference the SAS data library in which you will store the access descriptor
   (MYLIB) and the SAS data library in which you will store the view descriptors
   (VLIB). You must associate a libref with its data library before you can use it in
   another SAS statement or procedure.

proc access dbms=ims;
   invokes the ACCESS procedure for the SAS/ACCESS interface to IMS.

create mylib.wiretrn.access;
   identifies the access descriptor, MYLIB.WIRETRN, that you want to create.

database=wiretrn dbtype=hdam;
   specifies the IMS database named WIRETRN on which the access descriptor is to
   be created. The database type is HDAM.

record='wire transaction' segment=wiretran seglng=100;
   specifies the user-specified record name, as well as the segment name and segment
   length, as specified in the IMS DBD for the WIRETRN database.

item='ssn - account' level=2 dbformat=$23. search=ssnacc key=y;
   identifies the item SSN - ACCOUNT. It has an internal format of type character,
   length 23 bytes. SSNACC is specified as a search field name. KEY=Y indicates
   that SSN - ACCOUNT is listed in the DBD as a key field for the WIRETRAN
   segment.

item='account type' level=2 dbformat=$1. search=accttype;
   identifies the item ACCOUNT TYPE with an internal format of type character,
   length 1 byte. ACCTTYPE is specified as a search field in the DBD.

item='wire date' level=2 dbformat=$8. search=wiredate;
   identifies the item WIRE DATE with an internal format of type character, length 8
   bytes. The search field WIREDATE is specified.

item='wire time' level=2 dbformat=$8. search=wiretime;
identifies the item WIRE TIME with the same attributes as WIRE DATE except it has the search field name WIRETIME.

item='wire amount' level=2 dbformat=pd5.2 search=wireammt
dbcontent=l;
identifies the item WIRE AMOUNT with a packed decimal database format of 5 bytes with 2 decimal places. DBCONTENT=L indicates that SAS should display a missing value when it finds low values (hexadecimal zeros) for this item. The search field is WIREAMMT.

item='wire descript' level=2 dbformat=$40. search=wiredesc;
identifies the item WIRE DESCRIPT with an internal format of type character, length 40 bytes. The search field is WIREDESC.

an=y;
generates unique SAS variable names and default formats based on the name of the IMS item and its DBFORMAT= value. Using AN=Y in an access descriptor means no changes can be made to the SAS names and formats in any view descriptors that use the access descriptor.

list all;
lists all the items in the access descriptor and SAS information for each item. The output is displayed in the SAS log.

create vlib.wiredata.view psbname=acctsam pcbindex=5;
creates a view descriptor called WIREDATA which references PSB ACCTSAM. The PCBINDEX=5 statement refers to the specific PCB in the PSB to be used at execution time.

select 'wire transaction';
selects the WIRETRAN segment of the IMS database to be included in the view, as defined in the access descriptor.

list view;
lists the SAS information about the record WIRE TRANSACTION you selected for this view. Output from this statement is shown in the SAS log.

run;
forces the execution of the ACCESS procedure.

Creating Access and View Descriptors in Separate PROC Steps

You can create view descriptors and access descriptors in separate PROC ACCESS steps. In the first PROC ACCESS step in the following example, you create the access descriptor MYLIB.WIRETRN, which is based on the WIRETRN database. In the second PROC ACCESS step, you create a view descriptor, VLIB.WIREDATA, which is based on the access descriptor MYLIB.WIRETRN.

```
proc access dbms=ims;
    create mylib.wiretrn.access;
      database=wiretrn dbtype=hdam;
      record='wire transaction' segment=wiretran
                                seglng=100;
        item='ssn - account' level=2 dbformat=$23.
                             search=ssnacc
                             key=y;
        item='account type'  level=2 dbformat=$1.
```

```
                                     search=accttype;
           item='wire date'     level=2 dbformat=$8.
                                     search=wiredate;
           item='wire time'     level=2 dbformat=$8.
                                     search=wiretime;
           item='wire amount'   level=2 dbformat=pd5.2
                                     search=wireammt
                                     dbcontent=l;
           item='wire descript' level=2 dbformat=$40.
                                     search=wiredesc;
      an=y;
      list all;
    run;

    proc access dbms=ims accdesc=mylib.wiretrn;
      create vlib.wiredata.view psbname=acctsam
          pcbindex=5;
        select 'wire transaction';
      list view;
    run;
```

   Note that the statement **proc access dbms=ims** is repeated in this example. See
"Creating Access and View Descriptors in One PROC Step" on page 42 for complete
reference information about this statement.

Using View Descriptors in SAS Programs

   You can use a view descriptor in any SAS procedure in which you could use other
SAS data sets. The next two examples include printing and reviewing variables for a
view descriptor.

Printing Data

   Printing IMS data that is described by a view descriptor is like printing any other
SAS data set, as shown in the following example:

```
options nodate linesize=120;

proc print data=vlib.wiredata;
    title2 'Wire Transactions';
run;
```

   The following output shows the output for the VLIB.WIREDATA view descriptor.

Output 3.1 Results of the PRINT Procedure

```
                             The SAS System
                            Wire Transactions

  OBS  SSN_ACCOUNT            ACCOUNT_TYPE  WIRE_DATE WIRE_TIME   WIRE_AMOUNT  WIRE_DESCRIPT

   1  335-45-3451345620145345     C         03/31/95  15:42:43    1563.23     BAD CUST_SSN
   2  434-62-1224345656336366     L         03/30/95  23:45:32     424.87     WIRED FROM SCNB 37262849393
   3  156-45-5672345689435776     S         04/06/95  12:23:42    -150.00     WIRED TO BOA 9383627274
   4  456-45-3462345620134522     C         04/06/95  13:12:34    -245.73     WIRED TO WELLS FARGO CHICAGO
   5  234-74-4612345689413263     S         04/06/95  15:45:42    -238.73     WIRED TO WELLS FARGO SAN FRANCISCO
   6  667-73-8275345620154633     S         03/31/95  15:42:43    1563.23     BAD ACCT_NUM
   7  234-74-4612345620113263     C         04/06/95  11:12:42    1175.00     WIRED FROM SCNB 73653728343
   8  156-45-5672345620123456     C         04/06/94  10:23:53    -136.29     WIRED TO SCNB 53472019836
   9  156-45-5672345620123456     C         04/06/95   9:35:53    1923.87     WIRED FROM CIBN 37284839328
  10  434-62-1224345620134564     C         04/06/95  13:23:52    -284.42     WIRED TO TVNB 837362636438
  11  667-73-8275345689454633     C         03/28/95  15:42:43    1563.23     BAD ACCT_NUM
```

When you use the PRINT procedure, you might want to take advantage of the OBS= and FIRSTOBS= data set options. The OBS= option enables you to specify the last observation to be processed; the FIRSTOBS= option enables you to specify the first. The options are not valid with any form of the WHERE expression. The OBS= option improves performance when the view descriptor describes a large amount of data and you just want to see an example of the output. Because each record must still be read and its position calculated, using the FIRSTOBS= option does not improve performance significantly. The POINT= and KEY= options of the MODIFY and SET statements are not currently supported by the IMS engine.

The following example uses the OBS= data set option to print the first five observations of data described by the view descriptor VLIB.WIREDATA, which describes the WIRETRAN segment of the IMS database WIRETRN:

```
options nodate linesize=120;

proc print data=vlib.wiredata(obs=5);
   title2 'First Five Observations Described by
          VLIB.WIREDATA';
run;
```

The following table shows the result of this example.

Output 3.2   Results of Using the FIRSTOBS= Option

```
                                  The SAS System
                      First Five Observations Described by VLIB.WIREDATA

  OBS   SSN_ACCOUNT              ACCOUNT_TYPE   WIRE_DATE   WIRE_TIME    WIRE_AMOUNT   WIRE_DESCRIPT

   1  335-45-3451345620145345       C          03/31/95    15:42:43      1563.23      BAD CUST_SSN
   2  434-62-1224345656336366       L          03/30/95    23:45:32       424.87      WIRED FROM SCNB 37262849393
   3  156-45-5672345689435776       S          04/06/95    12:23:42      -150.00      WIRED TO BOA 9383627274
   4  456-45-3462345620134522       C          04/06/95    13:12:34      -245.73      WIRED TO WELLS FARGO CHICAGO
   5  234-74-4612345689413263       S          04/06/95    15:45:42      -238.73      WIRED TO WELLS FARGO SAN FRANCISCO
```

For more information about the PRINT procedure, see *SAS Language Reference: Concepts* and *Base SAS Procedures Guide*. For more information about the OBS= and FIRSTOBS= options, see *SAS Language Reference: Dictionary*.

Reviewing Variables

If you want to use IMS data that is described by a view descriptor in your SAS program, you can use the CONTENTS or DATASETS procedure to display the view's variable and format information. You use these procedures with view descriptors in the same way you use them with other SAS data sets.

The following example uses the DATASETS procedure to give you information about the view descriptor VLIB.WIREDATA, which describes the data in the WIRETRAN segment of the IMS database WIRETRN:

```
options nodate linesize=132;

proc datasets library=vlib memtype=view;
   contents data=wiredata;
   title2 ' ';
run;
```

The following output shows the first display of the information for this example.

Output 3.3   Results of Using the DATASETS Procedure with a View Descriptor

```
                           DATASETS PROCEDURE

         Data Set Name: VLIB.WIREDATA          Observations:          .
         Member Type:   VIEW                   Variables:             6
         Engine:        SASIOIMS               Indexes:               0
         Created:       .                      Observation Length:    88
         Last Modified: .                      Deleted Observations: 0
         Protection:                           Compressed:            NO
         Data Set Type:                        Sorted:                NO
         Label:

                 -----Engine/Host Dependent Information-----


             -----Alphabetic List of Variables and Attributes-----

     #     Variable        Type    Len    Pos    Format    Informat    Label
     -------------------------------------------------------------------------
     2     ACCOUNT_TYPE    Char      1     23     $1.       $1.         ACCOUNT TYPE
     1     SSN_ACCOUNT     Char     23      0     $23.      $23.        SSN - ACCOUNT
     5     WIRE_AMOUNT     Num       8     40     12.2      12.2        WIRE AMOUNT
     3     WIRE_DATE       Char      8     24     $8.       $8.         WIRE DATE
     6     WIRE_DESCRIPT   Char     40     48     $40.      $40.        WIRE DESCRIPT
     4     WIRE_TIME       Char      8     32     $8.       $8.         WIRE TIME
```

As you can see from the output produced by the DATASETS procedure, the
VLIB.WIREDATA view descriptor has six variables: ACCOUNT_TYPE,
SSN_ACCOUNT, WIRE_AMOUNT, WIRE_DATE, WIRE_DESCRIPT, and
WIRE_TIME. The variables are listed in alphabetic order, and the column labeled with
a # (pound sign) in the listing shows the order of each variable as it appears in the
WIRETRAN database segment. You cannot change a view descriptor's variable labels
using the DATASETS procedure. The labels are generated from the IMS item names
when the view descriptor is created.

   For more information about the DATASETS procedure, see *SAS Language Reference:
Concepts* and the *Base SAS Procedures Guide*.
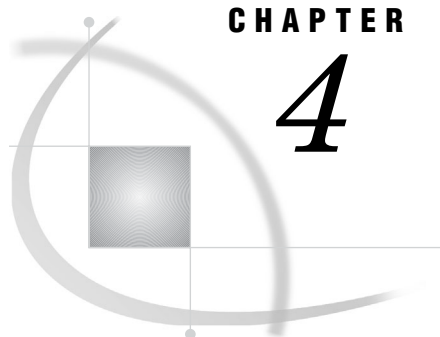
IMS Data in SAS Programs

Introduction to Using IMS Data in SAS Programs

An advantage of the SAS/ACCESS to IMS interface view engine is that it enables SAS to read and write IMS data directly from SAS programs without having to code DL/I calls. This section presents examples that use IMS data that is described by view descriptors as input data for SAS programs. Throughout the examples, the SAS terms *variable* and *observation* are used instead of the IMS terms *field* and *segment* because this section illustrates using SAS procedures and the DATA step. The examples include charting data using the Version 7 SQL procedure to combine data from various sources, and updating a Version 6 SAS data file with data from IMS.

READ, WRITE, ALTER, or PW passwords can be assigned to a view descriptor, access descriptor, PROC SQL view, DATA step view, or SAS data file. See Chapter 6, "ACCESS Procedure Reference," on page 95 and "SAS Passwords for SAS/ACCESS Descriptors" on page 97 for information about assigning passwords.

Appendix 2 includes definitions of all the view descriptors referenced in this section. Appendix 2 also includes the IMS database data, SAS data files, and a DB2 table used in some of the examples.

Charting IMS Data

GCHART procedure programs work with data that is described by view descriptors just as they do with other SAS data sets. The following example creates a horizontal bar chart of the number of checking account withdrawals per day. This example uses

the view descriptor VLIB.CDBTDATE to describe the CHCKDEBT segment of the
ACCTDBD database:

```
options nodate linesize=132;
goptions device=chardrvw;

proc gchart data=vlib.cdbtdate;
   vbar check_date / discrete;
   title2 'Checking Account Withdrawals Per Day';
run;
```

The following display shows the output for this example. CDBTDATE represents the
date of each checking account withdrawal; the number of checking account withdrawals
is represented by the length of the bar. For more information about the GCHART
procedure, see *SAS Language Reference: Concepts* and the *SAS/GRAPH Reference,
Volumes 1 and 2*.

Display 4.1   Results of Charting IMS Data
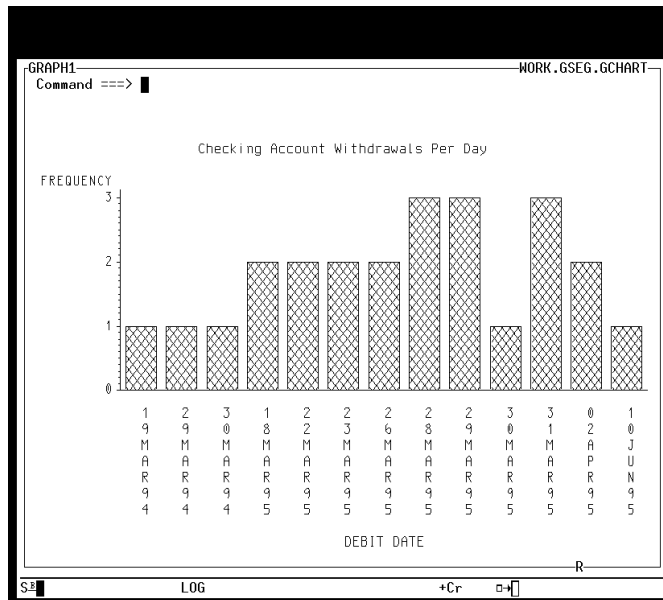
If you have SAS/GRAPH software, you can create colored block charts, plots, and
other graphics based on IMS data. See *SAS/GRAPH Reference, Volumes 1 and 2* for
more information about the kinds of graphics that you can produce with this SAS
product.

Calculating Statistics with IMS Data

You can execute statistical procedures using IMS data that is described by a view
descriptor just as you would using SAS data files. This section shows examples using
the FREQ, MEANS, and RANK procedures.

Calculating Statistics Using the FREQ Procedure

Suppose you want to find the percentages of your accounts in each city where you have a bank so that you can decide where to increase your marketing. The following example calculates the percentages of customers for each city appearing in the IMS database ACCTDBD using the view descriptor VLIB.CUSTINFO:

```
options nodate linesize=80;

proc freq data=vlib.custinfo;
   table city;
   title2 'Cities in the ACCTDBD Database';
run;
```

The following output shows the one-way frequency table that this example generates.

Output 4.1   Results of Calculating Statistics Using the FREQ Procedure

```
                         The SAS System
                   Cities in the ACCTDBD Database


                            CITY

                                       Cumulative  Cumulative
         CITY              Frequency   Percent  Frequency   Percent
         ------------------------------------------------------------
         CHARLOTTESVILLE        2        20.0         2       20.0
         GORDONSVILLE           3        30.0         5       50.0
         ORANGE                 2        20.0         7       70.0
         RAPIDAN                1        10.0         8       80.0
         RICHMOND               2        20.0        10      100.0
```

For more information about the FREQ procedure, see *SAS Language Reference: Concepts* and the *Base SAS Procedures Guide*.

Calculating Statistics Using the MEANS Procedure

In an analysis of recent accounts, suppose that you also want to determine some statistics by customer. In the following example, PROC MEANS is used to generate the mean debit amount for each customer (including the number of observations (N) and the number of missing values (NMISS)):

```
proc sort data=vlib.trans out=mydata.trandata;
   by soc_sec_number;
run;

options nodate linesize=80;

proc means data=mydata.trandata mean
   sum n nmiss maxdec=0;
   by soc_sec_number;
   var check_debit_amount;
   title2 'Mean Debit Amount Per Customer';
run;
```

In the example, the view descriptor VLIB.TRANS selects CUSTOMER, CHCKACCT, and CHCKDEBT segment data from the IMS database ACCTDBD. Since the ACCTDBD database is an HDAM and therefore is not indexed, the data that is described by the view descriptor must be sorted before using PROC MEANS. The sorted data is stored in a SAS data file called MYDATA.TRANDATA, which is then used as input to PROC MEANS.

If your database is indexed, you can use a SAS BY statement for the indexed field so that data is returned as if it is sorted. Database access methods HIDAM, HISAM, and SHISAM are indexed. If your database is not indexed, you need to sort the IMS data before using the MEANS procedure with a BY statement. Because you cannot sort data in an IMS database, you must use the OUT= option to extract data from the database so that you can pass it to the MEANS procedure.

Note: You can store the sorted data in a temporary data set if space is a concern. △

Note: If the view descriptor describes a path of data that includes segments from multiple hierarchical levels, the parent segment information is repeated for each SAS observation. This can cause misleading statistical results. To avoid misleading results, perform mathematical operations using only the data in the segment at the lowest hierarchical level. You can also avoid misleading results by creating a view descriptor that describes only the data in the segment at the lowest hierarchical level. △

The following output shows the output for this example.

Output 4.2 Results of Calculating Statistics Using the MEANS Procedure

```
                            The SAS System
                     Mean Debit Amount Per Customer

------------------------- SOC_SEC_NUMBER=156-45-5672 -------------------------

                          The MEANS Procedure

         Analysis Variable : CHECK_DEBIT_AMOUNT CHECK_DEBIT_AMOUNT

                                                    N
                   Mean            Sum     N      Miss
                 -----------------------------------------
                     27            110     4        0
                 -----------------------------------------


------------------------- SOC_SEC_NUMBER=178-42-6534 -------------------------

         Analysis Variable : CHECK_DEBIT_AMOUNT CHECK_DEBIT_AMOUNT

                                                    N
                   Mean            Sum     N      Miss
                 -----------------------------------------
                     26             26     1        0
                 -----------------------------------------


------------------------- SOC_SEC_NUMBER=234-74-4612 -------------------------

         Analysis Variable : CHECK_DEBIT_AMOUNT CHECK_DEBIT_AMOUNT

                                                    N
                   Mean            Sum     N      Miss
                 -----------------------------------------
                      .              .     0        1
                 -----------------------------------------


------------------------- SOC_SEC_NUMBER=434-62-1224 -------------------------

         Analysis Variable : CHECK_DEBIT_AMOUNT CHECK_DEBIT_AMOUNT

                                                    N
                   Mean            Sum     N      Miss
                 -----------------------------------------
                    162           1620    10        0
                 -----------------------------------------
```

```
                                The SAS System
                        Mean Debit Amount Per Customer

------------------------- SOC_SEC_NUMBER=434-62-1234 -------------------------

                             The MEANS Procedure

            Analysis Variable : CHECK_DEBIT_AMOUNT CHECK_DEBIT_AMOUNT


                                                    N
                    Mean            Sum      N    Miss
                ------------------------------------------
                      .               .      0      1
                ------------------------------------------



------------------------- SOC_SEC_NUMBER=436-42-6394 -------------------------

            Analysis Variable : CHECK_DEBIT_AMOUNT CHECK_DEBIT_AMOUNT


                                                    N
                    Mean            Sum      N    Miss
                ------------------------------------------
                      .               .      0      1
                ------------------------------------------

------------------------- SOC_SEC_NUMBER=456-45-3462 -------------------------

            Analysis Variable : CHECK_DEBIT_AMOUNT CHECK_DEBIT_AMOUNT


                                                    N
                    Mean            Sum      N    Miss
                ------------------------------------------
                      66             263     4      0
                ------------------------------------------



------------------------- SOC_SEC_NUMBER=657-34-3245 -------------------------

            Analysis Variable : CHECK_DEBIT_AMOUNT CHECK_DEBIT_AMOUNT


                                                    N
                    Mean            Sum      N    Miss
                ------------------------------------------
                      .               .      0      1
                ------------------------------------------



------------------------- SOC_SEC_NUMBER=667-73-8275 -------------------------

                             The MEANS Procedure

            Analysis Variable : CHECK_DEBIT_AMOUNT CHECK_DEBIT_AMOUNT


                                                    N
                    Mean            Sum      N    Miss
                ------------------------------------------
                      355            1065    3      2
                ------------------------------------------



------------------------- SOC_SEC_NUMBER=667-82-8275 -------------------------

            Analysis Variable : CHECK_DEBIT_AMOUNT CHECK_DEBIT_AMOUNT


                                                    N
                    Mean            Sum      N    Miss
                ------------------------------------------
                      .               .      0      1
                ------------------------------------------
```

For more information about PROC MEANS, see *SAS Language Reference: Concepts* and the *Base SAS Procedures Guide*.

Calculating Statistics Using the RANK Procedure

You can use advanced statistical procedures on IMS data that is described by a view descriptor just as you would using a SAS data file. The following example uses the RANK procedure to rank checking account deposits by amount. It also assigns the variable name CRDRANK to the new item created by the RANK procedure, extracts and sorts the data, and prints the sorted output data. The view descriptor VLIB.CREDITS describes the CUSTOMER, CHCKACCT, and CHCKCRDT segments in the ACCTDBD database.

```
proc rank data=vlib.credits out=mydata.rankcred;
   var check_credit_amount;
   ranks crdrank;
run;

proc sort data=mydata.rankcred;
   by crdrank;
run;

options nodate linesize=132;

proc print data=mydata.rankcred;
   title2 'Deposits in Ascending Order';
run;
```

The following output shows the result of this example.

Output 4.3   Results of Calculating Statistics using the RANK Procedure

```
                              The SAS System
                         Deposits in Ascending Order


                                  CHECK_    CHECK_   CHECK_       CHECK_
          SOC_SEC_    CHECK_ACCOUNT_   CREDIT_   CREDIT_  CREDIT_      CREDIT_
   OBS     NUMBER         NUMBER       AMOUNT     DATE     TIME         DESC


    1    436-42-6394   345620135872      50.00   02APR95  12:16:34  ACH DEPOSIT
    2    456-45-3462   345620134522      50.00   05APR95  12:14:52  ACH DEPOSIT
    3    156-45-5672   345620123456     100.00   01APR95  12:24:34  ATM DEPOSIT
    4    667-82-8275   382957492811     100.00   16APR95  09:21:14  ACH DEPOSIT
    5    434-62-1224   345620134663     120.00   28MAR95  10:26:45  ACH DEPOSIT
    6    657-34-3245   345620131455     230.00   04APR95  14:24:11  ACH DEPOSIT
    7    434-62-1234   345620104732     400.00   02APR95  10:23:46  ACH DEPOSIT
    8    234-74-4612   345620113263     672.32   31MAR95            ATM DEPOSIT
    9    178-42-6534   745920057114    1300.00   12JUN95  14:34:12  ACH DEPOSIT
   10    434-62-1224   345620134564    1342.42   22MAR95  23:23:52  ACH DEPOSIT
   11    667-73-8275   345620145345    1563.23   31MAR95  15:42:43  MAIN ST BRANCH DEPOSIT
   12    667-73-8275   345620154633    1563.23   31MAR95  15:42:43  BAD ACCT_NUM
```

For more information about PROC RANK and other advanced statistics procedures, see the *Base SAS Procedures Guide*.

Selecting and Combining IMS Data

A great majority of SAS programs select and combine data from various sources. The method you use depends on the configuration of the data. The next three examples show you how to select and combine data using two different methods: the WHERE statement used in a DATA step and the SQL procedure. When choosing between these methods, you should first read the performance considerations discussed in Chapter 7, "Advanced User Topics for the SAS/ACCESS Interface View Engine for IMS," on page 131.

Selecting and Combining Data Using the WHERE Statement

Suppose you have two view descriptors, VLIB.CHKCRD and VLIB.CHKDEB, that contain information about the checking accounts of customers. The view descriptor VLIB.CHKCRD describes the checking credit data in the CUSTOMER, CHCKACCT, and CHCKCRDT segments, and the view descriptor VLIB.CHKDEB describes the checking debit data in the CUSTOMER, CHCKACCT, and CHCKDEBT segments. You could use the SET statement to concatenate the data in these files and create a SAS data file that contains information about checking account transactions by customer. Since you are accessing the same database more than once, you need to reference the same PSB in both view descriptors, but use different PCB index values, where each value references an ACCTDBD PCB that is sensitive to the segments defined in the view. In this example, VLIB.CHKCRD uses a PCB index value of 2, and VLIB.CHKDEB uses a PCB index value of 3 in the ACCUPSB PSB.

The PROC SORT statement orders the accounts by Social Security number and checking account number.

```
data chktrans (keep=soc_sec_number
  check_account_number trantype date amount);
    length trantype $ 6;
    format date date9. amount dollar12.2;
    set vlib.chkcrd(in=crd) vlib.chkdeb(in=dbt);
    where check_balance>0;
    if crd then do;
       trantype='Credit';
       date=check_credit_date;
       amount=check_credit_amount;
    end;
    else if dbt then do;
       trantype='Debit';
       date=check_debit_date;
       amount=check_debit_amount;
    end
run;

proc sort;
    by soc_sec_number check_account_number;
run;

options nodate linesize=80;

proc print data=chktrans;
    by soc_sec_number;
    var check_account_number trantype date amount;
```

```
title2 'Checking Account Transactions by SSN';
run;
```

In the SAS WHERE statement, be sure to use the IMS item name as the search criteria when VALIDVARNAME=V7 and the SAS variable name when VALIDVARNAME=V6. The following output shows the result of the new temporary SAS data file WORK.CHKTRANS.

Output 4.4 Results of Selecting and Combining Data Using the WHERE Statement

```
                     Checking Account Transactions by SSN

--------------------  SOC_SEC_NUMBER=156-45-5672  --------------------

              CHECK_ACCOUNT_
     OBS         NUMBER        TRANTYPE       DATE         AMOUNT

      1        345620123456     Credit     01APR1991       $100.00
      2        345620123456     Debit      28MAR1991        $13.29
      3        345620123456     Debit      31MAR1991        $32.87
      4        345620123456     Debit      02APR1991        $50.00
      5        345620123456     Debit      31MAR1991        $13.42


--------------------  SOC_SEC_NUMBER=178-42-6534  --------------------

              CHECK_ACCOUNT_
     OBS         NUMBER        TRANTYPE       DATE         AMOUNT

      6        745920057114     Credit     12JUN1991     $1,300.00
      7        745920057114     Debit      10JUN1991        $25.89


--------------------  SOC_SEC_NUMBER=234-74-4612  --------------------

              CHECK_ACCOUNT_
     OBS         NUMBER        TRANTYPE       DATE         AMOUNT

      8        345620113263     Credit     31MAR1991       $672.32
      9        345620113263     Debit          .             .


 --------------------  SOC_SEC_NUMBER=434-62-1224  --------------------

              CHECK_ACCOUNT_
     OBS         NUMBER        TRANTYPE       DATE         AMOUNT

     10        345620134564     Credit     22MAR1991     $1,342.42
     11        345620134564     Debit      18MAR1991       $432.87
     12        345620134564     Debit      18MAR1991        $19.23
     13        345620134564     Debit      22MAR1991       $723.23
     14        345620134564     Debit      22MAR1991        $82.32
     15        345620134564     Debit      26MAR1991        $73.62
     16        345620134564     Debit      26MAR1991        $31.23
     17        345620134564     Debit      29MAR1990       $162.87
     18        345620134564     Debit      29MAR1991         $7.12
     19        345620134564     Debit      31MAR1991        $62.34
     20        345620134663     Credit     28MAR1991       $120.00
     21        345620134663     Debit      28MAR1991        $25.00


--------------------  SOC_SEC_NUMBER=434-62-1234  --------------------

              CHECK_ACCOUNT_
     OBS         NUMBER        TRANTYPE       DATE         AMOUNT

     22        345620104732     Credit     02APR1991       $400.00
     23        345620104732     Debit          .             .
```

Checking Account Transactions by SSN

```
---------------------   SOC_SEC_NUMBER=436-42-6394  --------------------

          CHECK_ACCOUNT_
     OBS      NUMBER       TRANTYPE       DATE         AMOUNT

     24    345620135872    Credit      02APR1991       $50.00
     25    345620135872    Debit       30MAR1990          .


---------------------   SOC_SEC_NUMBER=456-45-3462  --------------------

          CHECK_ACCOUNT_
     OBS      NUMBER       TRANTYPE       DATE         AMOUNT

     26    345620134522    Credit      05APR1991       $50.00
     27    345620134522    Debit       29MAR1991       $42.73
     28    345620134522    Debit       29MAR1991      $172.45
     29    345620134522    Debit       30MAR1991       $38.23
     30    345620134522    Debit       02APR1991       $10.00


---------------------   SOC_SEC_NUMBER=657-34-3245  --------------------

          CHECK_ACCOUNT_
     OBS      NUMBER       TRANTYPE       DATE         AMOUNT

     31    345620131455    Credit      04APR1991      $230.00
     32    345620131455    Debit           .             .


---------------------   SOC_SEC_NUMBER=667-73-8275  --------------------

          CHECK_ACCOUNT_
     OBS      NUMBER       TRANTYPE       DATE         AMOUNT

     33    345620145345    Credit      31MAR1991     $1,563.23
     34    345620145345    Debit       19MAR1990          .
     35    345620145345    Debit       23MAR1991      $820.00
     36    345620145345    Debit       23MAR1991       $52.00
     37    345620145345    Debit       28MAR1991      $193.00
     38    345620154633    Credit      31MAR1991     $1,563.23
     39    345620154633    Debit           .             .


---------------------   SOC_SEC_NUMBER=667-82-8275  --------------------

          CHECK_ACCOUNT_
     OBS      NUMBER       TRANTYPE       DATE         AMOUNT

     40    382957492811    Credit      16APR1991      $100.00
     41    382957492811    Debit           .             .
```

The first line of the DATA step uses the KEEP= data set option. This option works with view descriptors just as it works with other SAS data sets. The KEEP= option specifies that you want only the listed variables included in the new SAS data file WORK.CHKTRANS, although you can use the other variables in the view descriptor within the DATA step. Note that the KEEP= option does not reduce the number of variables mapped by the view descriptor and, therefore, does not reduce the amount of data read by the engine.

When you reference a view descriptor in a SAS procedure or DATA step, it is more efficient to use a SAS WHERE statement than a subsetting IF statement because an IF statement does not reduce the amount of data read. A DATA step or SAS procedure passes the SAS WHERE statement to the interface view engine, which attempts to

create SSAs from the WHERE statement. If the engine can create the SSAs, it processes the SAS WHERE statement and returns to SAS only the data that satisfies the WHERE statement. Otherwise, all the data referenced by the view descriptor is returned to SAS for processing. Processing IMS data using a WHERE statement that the IMS engine can turn into SSAs reduces the amount of data read and retrieved by the engine. This improves engine performance significantly. For more information about how IMS handles WHERE statements, see "Performance and Efficient View Descriptors" on page 124.

For more information about the SAS WHERE statement, refer to *SAS Language Reference: Dictionary*.

Selecting and Combining Data Using the SAS SQL Procedure

This section provides two examples of using the SAS SQL procedure on IMS data. The SQL procedure implements the Structured Query Language (SQL) in Version 7 and later of SAS. The SQL procedure is a good way to perform SQL operations with IMS, which by itself has no SQL capabilities. The first example illustrates how to use PROC SQL to combine data from three sources. The second example shows how to use the GROUP BY clause to create new items from data that is described by a view descriptor.

Combining Data from Various Sources

Suppose you have the following items:

- □ a view descriptor, VLIB.CUSTACCT, that is based on the CUSTOMER and CHCKACCT segments of the IMS database ACCTDBD.
- □ a SAS data file, MYDATA.CHGDATA, which contains checking account numbers and checking fees.
- □ a view descriptor, MYDATA.BANKCHRG, that is based on data in a DB2 table that contains additional banking fees. (The MYDATA.BANKCHRG view descriptor has been created using the SAS/ACCESS interface to DB2.)

You can use PROC SQL to create a view that joins all these sources of data. When you use the PROC SQL view in your SAS program, the joined data is presented in a single output table. In this example, using the SAS WHERE or subsetting IF statements would not be an appropriate way of presenting data from various sources because you want to compare variables from several sources rather than simply merge or concatenate the data. For more information about the DB2 table that is used in this example, see Appendix 2, "Example Data," on page 255.

CAUTION:
   When you use PROC SQL to access more than one IMS database, the view descriptors for each database must use the same PSB. In addition, a PCB must be included in that PSB for each database you want to access. If you are accessing the same database multiple times, each view descriptor must specify a different PCB using the PCB index field. △

The following code prints the view descriptors and the SAS data file:

```
options nodate linesize=120;

   proc print data=vlib.custacct;
      title2 'Data Described by VLIB.CUSTACCT';
   run;

   options nodate linesize=80;
```

```
proc print data=mydata.bankchrg;
   title2 'Data Described by MYDATA.BANKCHRG';
run;


proc print data=mydata.chgdata;
   title2 'SAS Data File MYDATA.CHGDATA';
run;
```

The following three outputs show the results of the PRINT procedure performed on the VLIB.CUSTACCT view descriptor (based on IMS data), the MYDATA.BANKCHRG view descriptor (based on DB2 data), and the MYDATA.CHGDATA data file.

Output 4.5   Data That is Described by VLIB.CUSTACCT

```
                         The SAS System
                  Data Described by VLIB.CUSTACCT

          SOC_SEC_                                  CHECK_ACCOUNT_
   OBS     NUMBER       CUSTOMER_NAME               NUMBER

    1    667-73-8275   WALLS, HOOPER J.             345620145345
    2    667-73-8275   WALLS, HOOPER J.             345620154633
    3    434-62-1234   SUMMERS, MARY T.             345620104732
    4    436-42-6394   BOOKER, APRIL M.             345620135872
    5    434-62-1224   SMITH, JAMES MARTIN          345620134564
    6    434-62-1224   SMITH, JAMES MARTIN          345620134663
    7    178-42-6534   PATTILLO, RODRIGUES          745920057114
    8    156-45-5672   O'CONNOR, JOSEPH             345620123456
    9    657-34-3245   BARNHARDT, PAMELA S.         345620131455
   10    667-82-8275   COHEN, ABRAHAM               382957492811
   11    456-45-3462   LITTLE, NANCY M.             345620134522
   12    234-74-4612   WIKOWSKI, JONATHAN S.        345620113263
```

Output 4.6 Data That is Described by MYDATA.BANKCHRG

```
                           The SAS System
                     Data Described by MYDATA.BANKCHRG

      OBS        ssn              accountn     chckchrg    atmfee    loanchrg

       1     667-73-8275       345620145345      3.75       5.00       2.00
       2     434-62-1234       345620104732     15.00      25.00     552.23
       3     436-42-6394       345620135872      1.50       7.50     332.15
       4     434-62-1224       345620134564      9.50       0.00       0.00
       5     178-42-6534                  .      0.50      15.00     223.77
       6     156-45-5672       345620123456      0.00       0.00       0.00
       7     657-34-3245       345620132455     10.25      10.00     100.00
       8     667-82-8275                  .      7.50       7.50     175.75
       9     456-45-3462       345620134522     23.00      30.00     673.23
      10     234-74-4612       345620113262      4.50       7.00       0.00
```

Output 4.7 Data in the SAS Data File MYDATA.CHGDATA

```
                 The SAS System
            SAS Data File MYDATA.CHGDATA


       OBS           account      charge

        1        345620135872       $10
        2        345620134522        $7
        3        345620123456       $12
        4        382957492811        $3
        5        345620134663        $8
        6        345620131455        $6
        7        345620104732        $9
```

The following SAS statements select and combine data from these three sources to
create a PROC SQL view, SQL.CHARGES. The SQL.CHARGES view retrieves checking
fee information so that the bank can charge customers for checking services.

```
options nodate linesize=132;
libname sql 'SAS-data-library';

proc sql;
  create view sql.charges as
    select distinct custacct.soc_sec_number,
           custacct.customer_name,
           custacct.check_account_number,
           chgdata.charge,
           bankchrg.chckchrg,
           bankchrg.atmfee,
           bankchrg.loanchrg
    from vlib.custacct,
         mydata.bankchrg,
         mydata.chgdata
    where custacct.soc_sec_number=bankchrg.ssn and
      custacct.check_account_number=chgdata.account;
title2 'Banking Charges for the Month';
```

```
select * from sql.charges;
```

The CREATE statement incorporates a WHERE clause along with the SELECT clause. The last SELECT statement retrieves and displays the PROC SQL view SQL.CHARGES. To select all the items from the view, use an asterisk (*) in place of item names. When an asterisk is used, the order of the items displayed matches the order of the items as specified in the SQL.CHARGES view definition. Notice that PROC SQL prints the output automatically on the display using the IMS item names instead of the SAS variable names. It also executes without a RUN statement when the procedure is submitted. The following output shows the data that is described by the PROC SQL view SQL.CHARGES.

Output 4.8   Results of Combining Data From Various Sources

```
                          The SAS System
                    Banking Charges for the Month


    SOC_SEC_                           CHECK_ACCOUNT_
      NUMBER       CUSTOMER_NAME              NUMBER   charge  chckchrg  atmfee  loanchrg
  --------------------------------------------------------------------------------------
    156-45-5672   O'CONNOR, JOSEPH        345620123456    $12     0.00    0.00      0.00
    434-62-1224   SMITH, JAMES MARTIN     345620134663     $8     9.50    0.00      0.00
    434-62-1234   SUMMERS, MARY T.        345620104732     $9    15.00   25.00    552.23
    436-42-6394   BOOKER, APRIL M.        345620135872    $10     1.50    7.50    332.15
    456-45-3462   LITTLE, NANCY M.        345620134522     $7    23.00   30.00    673.23
    657-34-3245   BARNHARDT, PAMELA S.    345620131455     $6    10.25   10.00    100.00
    667-82-8275   COHEN, ABRAHAM          382957492811     $3     7.50    7.50    175.75
```

Creating New Items with the GROUP BY Clause

It is often useful to create new items with summary or aggregate functions such as the SUM function. Although you cannot use the ACCESS procedure to create new items, you can easily use the SQL procedure with data that is described by a view descriptor to display output that contains new items.

This example uses PROC SQL to retrieve and manipulate data from the view descriptor VLIB.SAVEBAL, which is based on the CUSTOMER and SAVEACCT segments in the ACCTDBD database. When this query (as a SELECT statement is often called) is submitted, it calculates and displays the average savings account balance for each city.

```
options nodate linesize=80;

proc sql;
   title2 'Average Savings Balance Per City';
   select distinct city,
       avg(savings_balance) label='Average Balance'
           format=dollar12.2
   from vlib.savebal
   where city is not missing
   group by city;
```

The following output shows the query's result.

Output 4.9   Results of Creating New Items with the GROUP BY Clause

```
                   The SAS System
              Average Savings Balance Per City

                                        Average
          CITY                          Balance
          ----------------------------------------
          CHARLOTTESVILLE               $1,673.35
          GORDONSVILLE                  $4,758.26
          ORANGE                          $615.60
          RAPIDAN                         $672.63
          RICHMOND                        $924.62
```

For more information about the SQL procedure, refer to the *SAS SQL Procedure User's Guide*.

Updating a SAS Data File with IMS Data

You can update a SAS data file with IMS data that is described by a view descriptor just as you can update a SAS data file using another SAS data file: by using a DATA step UPDATE statement. In this section, the term *transaction data* refers to the new data that is to be added to the original file.

You can even perform updates when the file to be updated is a Version 6 data file with user-defined, 8-byte SAS variable names and the transaction data is from Version 7 and later data sets containing generated variable names of up to 32 bytes.

You have two choices when you update a Version 6 SAS data file with data from later releases:

□ operate the current release in default mode. Your Version 6 program will run, but WHERE processing will not be available.

□ set the VALIDVARNAME system option to V6 to operate in Version 6 mode. The V6 option offers functionality comparable to Version 6 of the interface view engine, including WHERE processing. The VALIDVARNAME system option controls what type of variable names will be used in the SAS session by converting any nonconforming names to the specified format. For more information about the VALIDVARNAME system, see *SAS Language Reference: Dictionary*.

The following examples illustrate the situations in which each of the options is appropriate.

Example of VALIDVARNAME=V6

Suppose you have a Version 6 SAS data set, VER6.SSNUMS, which contains some customer names and Social Security numbers. You want to update this data set with data that is described by VLIB.SSNAME, a view descriptor based on the CUSTOMER segment of the IMS database ACCTDBD. Since this will require you to first sort the data then create an output data set with the sorted data, this is a good situation for using VALIDVARNAME=V6.

To perform the update, you would enter the following SAS statements:

```
options validvarname=V6;
options nodate linesize=80;
```

```
libname ver6 'SAS-data-library';

proc sort data=ver6.ssnums;
   by ssnumb;
run;

proc print data=ver6.ssnums;
   title2 'VER6.SSNUMS Data File';
run;

proc sort data=vlib.ssname out=mydata.newnums;
   by ssnumb;
run;

proc print data=mydata.newnums;
   title2 'Data Described by MYDATA.NEWNUMS';
run;

data mydata.newnames;
   update ver6.ssnums mydata.newnums;
   by ssnumb;
run;

proc print data=mydata.newnames;
   title2 'MYDATA.NEWNAMES Data File';
run;
```

The new SAS data file MYDATA.NEWNAMES is a Version 6 data file stored in a Version 6 data library associated with the libref MYDATA.

The following three outputs show the results of PRINT procedures for the original data file, the transaction data, and the updated data file.

Output 4.10   Data in the Data File to Be Updated, VER6.SSNUMS

```
                The SAS System
              VER6.SSNUMS Data File

    OBS       SSNUMB        NAME

     1      267-83-2241    GORDIEVSKY, OLEG
     2      276-44-6885    MIFUNE, YUKIO
     3      352-44-2151    SHIEKELESLAM, SHALA
     4      436-46-1931    NISHIMATSU-LYNCH, CAROL
```

Output 4.11   Data That is Described by Updated Data File MYDATA.NEWNUMS

```
                          The SAS System
                  Data Described by MYDATA.NEWNUMS

      OBS   SSNUMB                NAME

        1   156-45-5672           O'CONNOR, JOSEPH
        2   178-42-6534           PATTILLO, RODRIGUES
        3   234-74-4612           WIKOWSKI, JONATHAN S.
        4   434-62-1224           SMITH, JAMES MARTIN
        5   434-62-1234           SUMMERS, MARY T.
        6   436-42-6394           BOOKER, APRIL M.
        7   456-45-3462           LITTLE, NANCY M.
        8   657-34-3245           BARNHARDT, PAMELA S.
        9   667-73-8275           WALLS, HOOPER J.
       10   667-82-8275           COHEN, ABRAHAM
```

Output 4.12   Results of Updating a Version 6 SAS Data File with IMS Data

```
                          The SAS System
                   MYDATA.NEWNAMES Data File


      OBS     SSNUMB        NAME

        1     156-45-5672   O'CONNOR, JOSEPH
        2     178-42-6534   PATTILLO, RODRIGUES
        3     234-74-4612   WIKOWSKI, JONATHAN S.
        4     267-83-2241   GORDIEVSKY, OLEG
        5     276-44-6885   MIFUNE, YUKIO
        6     352-44-2151   SHIEKELESLAM, SHALA
        7     434-62-1224   SMITH, JAMES MARTIN
        8     434-62-1234   SUMMERS, MARY T.
        9     436-42-6394   BOOKER, APRIL M.
       10     436-46-1931   NISHIMATSU-LYNCH, CAROL
       11     456-45-3462   LITTLE, NANCY M.
       12     657-34-3245   BARNHARDT, PAMELA S.
       13     667-73-8275   WALLS, HOOPER J.
       14     667-82-8275   COHEN, ABRAHAM
```

For more information about the UPDATE statement, see *SAS Language Reference: Dictionary*.

Example of VALIDVARNAME=V7

The following is an example of a Version 7 or later update of data. The Version 7 or later data set, MYDATA.SSNUMS, is updated with data that is described by the view descriptor VLIB.SSNAME. Both the data in the data set and in the view descriptor are sorted by social security number before the output data set is used to update the existing data set.

To perform the update, you would enter the following statements:

```
proc sort data=mydata.ssnums;
   by soc_sec_number;
run;
```

```
proc print data=mydata.ssnums;
   title2 'MYDATA.SSNUMS Data Set';
run;

proc sort data=vlib.ssname out=mydata.newnums;
   by soc_sec_number;
run;

proc print data=mydata.newnums;
   title2 'Data Described by MYDATA.NEWNUMS';
run;

data mydata.newnames;
   update mydata.ssnums mydata.newnums;
   by soc_sec_number;
   run;

proc print data=mydata.newnames;
   title2 'MYDATA.NEWNAMES Data Set';
   run;
```

The new SAS data file MYDATA.NEWNAMES is a Version 7 or later data file that is stored in a Version 7 or later data library associated with the libref MYDATA. The following three outputs show the results of the PRINT procedures for the original data file, the transaction data, and the updated data file.

Output 4.13 Data in the Data File to be Updated, MYDATA.SSNUMS

```
                The SAS System
           MYDATA.SSNUMS Data Set


           soc_sec_
   OBS     number       customer_name

   1     267-83-2241    GORDIEVSKY, OLEG
   2     276-44-6885    MIFUNE, YUKIO
   3     352-44-2151    SHIEKELESLAM, SHALA
   4     436-46-1931    NISHIMATSU-LYNCH, CAROL
```

Output 4.14 Data That is Described by Updated Data File MYDATA.NEWNUMS

```
                The SAS System
          Data Described by MYDATA.NEWNUMS


           SOC_SEC_
    OBS     NUMBER        CUSTOMER_NAME

    1     156-45-5672    O'CONNOR, JOSEPH
    2     178-42-6534    PATTILLO, RODRIGUES
    3     234-74-4612    WIKOWSKI, JONATHAN S.
    4     434-62-1224    SMITH, JAMES MARTIN
    5     434-62-1234    SUMMERS, MARY T.
    6     436-42-6394    BOOKER, APRIL M.
    7     456-45-3462    LITTLE, NANCY M.
    8     657-34-3245    BARNHARDT, PAMELA S.
    9     667-73-8275    WALLS, HOOPER J.
   10     667-82-8275    COHEN, ABRAHAM
```

Output 4.15 Results of Updating a Version 7 or Later SAS Data File with IMS Data

```
                    The SAS System
                MYDATA.NEWNAMES Data Set

             soc_sec_
     OBS      number         customer_name

      1    156-45-5672    O'CONNOR, JOSEPH
      2    178-42-6534    PATTILLO, RODRIGUES
      3    234-74-4612    WIKOWSKI, JONATHAN S.
      4    267-83-2241    GORDIEVSKY, OLEG
      5    276-44-6885    MIFUNE, YUKIO
      6    352-44-2151    SHIEKELESLAM, SHALA
      7    434-62-1224    SMITH, JAMES MARTIN
      8    434-62-1234    SUMMERS, MARY T.
      9    436-42-6394    BOOKER, APRIL M.
     10    436-46-1931    NISHIMATSU-LYNCH, CAROL
     11    456-45-3462    LITTLE, NANCY M.
     12    657-34-3245    BARNHARDT, PAMELA S.
     13    667-73-8275    WALLS, HOOPER J.
     14    667-82-8275    COHEN, ABRAHAM
```

C H A P T E R

5

Browsing and Updating IMS Data

Introduction to Browsing and Updating IMS Data

The SAS/ACCESS interface to IMS enables you to browse and update your IMS data directly from a SAS session or program. This section shows you how to use SAS procedures to review and update IMS data that is described by SAS/ACCESS view descriptors. The examples in this section use the view descriptors VLIB.CUSTINFO and VLIB.CHCKACCT. See Appendix 2, "Example Data," on page 255 for definitions of all the view descriptors referenced in this section. Appendix 2, "Example Data," on page 255 also includes the IMS database and SAS data files and data sets.

To browse or update IMS data, you must use a Program Specification Block (PSB) that contains a Program Communication Block (PCB) with the level of access desired. You need to have this desired level of access to the database, to the segments in that database, and to the fields in those segments. The types of access that a PCB can allow include the following:

G   get

I   insert

R   replace

D   delete

A   all.

Refer to Chapter 2, "IMS Essentials," on page 11 and "Program Specification Block" on page 24 for more information about accessing IMS data.

READ, WRITE, ALTER, or PW passwords can be assigned to a view descriptor, access descriptor, PROC SQL view, DATA step view, or SAS data file. See Chapter 6, "ACCESS Procedure Reference," on page 95 and "SAS Passwords for SAS/ACCESS Descriptors" on page 97 for more information about assigning passwords.

Browsing and Updating IMS Data with SAS/FSP Procedures

If your site has SAS/FSP software as well as SAS/ACCESS software, you can browse and update IMS data that is described by a view descriptor from within a SAS/FSP procedure.

You can use any of three SAS/FSP procedures: FSBROWSE, FSEDIT, and FSVIEW. The FSBROWSE and FSEDIT procedures display one observation at a time, while the FSVIEW procedure produces multiple observations in a tabular format, similar to the PRINT procedure. PROC FSVIEW enables you both to browse and update IMS data, depending on which option you choose. The FSBROWSE, FSEDIT, or FSVIEW procedures can be used only with data that is accessed by a view descriptor, a PROC SQL view, a DATA step view, or a SAS data file. You cannot reference an access descriptor with any SAS procedure or in the SAS DATA step.

Note: The formats assigned by the ACCESS procedure are by default used as informats by the SAS/FSP procedures when you add or update a path of data. △

Browsing Data Using the FSBROWSE Procedure

The FSBROWSE procedure enables you to look at IMS data but does not allow you to change it. For example, the following SAS statements enable you to browse one record of VLIB.CUSTINFO at a time:

```
proc fsbrowse data=vlib.custinfo;
run;
```

The following display shows the last observation of the data that is described by the VLIB.CUSTINFO view descriptor. To browse each observation, issue the FORWARD and BACKWARD commands. Because a view descriptor can describe only one path of data in an IMS database, you can browse observations in only one path of data.

Display 5.1   Browsing IMS Data in the FSBROWSE Window

```
┌─────────────────────── SAS: FSBROWSE VLIB.CUSTINFO—Obs 10 ───────────┬─┬─┐
│ Command ===> █                                                        │ │ │
│                                                                        │
│                                                                        │
│              soc_sec_number: 234-74-4612                               │
│                                                                        │
│              customer_name:  WIKOWSKI, JONATHAN S                      │
│                                                                        │
│              addr_line_1:    4356 CAMPUS DRIVE                         │
│                                                                        │
│              city:           RICHMOND                                  │
│                                                                        │
│              state:          VA                                        │
│                                                                        │
│              country:        USA                                       │
│                                                                        │
│              zip_code:       26502-5317                                │
│                                                                        │
│              home_phone:     803-467-4587                              │
│                                                                        │
│              office_phone:   803-654-7238                              │
│                                                                        │
└────────────────────────────────────────────────────────────────────────┘
```

For more information about the FSBROWSE procedure, see "The FSBROWSE Procedure" in *SAS/FSP Procedures Guide*.

Note: Accessing observations by observation number is not supported for IMS view descriptors within the FSBROWSE procedure, but a WHERE command can be used to view a subset of the data. △

Updating Data Using the FSEDIT Procedure

The FSEDIT procedure enables you to update the IMS data that is described by a view descriptor if the view descriptor specifies in your PSB a PCB that provides you with the appropriate level of update access (insert, replace, delete, or all) for the database segments. For example, if the area codes used in HOME_PHONE and OFFICE_PHONE are incorrect for Richmond, you can correct them with the FSEDIT procedure.

For example, the following statements enable you to edit one record of VLIN.CUSTINFO at a time:

```
proc fsedit data=vlib.custinfo;
run;
```

An FSEDIT window appears that looks like the FSBROWSE window. Scroll to the observation you want, or enter a WHERE statement to display the correct observation. You can then add or further update the information about customer , as shown in the following display.

Display 5.2   Updating IMS Data in the FSEDIT Window

```
┌─────────────────────── SAS: FSEDIT VLIB.CUSTINFO—Obs 10 ──────────────── ▫ □
│ Command ===> █
│
│
│                    soc_sec_number: 234-74-4612
│
│                    customer_name:  WIKOWSKI, JONATHAN S
│
│                    addr_line_1:    4356 CAMPUS DRIVE
│
│                    city:           RICHMOND
│
│                    state:          VA
│
│                    country:        USA
│
│                    zip_code:       26502-5317
│
│                    home_phone:     803-467-4587
│
│                    office_phone:   803-654-7238
│
│
│
└───────────────────────────────────────────────────────────────────────────
```

For more information about the FSEDIT procedure, see "The FSEDIT Procedure" in *SAS/FSP Procedures Guide*.

Browsing and Updating Data Using the FSVIEW Procedure

The FSVIEW procedure enables you to browse or update IMS data that is described by a view descriptor, depending on how you invoke the procedure.

Browsing Data Using the FSVIEW Procedure

For example, to browse IMS data in a tabular format, you could submit the following PROC FSVIEW statements in the Program Editor:

```
proc fsview data=vlib.custinfo;
run;
```

Browse mode is the default for the FSVIEW procedure. The statements produce the window shown in the following display.

Display 5.3   Browsing IMS Data in the FSVIEW Window

```
┌───────────────────────── SAS: FSVIEW:  VLIB.CUSTINFO (B) ──────────────── ▫ □
│ Command ===> █
│
│ Obs    soc_sec_number  customer_name       addr_line_1        city
│
│   1    667-73-8275     WALLS, HOOPER J.     4525 CLARENDON RD  RAPIDAN          △
│   2    434-62-1234     SUMMERS, MARY T.     4322 LEON ST.      GORDONSVILLE
│   3    436-42-6394     BOOKER, APRIL M.     9712 WALLINGFORD PL. GORDONSVILLE
│   4    434-62-1224     SMITH, JAMES MARTIN  133 TOWNSEND ST.   GORDONSVILLE
│   5    178-42-6534     PATTILLO, RODRIGUES  9712 COOK RD.      ORANGE
│   6    156-45-5672     O'CONNOR, JOSEPH     235 MAIN ST.       ORANGE
│   7    657-34-3245     BARNHARDT, PAMELA S. RT 2 BOX 324       CHARLOTTESVILLE
│   8    667-82-8275     COHEN, ABRAHAM       45 DUKE ST.        CHARLOTTESVILLE
│   9    456-45-3462     LITTLE, NANCY M.     4543 ELGIN AVE.    RICHMOND
│  10    234-74-4612     WIKOWSKI, JONATHAN S 4356 CAMPUS DRIVE  RICHMOND
│
│
│
│
│
│
│
│
│                                                                                ▽
│      ◁                                                            ▷
└───────────────────────────────────────────────────────────────────────────
```

Updating Data Using the FSVIEW Procedure

To edit IMS data in a tabular format, you must add the EDIT or MODIFY option to the PROC FSVIEW statement, as shown here:

```
proc fsview data=vlib.custinfo edit;
run;
```

Note: The CANCEL command in the FSVIEW window does *not* cancel your changes; it ends the browse or edit session. △

Specifying a SAS WHERE Statement While Browsing or Updating Data

If the IMS engine can generate SSAs from the WHERE statement, it then retrieves a subset of the IMS data. If the engine cannot generate SSAs from the WHERE statement, the WHERE statement is passed to SAS for processing. You can also use a SAS WHERE command to retrieve a subset of IMS data after you have invoked one of the SAS/FSP procedures using the PROC statements.

If you use a SAS WHERE *statement,* only the observations specified by that SAS WHERE statement are available. The other observations are not available until you exit the procedure. This is called a *permanent WHERE clause.*

If you use the SAS WHERE *command,* you can clear the command to make all the observations available. This is called a *temporary WHERE clause.*

In the following example, the FSEDIT procedure uses a SAS WHERE statement to retrieve a subset of customers from Richmond.

```
proc fsedit data=vlib.custinfo;
   where city='RICHMOND';
run;
```

The following display shows the FSEDIT window after the statements have been submitted.

Display 5.4   Submitting a SAS WHERE Statement While Invoking PROC FSEDIT
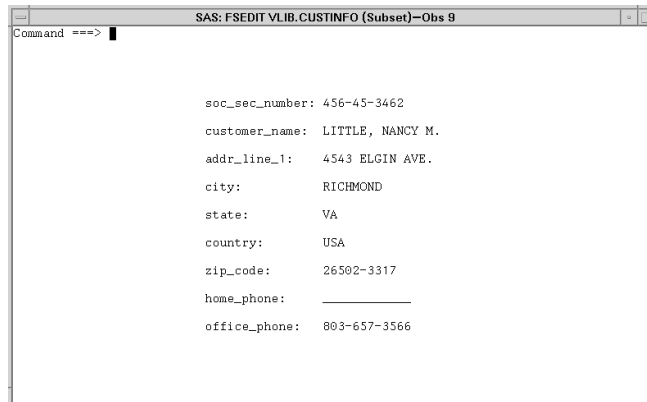
```
┌─────────────────────────────────────────────────────────────────┐
│ ─              SAS: FSEDIT VLIB.CUSTINFO (Subset)—Obs 9      ∘ □  │
│ Command ===> █                                                    │
│                                                                   │
│                                                                   │
│               soc_sec_number: 456-45-3462                         │
│                                                                   │
│               customer_name:  LITTLE, NANCY M.                    │
│                                                                   │
│               addr_line_1:    4543 ELGIN AVE.                     │
│                                                                   │
│               city:           RICHMOND                            │
│                                                                   │
│               state:          VA                                  │
│                                                                   │
│               country:        USA                                 │
│                                                                   │
│               zip_code:       26502-3317                          │
│                                                                   │
│               home_phone:     _____                         │
│                                                                   │
│               office_phone:   803-657-3566                        │
│                                                                   │
│                                                                   │
│                                                                   │
└─────────────────────────────────────────────────────────────────┘
```

Only the two observations with a CITY value of **RICHMOND** are retrieved for editing; you must scroll forward to see the second observation. The word **(Subset)** appears after VLIB.CUSTINFO in the window title to remind you that the retrieved data is a subset of the data that is described by the view descriptor. You can then edit each observation by typing over any incorrect information. Issue the END command to end your editing session. If you want to cancel changes to an observation, you can issue the CANCEL command before you scroll to another observation. Once you scroll, the changes are saved.

You can also enter a SAS WHERE command to display a subset of your data. A SAS WHERE command is a SAS WHERE expression that you enter on the command line. For example, to begin the FSEDIT procedure, you can submit the following statements in the Program Editor:

```
proc fsedit data=vlib.custinfo;
run;
```

The following display shows what the FSEDIT display looks like when the following command-line command is entered within the FSEDIT window:

```
where city='RICHMOND'
```

Display 5.5   Entering a SAS WHERE Command in an FSEDIT Window


```
┌─────────────────────────────────────────────────────────────────────┐
│  —                      SAS: FSEDIT VLIB.CUSTINFO—Where ... Obs 9          ∘ □ │
│ Command ===> ▮                                                          │
│                                                                        │
│                                                                        │
│                   soc_sec_number: 456-45-3462                          │
│                                                                        │
│                   customer_name:  LITTLE, NANCY M.                     │
│                                                                        │
│                   addr_line_1:    4543 ELGIN AVE.                      │
│                                                                        │
│                   city:           RICHMOND                             │
│                                                                        │
│                   state:          VA                                   │
│                                                                        │
│                   country:        USA                                  │
│                                                                        │
│                   zip_code:       26502-3317                           │
│                                                                        │
│                   home_phone:     _____                          │
│                                                                        │
│                   office_phone:   803-657-3566                         │
│                                                                        │
└─────────────────────────────────────────────────────────────────────┘
```
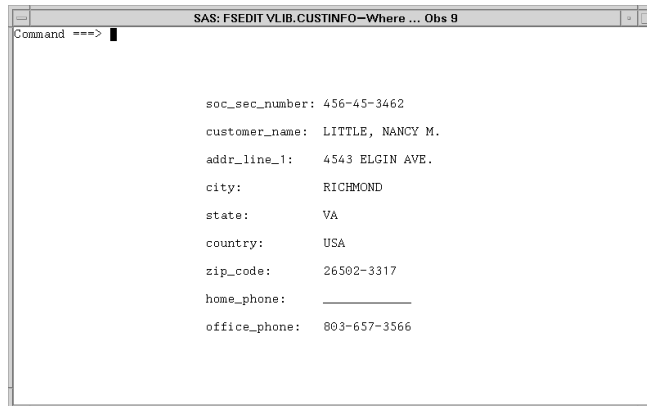
Only the two observations with a CITY value of **RICHMOND** are retrieved for editing; you must scroll forward to see the second observation. You can then edit each observation, as described earlier.

Although these examples have shown how to use a SAS WHERE statement and command with the FSEDIT procedure, you can use a SAS WHERE statement and command in the same way with the FSBROWSE and FSVIEW procedures. For more information about the SAS WHERE statement, refer to *SAS Language Reference: Dictionary*. For more information about the SAS WHERE command within the SAS/FSP procedures, refer to *SAS/FSP Procedures Guide*.

Scrolling with SAS/FSP Procedures

Scrolling through data using the FSEDIT, FSBROWSE, or FSVIEW procedures is different when you are using view descriptors instead of SAS data files. While the FORWARD command works identically in both cases, the BACKWARD command does not.

Scrolling backward with SAS/FSP procedures can be slow when you are working with a large database, particularly when you are looking at a path of data in a record near the end of the database. To scroll backward through an IMS database, the IMS engine must read forward in the database from the beginning until it reaches the observation preceding the one that is displayed when the BACKWARD command was issued. For example, suppose the view defines 5,000 observations, and the current observation is 3,400. To scroll backward to observation 3,399, the FSEDIT procedure must sequentially read observations 1 through 3,398. This can be expensive and time consuming.

Inserting and Deleting Segments with SAS/FSP Procedures

Inserting and deleting database segments with SAS/FSP procedures is also different when you are using view descriptors rather than SAS data files.

You can use the FSEDIT and FSVIEW procedures to insert segments into one path of an IMS database on which a view descriptor is based, assuming you are using a PCB that enables you insert access to the database segments. There are two ways to add a new segment to an IMS database using SAS/FSP procedures:

□ To insert one path of data, type **ADD** on the command line and press ENTER. You can then enter an entire path of data, which the IMS engine inserts in the database using a path call.

□ To insert a path of data under an existing parent segment, use a WHERE statement or scroll to the parent segment under which you want to insert the path of data. If there are no child segments under the parent segment, enter the path of data and press ENTER. The IMS engine inserts the new segments under the existing parent segment. If child segments do exist, display one of the paths of data and type the new data over the old path of data, making sure that you change the key field value in the segments to be inserted. The IMS engine then inserts the new segment.

If the view descriptor you are using does not include all the variables defined in the access descriptor for the segment to be inserted, low values (hexadecimal zeroes) are placed in those fields in the new segment occurrence inserted into the database. For more information about inserting segments when the SAS observations contain missing values, see "Handling Missing Values" on page 136 in Chapter 7, "Advanced User Topics for the SAS/ACCESS Interface View Engine for IMS," on page 131. Refer to *SAS/FSP Procedures Guide* for more information about how to use the ADD and DUP commands in the FSEDIT procedure and the AUTOADD and DUP commands in the FSVIEW procedure.

When the DELETE command is used while the FSEDIT or FSVIEW procedure is referencing a view descriptor, the lowest-level existing database segment referenced in the view descriptor is removed permanently from the IMS database. Refer to *SAS/FSP Procedures Guide* for more information about this command.

CAUTION:

If you delete segments using a view descriptor that references only the upper hierarchical level segments in the database, any children of these segments will also be deleted, even though those child segments are not included in the view descriptor. △

For example, consider a database consisting of a root segment, a child segment under the root, and another child segment under that child. If you delete a segment in that database using a view descriptor that references only the root and one child, the DELETE command will delete the entire path of data below the root segment. There are two ways you can delete an entire database record:

□ Use the DELETE command with a view descriptor that references the root segment only.

□ Use the DELETE command multiple times with a view descriptor that references an entire path of data in the database. Each time you use the DELETE command, only the lowest existing segment in the path is deleted.

See "Delete Processing" on page 148 in Chapter 7, "Advanced User Topics for the SAS/ACCESS Interface View Engine for IMS," on page 131 for more information about deleting segments.

The following example illustrates how to use the DELETE command in the FSEDIT procedure. Suppose you want to edit the IMS data that is described by

VLIB.CUSTINFO to eliminate customers who have closed their bank accounts. If you are using a PCB that provides you with delete authority, you can perform this function by using the FSEDIT procedure from the ACCESS window or with a PROC FSEDIT statement. Scroll forward to the observations to be deleted and enter **DELETE** on the command line, as shown in the following display.

Display 5.6   Deleting an IMS Segment in an FSEDIT Window

```
┌──                      SAS: FSEDIT VLIB.CUSTINFO─Where ... Obs 10              ▫ □ ┐
│Command ===> delete█                                                              │
│                                                                                  │
│                                                                                  │
│                          soc_sec_number: 234-74-4612                             │
│                                                                                  │
│                          customer_name:  WIKOWSKI, JONATHAN S                    │
│                                                                                  │
│                          addr_line_1:    4356 CAMPUS DRIVE                       │
│                                                                                  │
│                          city:           RICHMOND                               │
│                                                                                  │
│                          state:          VA                                      │
│                                                                                  │
│                          country:        USA                                     │
│                                                                                  │
│                          zip_code:       26502-5317                             │
│                                                                                  │
│                          home_phone:     803-467-4587                           │
│                                                                                  │
│                          office_phone:   803-654-7238                           │
│                                                                                  │
│                                                                                  │
│                                                                                  │
└──────────────────────────────────────────────────────────────────────────────────┘
```

The DELETE command deletes this root segment from the IMS database that is described by VLIB.CUSTINFO and any child segments under it, and displays a message to that effect, as shown in the following display.

Display 5.7   Using the DELETE Command in the FSEDIT Window

```
┌──                      SAS: FSEDIT VLIB.CUSTINFO─Where ... DELETED 10          ▫ □ ┐
│Command ===> █                                                                     │
│NOTE: Observation has been deleted.                                               │
│                                                                                  │
│                          soc_sec_number: _____                              │
│                                                                                  │
│                          customer_name:  _____                     │
│                                                                                  │
│                          addr_line_1:    _____                     │
│                                                                                  │
│                          city:           _____                           │
│                                                                                  │
│                          state:          __                                      │
│                                                                                  │
│                          country:        ___                                     │
│                                                                                  │
│                          zip_code:       _____                               │
│                                                                                  │
│                          home_phone:     _____                              │
│                                                                                  │
│                          office_phone:   _____                              │
│                                                                                  │
│                                                                                  │
│                                                                                  │
└──────────────────────────────────────────────────────────────────────────────────┘
```

For more information about using SAS/FSP procedures, see *SAS/FSP Procedures Guide*.

Browsing and Updating IMS Data with the SQL Procedure

The SQL procedure enables you to retrieve and update data from IMS databases. You can retrieve and browse IMS data by specifying a view descriptor in the SQL procedure's SELECT statement.

To update the data, you can specify view descriptors in the SQL procedure's INSERT, DELETE, and UPDATE statements. The specified view descriptor can access data from

only one IMS database path. You must use a PCB that provides you with the appropriate level of access (insert, replace, delete, or all) for the segments that you want to update before you can edit the IMS data.

The following list summarizes these SQL procedure statements:

SELECT   retrieves, manipulates, and displays data from IMS databases. A SELECT statement is usually referred to as a query because it queries the database for information.

DELETE   deletes segments from an IMS database.

INSERT   inserts segments in an IMS database.

UPDATE   updates the data values in an IMS database.

If you want to use the SQL procedure to join or access more than one IMS database, you must use a PSB in your view descriptors that includes a PCB for each database to be accessed. Each view descriptor to be joined must use the same PSB. If you join two view descriptors that reference different paths in the same database, each view descriptor must reference in the PSB (that refers to the same database) a different PCB by using the **PCB Index** field. That is, to access the same database using different view descriptors in any SAS procedure, you must include multiple PCBs for that database.

When using PROC SQL, notice that the data is displayed in the SAS Output window in the SAS windowing environment and written to the SASLIST DDname in batch mode, interactive line mode, and noninteractive mode. This procedure displays output data automatically without the PRINT procedure and executes without a RUN statement when an SQL procedure statement is submitted.

Retrieving and Updating Data with the SQL Procedure

Note: The following PROC SQL examples assume the ACCTDBD database has not been updated by the earlier SAS/FSP examples. △

You can use the SELECT statement to browse IMS data that is described by a view descriptor. The query in the following example retrieves all the observations in the IMS ACCTDBD database that are described by the VLIB.CUSTINFO view descriptor.

```
options linesize=132;

proc sql;
title2 'IMS Data Retrieved by a PROC SQL query';
select * /* An asterisk means select all variables */
   from vlib.custinfo;
```

The OPTIONS statement is used to reset the default output width to 132 columns. The following output displays the query's output. Note that PROC SQL displays labels, which are the IMS item names. In Version 7 and later, the item names are also the SAS variable names, as shown here.

Output 5.1   Results of Retrieving IMS Data with a PROC SQL Query

```
                                       The SAS System
                           IMS Data Retrieved by a PROC SQL query


    SOC_SEC_
    NUMBER        CUSTOMER_NAME                         ADDR_LINE_1            ADDR_LINE_2
    CITY                         STATE  COUNTRY         ZIP_CODE   HOME_PHONE  OFFICE_PHONE
    ---------------------------------------------------------------------------------------------------
    667-73-8275   WALLS, HOOPER J.                                            4525 CLARENDON RD
    RAPIDAN                      VA     USA             22215-5600 803-657-3098 803-645-4418

    434-62-1234   SUMMERS, MARY T.                                           4322 LEON ST.
    GORDONSVILLE                 VA     USA             26001-0670 803-657-1687

    436-42-6394   BOOKER, APRIL M.                                          9712 WALLINGFORD PL.
    GORDONSVILLE                 VA     USA             26001-0670 803-657-1346

    434-62-1224   SMITH, JAMES MARTIN                                       133 TOWNSEND ST.
    GORDONSVILLE                 VA     USA             26001-0670 803-657-3437

    178-42-6534   PATTILLO, RODRIGUES                                       9712 COOK RD.
    ORANGE                       VA     USA             26042-1650 803-657-1346 803-657-1345

    156-45-5672   O'CONNOR, JOSEPH                                          235 MAIN ST.
    ORANGE                       VA     USA             26042-1650 803-657-5656 803-623-4257

    657-34-3245   BARNHARDT, PAMELA S.                                      RT 2 BOX 324
    CHARLOTTESVILLE              VA     USA             25804-0997 803-345-4346 803-355-2543

    667-82-8275   COHEN, ABRAHAM                                            2345 DUKE ST.
    CHARLOTTESVILLE              VA     USA             25804-0997 803-657-7435 803-645-4234

    456-45-3462   LITTLE, NANCY M.                                          4543 ELGIN AVE.
    RICHMOND                     VA     USA             26502-3317 803-657-3566

    234-74-4612   WIKOWSKI, JONATHAN S.                                     4356 CAMPUS DRIVE
    RICHMOND                     VA     USA             26502-5317 803-467-4587 803-654-7238
```

You can specify a WHERE clause as part of the SQL procedure's SELECT statement to retrieve a subset of the database data. The following example displays a list of customers who have accounts with the Richmond branch of the bank:

```
title2 'IMS Data Retrieved by a WHERE Statement';
   select *
      from vlib.custinfo
      where city='RICHMOND';
```

Notice that the PROC SQL statement is not repeated in this query. With the SQL procedure, you do not need to repeat the PROC SQL statement unless you submit another SAS procedure, a DATA step, or a QUIT statement between PROC SQL statements. The following output displays the customers of the Richmond branch who are described by VLIB.CUSTINFO.

Output 5.2 Results of Retrieving IMS Data Using a WHERE Statement

```
                                    The SAS System
                         IMS Data Retrieved Using a WHERE Statement

SOC_SEC_
NUMBER        CUSTOMER_NAME                        ADDR_LINE_1              ADDR_LINE_2
CITY                       STATE  COUNTRY          ZIP_CODE   HOME_PHONE   OFFICE_PHONE
-----------------------------------------------------------------------------------------------
456-45-3462   LITTLE, NANCY M.                                             4543 ELGIN AVE.
RICHMOND                   VA     USA              26502-3317 803-657-3566

234-74-4612   WIKOWSKI, JONATHAN S.                                        4356 CAMPUS DRIVE
RICHMOND                   VA     USA              26502-5317 803-467-4587  803-654-7238
```

Updating Data with the SQL Procedure

You can use the UPDATE statement to update the data in an IMS database as was done earlier in this section using the FSEDIT procedure. Remember that when you reference a view descriptor in an SQL procedure statement, you are updating the IMS data that is described by the view descriptor, not the view descriptor itself. Use the WHERE clause to position the IMS engine on the database segment to be updated by specifying values for the key fields of parent segments.

The following UPDATE statements update the values that are contained in the last observation of VLIB.CUSTINFO:

```
update vlib.custinfo
   set zip_code = '27702-3317'
   where soc_sec_number = '234-74-4612';

update vlib.custinfo
   set addr_line_2 = '151 Knox St.'
   where soc_sec_number = '234-74-4612';

title2 'Updated Data in IMS ACCTDBD Database';
select *
  from vlib.custinfo
  where soc_sec_number = '234-74-4612';
```

The SELECT statement in this example retrieves and displays the updated data in the following output. (Because you are referencing a view descriptor, you use the SAS names for items in the UPDATE statement; the SQL procedure displays the variable labels as stored in the view.)

Output 5.3 Results of Updating IMS Data with the UPDATE Statement

```
                                    The SAS System
                         Updated Data in IMS ACCTDBD Database

SOC_SEC_
NUMBER        CUSTOMER_NAME                        ADDR_LINE_1              ADDR_LINE_2
CITY                       STATE  COUNTRY          ZIP_CODE   HOME_PHONE   OFFICE_PHONE
-----------------------------------------------------------------------------------------------
234-74-4612   WIKOWSKI, JONATHAN S.                                        151 Knox St.
RICHMOND                   VA     USA              27702-3317 803-467-4587  803-654-7238
```

Inserting and Deleting Data with the SQL Procedure

You can use the INSERT statement to add segments to an IMS database or use the DELETE statement to remove segments from an IMS database, as you did earlier in this section with the FSEDIT procedure. When inserting children under a parent segment, you must indicate the key values of the parent segments in the SET= statement. Use a view descriptor that describes the entire path of data down to the lowest segment to be inserted. In the following example, the root segment that contains the value **234-74-4612** for the SOC_SEC_NUMBER variable is deleted from the ACCTDBD database. Note that any child segments that exist under the parent segment in this example will also be deleted.

```
options linesize=132;

  proc sql;
  delete from vlib.custinfo
     where soc_sec_number = '234-74-4612';

  title2 'Observation Deleted from IMS
          ACCTDBD Database';
  select *
     from vlib.custinfo;
```

The SELECT statement then displays the data for VLIB.CUSTINFO in the following output.

Output 5.4   Results of Deleting IMS Data with the DELETE Statement

```
                                      The SAS System
                            Observation Deleted from IMS ACCTDBD Database

   SOC_SEC_
   NUMBER          CUSTOMER_NAME                          ADDR_LINE_1                  ADDR_LINE_2
   CITY                            STATE  COUNTRY         ZIP_CODE   HOME_PHONE  OFFICE_PHONE
   -----------------------------------------------------------------------------------------------------------
   667-73-8275   WALLS, HOOPER J.                                                4525 CLARENDON RD
   RAPIDAN                         VA     USA             22215-5600 803-657-3098 803-645-4418

   434-62-1234   SUMMERS, MARY T.                                                4322 LEON ST.
   GORDONSVILLE                    VA     USA             26001-0670 803-657-1687

   436-42-6394   BOOKER, APRIL M.                                                9712 WALLINGFORD PL.
   GORDONSVILLE                    VA     USA             26001-0670 803-657-1346

   434-62-1224   SMITH, JAMES MARTIN                                             133 TOWNSEND ST.
   GORDONSVILLE                    VA     USA             26001-0670 803-657-3437

   178-42-6534   PATTILLO, RODRIGUES                                             9712 COOK RD.
   ORANGE                          VA     USA             26042-1650 803-657-1346 803-657-1345

   156-45-5672   O'CONNOR, JOSEPH                                                235 MAIN ST.
   ORANGE                          VA     USA             26042-1650 803-657-5656 803-623-4257

   657-34-3245   BARNHARDT, PAMELA S.                                            RT 2 BOX 324
   CHARLOTTESVILLE                 VA     USA             25804-0997 803-345-4346 803-355-2543

   667-82-8275   COHEN, ABRAHAM                                                  2345 DUKE ST.
   CHARLOTTESVILLE                 VA     USA             25804-0997 803-657-7435 803-645-4234

   456-45-3462   LITTLE, NANCY M.                                                4543 ELGIN AVE.
   RICHMOND                        VA     USA             26502-3317 803-657-3566
```

CAUTION:

Use a WHERE clause in a DELETE statement in the SQL procedure. If you omit the WHERE clause from the DELETE statement in the SQL procedure, you delete the lowest level segment for each database path that is defined by the view descriptor in the IMS database. If the view descriptor describes only the root segment, the entire database will be deleted. △

For more information about the SQL procedure, see the *SAS SQL Procedure User's Guide*.

Updating Data with the MODIFY Statement

The MODIFY statement extends the capabilities of the DATA step by enabling you to modify IMS data that is accessed by a view descriptor or a SAS data file without creating an additional copy of the file. To use the MODIFY statement with a view descriptor, you must have update privileges defined in the PCB associated with the view, even if your program doesn't intend to modify the data.

You can specify either a view descriptor or a SAS data file as the data set to be opened for update by using the MODIFY statement. In the following example, the data set to be opened for update is the view descriptor VLIB.CUSTINFO, which describes data in the IMS sample database ACCTDBD. See Appendix 2, "Example Data," on page 255 for the code used to generate this view descriptor and the access descriptor MYLIB.ACCOUNT. The updates made to VLIB.CUSTINFO will be used to change the data in the ACCTDBD database. In order to update VLIB.CUSTINFO, you create a SAS data set, MYDATA.PHONENUM, to supply transaction information.

The MODIFY statement updates the ACCTDBD database with data from the MYDATA.PHONENUM data set in the following example:

```
data vlib.custinfo
    work.phoneupd (keep=soc_sec_number home_phone
        office_phone)
    work.nossnumb (keep=soc_sec_number home_phone
        office_phone);
  modify vlib.custinfo mydata.phonenum;
  by soc_sec_number;
  select (_iorc_);
    when (%sysrc(_sok))
  /* soc_sec_number found in ACCTDBD        */
      do;
        replace vlib.custinfo;
        output phoneupd;
      end;
    when (%sysrc(_dsenmr))
  /* soc_sec_number not found in ACCTDBD    */
      do;
        _error_=0;
        output nossnumb;
  /* stores misses in NOSSNUMB      */
      end;
    otherwise
  /* traps unexpected outcomes      */
      do;
        put 'Unexpected error condition:
            _iorc_ = ' _iorc_;
```

```
          put 'for SOC_SEC_NUMBER=' soc_sec_number
             '. Data step continuing.';
          _error_=0;
       end;
   end;
run;
```

For each iteration of the DATA step, SAS attempts to read one observation (or record) of the ACCTDBD database as defined by VLIB.CUSTINFO, based on SOC_SEC_NUMBER values supplied by MYDATA.PHONENUM. If a match on SOC_SEC_NUMBER values is found, the current segment data in ACCTDBD is replaced with the updated information in MYDATA.PHONENUM, then SOC_SEC_NUMBER, HOME_PHONE and OFFICE_PHONE are stored in the PHONEUPD data file. If the SOC_SEC_NUMBER value supplied by MYDATA.PHONENUM has no match in VLIB.CUSTINFO, the transaction information is written to the data file NOSSNUMB.

To further understand this type of processing, be aware that for each iteration of the DATA step (that is, each execution of the MODIFY statement), MYDATA.PHONENUM is processed sequentially. For each iteration, the current value of SOC_SEC_NUMBER is used to attach a WHERE clause to a request for an observation from VLIB.CUSTINFO as defined by the view. The engine then tries to generate a retrieval request with a qualified SSA from the WHERE clause. If the engine generates a qualified SSA, a GET UNIQUE call is issued, and data that is defined by the view is accessed directly. If the engine cannot generate a qualified SSA from the WHERE clause, a sequential pass of the database is required for each transaction observation in MYDATA.PHONENUM.

To print the PHONEUPD data file to see the SOC_SEC_NUMBER items that were updated, submit the following statements.

```
/*  Print data set named phoneupd  */
proc print data=work.phoneupd nodate;
  title2 'SSNs updated.';
run;
```

The results are shown in the following output:

Output 5.5   Contents of the PHONEUPD Data File

```
                     The SAS System
                     SSNs updated.

          SOC_SEC_
   OBS     NUMBER          HOME_PHONE         OFFICE_PHONE

    1     667-73-8275     703-657-3098      703-645-4418
    2     434-62-1234     703-645-441
    3     178-42-6534     703-657-1346      703-657-1345
    4     156-45-5672     703-657-5656      703-623-4257
    5     657-34-3245     703-345-4346      703-355-5438
    6     456-45-3462     703-657-3566      703-645-1212
```

To print the NOSSNUMB data set to see the SOC_SEC_NUMBER items that were not updated submit the following statements.

```
/*  Print data set named nossnumb  */
proc print data=work.nossnumb nodate;
   title2 'SSNs not updated.';
```

```
run;
```

The results produced are shown in the following ouptut:

Output 5.6   Contents of the NOSSUNUMB Data File

```
                    The SAS System
                  SSNs not updated.

            SOC_SEC_
    OBS      NUMBER         HOME_PHONE       OFFICE_PHONE

     1     416-41-3162    703-657-3166     703-615-1212
```

Updating SAS Files with IMS Data

You can update a SAS data file or data set with IMS data that is described by a view descriptor just as you can update a SAS data file with data from another SAS data file.

Suppose you have a SAS data set, MYDATA.BIRTHDAY, that contains employee ID numbers, last names, and birthdays. (See Appendix 2, "Example Data," on page 255 for a description of MYDATA.BIRTHDAY.) You want to update this data set with data that is described by VLIB.EMPBDAY, a view descriptor that is based on the IMS EMPLINF2 database. To perform this update, enter the following SAS statements:

```
libname vlib 'sas-data-library';
libname mydata 'sas-data-library';
options nodate;

/*-------------------------------------------*/
/*  Update the BIRTHDAY SAS data set         */
/*  with data from IMS                       */
/*  EMPLINF2 database                        */
/*-------------------------------------------*/
options linesize=80;
proc sort data=mydata.birthday;
  by employee_id;
run;

proc print data=mydata.birthday;
  title2 'Sorted SAS Data Set MYDATA.BIRTHDAY';
run;

proc print data=vlib.empbday;
  title2 'Data Described by VLIB.EMPBDAY';
run;

data mydata.newbday;
  update mydata.birthday vlib.empbday;
  by employee_id;
run;

proc print data=mydata.newbday;
  title2 'SAS Data Set MYDATA.NEWBDAY';
run;
```

The EMPLINF2 database is a HIDAM database whose root segment is sequenced by the key field EMPID, so when the UPDATE statement references the view descriptor VLIB.EMPBDAY, the data is presented to SAS for updating in sorted order by EMPLOYEE_ID. However, the SAS data set MYDATA.BIRTHDAY must be sorted before the update because the UPDATE statement expects both the original file and the transaction file to be sorted by the same BY variable.

The following three outputs show the results of the print procedures.

Output 5.7   Data Set to be Updated, MYDATA.BIRTHDAY, in EMPID Order

```
             The SAS System
       Sorted SAS Data Set MYDATA.BIRTHDAY


          employee_
   OBS       id        last_name      birthday

   1        1005       Knapp          06OCT38
   2        1024       Mueller        17JUN53
   3        1078       Gibson         23APR36
   4        1247       Garcia         04APR54
```

Output 5.8   IMS Data That is Described by the View Descriptor VLIB.EMPBDAY

```
                           The SAS System
                      Data Described by VLIB.EMPBDAY

             EMPLOYEE_                                          PHONE_
     OBS        ID       LAST_NAME      FIRST_NAME    BIRTHDAY  EXTENSION

       1       1001     Waterhouse     Clifton P.     01JAN48   X5109
       2       1002     Bowman         Hugh E.        14JUL31   X5901
       3       1003     Salazar        Yolanda        12DEC40   X5169
       4       1004     Knight         Althea         09APR50   X5218
       5       1005     Knapp          Patrice R.     04OCT37   X5012
       6       1006     Garrett        Olan M.        23JAN35   X5208
       7       1007     Brown          Virgina P.     24MAY46   X5258
       8       1008     Hernandez      Jesse L.       26MAR33   X5448
       9       1009     Jones          Michael Y.     21MAY31   X5713
      10       1010     Smith          Janet F.       07AUG47   X5621
      11       1011     Van Hotten     Gwendolyn      13SEP42   X5311
      12       1012     Quintero       Pedro          21FEB48   X5348
      13       1015     Scholl         Madison A.     19MAR45   X5419
      14       1017     Waggonner      Merrilee D.    27APR36   X5914
      15       1020     Rudd           Fred                 .
      16       1024     Mueller        Patsy          17JUN52   X5822
      17       1031     Chan           Tai            04JUL46   X5331
      18       1049     Fernandez      Sophia         11SEP44   X5847
      19       1050     Ameer          David          10OCT51   X5495
      20       1062     Littlejohn     Fannie         17MAY54   X5653
      21       1067     Cahill         Jacob          25DEC40   X5042
      22       1071     Canady         Frank A.       19NOV41   X5406
      23       1074     Millsap        Joel B.        12JUN36   X5224
      24       1077     Gibson         Teddy B.       23APR46   X5703
      25       1078     Gibson         George J.      23APR46   X5703
      26       1083     Savage         William D.     20JAN53   X5505
      27       1086     Schmidt        Penny          19FEB27   X5822
      28       1092     Polanski       Ivan L.        11OCT47   X5621
      29       1101     Nathaniel      Darryl         21MAR44   X5544
      30       1105     Faulkner       Carrie Ann     17AUG51   X5417
      31       1112     Jones          Rita M.        24DEC48   X5271
      32       1119     Goodson        Alan F.        21JUN50   X5512
      33       1120     Reid           David G.       15AUG45   X5369
      34       1123     Freeman        Leopold        09FEB35   X5604
      35       1133     Williamson     Janice L.      19MAY52   X5802
      36       1139     Seaton         Gary           03OCT56   X5545
      37       1145     Juarez         Armando        28MAY47   X5987
      38       1156     Reed           Kenneth D.     05JAN55   X5307
      39       1161     Richardson     Travis Z.      30NOV37   X5325
      40       1213     Johnson        Bradford       15APR54   X5446
      41       1217     Rodriguez      Romualdo R.    09FEB29   X5874
      42       1219     Kaatz          Freddie        21JUN57   X5387
      43       1234     Shropshire     Leland G.      04SEP49   X5616
      44       1238     Throckmort     Stewart Q.     04AUG31   X5391
      45       1247     Garcia         Francisco      05MAY55   X5348
      46       1261     Collins        Lillian        01MAY51   X5616
      47       1265     Slye           Leonard R.     18DEC60   X5123
      48       1266     Redfox         Richard B.     04APR44   X5386
      49       1272     Smith          Garland P.     05APR54   X5415
      50       1313     Smith          Jerry Lee      13SEP42   X5169
      51       1327     Brooks         Ruben R.       25FEB52   X5347
      52       1900     Smith          John                 .
```

Output 5.9   Data in the New Data Set MYDATA.NEWBDAY

```
                             The SAS System
                        SAS Data Set MYDATA.NEWBDAY

        employee_                                                PHONE_
OBS        id        last_name     birthday    FIRST_NAME        EXTENSION

  1       1001       Waterhouse    01JAN48     Clifton P.        X5109
  2       1002       Bowman        14JUL31     Hugh E.           X5901
  3       1003       Salazar       12DEC40     Yolanda           X5169
  4       1004       Knight        09APR50     Althea            X5218
  5       1005       Knapp         04OCT37     Patrice R.        X5012
  6       1006       Garrett       23JAN35     Olan M.           X5208
  7       1007       Brown         24MAY46     Virgina P.        X5258
  8       1008       Hernandez     26MAR33     Jesse L.          X5448
  9       1009       Jones         21MAY31     Michael Y.        X5713
 10       1010       Smith         07AUG47     Janet F.          X5621
 11       1011       Van Hotten    13SEP42     Gwendolyn         X5311
 12       1012       Quintero      21FEB48     Pedro             X5348
 13       1015       Scholl        19MAR45     Madison A.        X5419
 14       1017       Waggonner     27APR36     Merrilee D.       X5914
 15       1020       Rudd             .        Fred
 16       1024       Mueller       17JUN52     Patsy             X5822
 17       1031       Chan          04JUL46     Tai               X5331
 18       1049       Fernandez     11SEP44     Sophia            X5847
 19       1050       Ameer         10OCT51     David             X5495
 20       1062       Littlejohn    17MAY54     Fannie            X5653
 21       1067       Cahill        25DEC40     Jacob             X5042
 22       1071       Canady        19NOV41     Frank A.          X5406
 23       1074       Millsap       12JUN36     Joel B.           X5224
 24       1077       Gibson        23APR46     Teddy B.          X5703
 25       1078       Gibson        23APR46     George J.         X5703
 26       1083       Savage        20JAN53     William D.        X5505
 27       1086       Schmidt       19FEB27     Penny             X5822
 28       1092       Polanski      11OCT47     Ivan L.           X5621
 29       1101       Nathaniel     21MAR44     Darryl            X5544
 30       1105       Faulkner      17AUG51     Carrie Ann        X5417
 31       1112       Jones         24DEC48     Rita M.           X5271
 32       1119       Goodson       21JUN50     Alan F.           X5512
 33       1120       Reid          15AUG45     David G.          X5369
 34       1123       Freeman       09FEB35     Leopold           X5604

 35       1133       Williamson    19MAY52     Janice L.         X5802
 36       1139       Seaton        03OCT56     Gary              X5545
 37       1145       Juarez        28MAY47     Armando           X5987
 38       1156       Reed          05JAN55     Kenneth D.        X5307
 39       1161       Richardson    30NOV37     Travis Z.         X5325
 40       1213       Johnson       15APR54     Bradford          X5446
 41       1217       Rodriguez     09FEB29     Romualdo R.       X5874
 42       1219       Kaatz         21JUN57     Freddie           X5387
 43       1234       Shropshire    04SEP49     Leland G.         X5616
 44       1238       Throckmort    04AUG31     Stewart Q.        X5391
 45       1247       Garcia        05MAY55     Francisco         X5348
 46       1261       Collins       01MAY51     Lillian           X5616
 47       1265       Slye          18DEC60     Leonard R.        X5123
 48       1266       Redfox        04APR44     Richard B.        X5386
 49       1272       Smith         05APR54     Garland P.        X5415
 50       1313       Smith         13SEP42     Jerry Lee         X5169
 51       1327       Brooks        25FEB52     Ruben R.          X5347
 52       1900       Smith            .        John
```

Appending IMS Data with the APPEND Procedure

You can append data that is described by SAS/ACCESS view descriptors and PROC SQL views to SAS data files and vice versa. You can also append data from one view descriptor to the data from another.

In the following example, two branch managers have kept separate records on customers' checking accounts. One manager has kept records in the CUSTOMER and CHCKACCT segments of the IMS database ACCTDBD, described by the view descriptor VLIB.CHCKACCT. The other manager has kept records in a Version 7 SAS data set, MYDATA.CHECKS. Due to a corporate reorganization, the two sources of data must be combined so that all customer data is stored in the IMS database ACCTDBD. A branch manager can use the APPEND procedure to perform this task, as the following example demonstrates.

```
options linesize=120;

   proc print data=vlib.chckacct;
      title2 'Data Described by VLIB.CHCKACCT';
   run;

   proc print data=mydata.checks;
      title2 'Data in MYDATA.CHECKS Data Set';
   run;
```

The data that is described by the VLIB.CHCKACCT view descriptor and the data in the SAS data set MYDATA.CHECKS are displayed in the following two outputs.

Output 5.10   Data That is Described by the VLIB.CHCKACCT View Descriptor

```
                                 The SAS System
                         Data Described by VLIB.CHCKACCT


         SOC_SEC_                                  CHECK_ACCOUNT_    CHECK_        CHECK_
   OBS    NUMBER     CUSTOMER_NAME                     NUMBER        DATE         BALANCE


    1   667-73-8275   WALLS, HOOPER J.               345620145345   15MAR95      1266.34
    2   667-73-8275   WALLS, HOOPER J.               345620154633   28MAR95      1298.04
    3   434-62-1234   SUMMERS, MARY T.               345620104732   27MAR95       825.45
    4   436-42-6394   BOOKER, APRIL M.               345620135872   26MAR95       234.89
    5   434-62-1224   SMITH, JAMES MARTIN            345620134564   16MAR95      2645.34
    6   434-62-1224   SMITH, JAMES MARTIN            345620134663   24MAR95       143.78
    7   178-42-6534   PATTILLO, RODRIGUES            745920057114   10JUN95      1502.78
    8   156-45-5672   O'CONNOR, JOSEPH               345620123456   27MAR95       463.23
    9   657-34-3245   BARNHARDT, PAMELA S.           345620131455   29MAR95      1243.25
   10   667-82-8275   COHEN, ABRAHAM                 382957492811   03APR95      7302.06
   11   456-45-3462   LITTLE, NANCY M.               345620134522   25MAR95       831.65
```

Output 5.11 Data in the MYDATA.CHECKS Data Set

```
                              The SAS System
                        Data in MYDATA.CHECKS Data Set

                                             check_
                               soc_sec_      account_         check_      check_
     OBS     customer_name      number        number         balance      date

      1    COWPER, KEITH       241-98-4542  183352795865       862.31     25MAR95
      2    OLSZEWSKI, STUART   309-22-4573  382654397566       486.00     02APR95
      3    NAPOLITANO, BARBARA 250-36-8831  284522378774       104.20     10APR95
      4    MCCALL, ROBERT      367-34-1543  644721295973       571.92     05APR95
```

Note: To use PROC APPEND, you must use a view descriptor that describes the entire path of data from the root segment down to the level where you want to append data. If a parent segment already exists with a key value equal to that specified in the input data set, the IMS engine inserts the remaining path of data under the parent segment. △

You can combine the data from these two sources using the APPEND procedure, as shown in the following example:

```
proc append base=vlib.chckacct data=mydata.checks;
   run;

   proc print data=vlib.chckacct;
     title2 'Appended Data';
   run;

   proc sql;
     delete from vlib.account
     where soc_sec_number in( '241--98--4542'
                              '250--36--8831'
                              '309--22--4573'
                              '367--34--1543' )
   run;
```

The database type determines where the segments are inserted. In this case, the database type is not an indexed database type, so the data in MYDATA.CHECKS is intermixed with the data that is described by VLIB.CHCKACCT. The following output displays the updated data that is described by the view descriptor, VLIB.CHCKACCT.

Output 5.12   Results of Appending Data with the APPEND Procedure

```
                                     The SAS System
                                     Appended Data

           SOC_SEC_                                     CHECK_ACCOUNT_     CHECK_        CHECK_
    OBS    NUMBER       CUSTOMER_NAME                        NUMBER         DATE        BALANCE


     1    667-73-8275   WALLS, HOOPER J.                  345620145345    15MAR95      1266.34
     2    667-73-8275   WALLS, HOOPER J.                  345620154633    28MAR95      1298.04
     3    434-62-1234   SUMMERS, MARY T.                  345620104732    27MAR95       825.45
     4    250-36-8831   NAPOLITANO, BARBARA               284522378774    10APR95       104.20
     5    241-98-4542   COWPER, KEITH                     183352795865    25MAR95       862.31
     6    436-42-6394   BOOKER, APRIL M.                  345620135872    26MAR95       234.89
     7    434-62-1224   SMITH, JAMES MARTIN               345620134564    16MAR95      2645.34
     8    434-62-1224   SMITH, JAMES MARTIN               345620134663    24MAR95       143.78
     9    178-42-6534   PATTILLO, RODRIGUES               745920057114    10JUN95      1502.78
    10    367-34-1543   MCCALL, ROBERT                    644721295973    05APR95       571.92
    11    156-45-5672   O'CONNOR, JOSEPH                  345620123456    27MAR95       463.23
    12    657-34-3245   BARNHARDT, PAMELA S.              345620131455    29MAR95      1243.25
    13    667-82-8275   COHEN, ABRAHAM                    382957492811    03APR95      7302.06
    14    456-45-3462   LITTLE, NANCY M.                  345620134522    25MAR95       831.65
    15    309-22-4573   OLSZEWSKI, STUART                 382654397566    02APR95       486.00
```

Note: The APPEND procedure issues a warning message when a variable in the view descriptor does not have a corresponding variable in the input data set. △

The PROC SQL code deletes the appended data so that the next PROC APPEND example will work without reinitializing the database.

You can use the APPEND procedure's FORCE option to force PROC APPEND to concatenate two data sets that have different variables or variable attributes.

The APPEND procedure also accepts a SAS WHERE statement to retrieve a subset of the data. In the following example, a subset of the observations from the DATA= data set is added to the BASE= data set.

```
proc append base=vlib.chckacct data=mydata.checks
   (where=(check_date >='26MAR95'd));
run;

proc print data=vlib.chckacct;
   title2 'Appended Data with a WHERE Data Set
           Option';
run;
```

Note that the WHERE= data set option applies only to the MYDATA.CHECKS data set. The following output displays the results.

Output 5.13 Results of Appending Data with a WHERE= Data Set Option

```
                                   The SAS System
                         Appended Data with a WHERE= Data Set Option

           SOC_SEC_                                  CHECK_ACCOUNT_     CHECK_        CHECK_
    OBS    NUMBER       CUSTOMER_NAME                    NUMBER          DATE        BALANCE


     1     667-73-8275  WALLS, HOOPER J.               345620145345    15MAR95       1266.34
     2     667-73-8275  WALLS, HOOPER J.               345620154633    28MAR95       1298.04
     3     434-62-1234  SUMMERS, MARY T.               345620104732    27MAR95        825.45
     4     250-36-8831  NAPOLITANO, BARBARA            284522378774    10APR95        104.20
     5     436-42-6394  BOOKER, APRIL M.               345620135872    26MAR95        234.89
     6     434-62-1224  SMITH, JAMES MARTIN            345620134564    16MAR95       2645.34
     7     434-62-1224  SMITH, JAMES MARTIN            345620134663    24MAR95        143.78
     8     178-42-6534  PATTILLO, RODRIGUES            745920057114    10JUN95       1502.78
     9     367-34-1543  MCCALL, ROBERT                 644721295973    05APR95        571.92
    10     156-45-5672  O'CONNOR, JOSEPH               345620123456    27MAR95        463.23
    11     657-34-3245  BARNHARDT, PAMELA S.           345620131455    29MAR95       1243.25
    12     667-82-8275  COHEN, ABRAHAM                 382957492811    03APR95       7302.06
    13     456-45-3462  LITTLE, NANCY M.               345620134522    25MAR95        831.65
    14     309-22-4573  OLSZEWSKI, STUART              382654397566    02APR95        486.00
```

Note that the IMS engine has no way to determine how large a database is. Therefore, if you use the APPEND procedure to add a database to itself, a loop can result. For more information about the APPEND procedure, see " APPEND Procedure" in the *Base SAS Procedures Guide*.

P A R T *3*
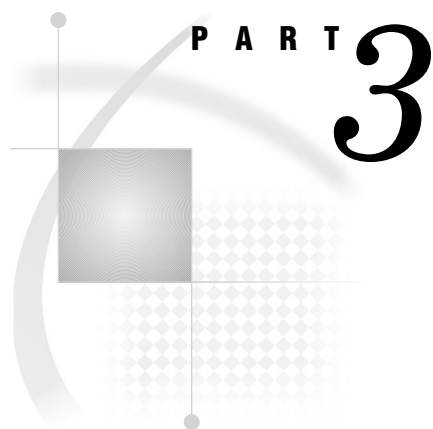
SAS/ACCESS Interface to the IMS Engine: Reference

CHAPTER

6
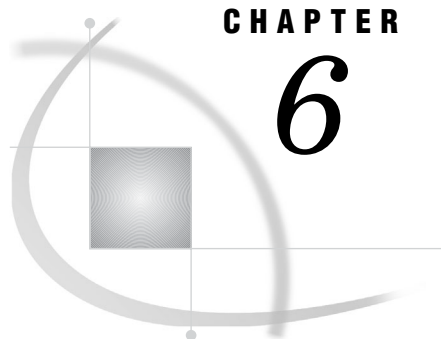
ACCESS Procedure Reference

Introduction to ACCESS Procedure Reference

The ACCESS procedure enables you to create and edit the descriptor files that are used by the SAS/ACCESS interface view engine to IMS (hereafter referred to as the IMS engine). The ACCESS procedure can be used in batch, interactive line, and noninteractive modes.

This section provides complete reference information for the ACCESS procedure. The PROC ACCESS statement is presented first, followed by the statement options and procedure statements. For examples of how to use the statement options, refer to "Invoking the ACCESS Procedure" on page 100. "Performance and Efficient View Descriptors" on page 124 presents several efficiency considerations for using the SAS/ACCESS interface to IMS.

Refer to the *SAS Language Reference: Dictionary* and the *SAS Companion for z/OS* for information about SAS data sets, data libraries, and their naming conventions, or for help with the terminology used in this procedure description.

ACCESS Procedure Syntax for IMS

PROC ACCESS *<options>*;

Creating or Updating Statement
  CREATE *libref.member-name.*ACCESS|VIEW;
  UPDATE *libref.member-name.*ACCESS|VIEW;

Database-Definition Statements
  DATABASE=*database-name* DBTYPE=*database-type*;
  RECORD=*record-name* SEGMENT=*segment-name*
      SEGLNG=*segment-length*;
  GROUP=*group-name* LEVEL=*level-number*
      KEY=Y|N|U OCCURS=*number-of-repeats*
      SEARCH=*search-name*;
  ITEM=*item-name* LEVEL=*level-number*
      DBFORMAT=*database-format*
      FORMAT=*SAS-format* SEARCH=*search-name*
      KEY=Y|N|U OCCURS=*number-of-repeats*
      DBCONTENT=*database-content*;
  DELETE *item-name*|*index-number*;
  INSERT*item-name*|*index-number*;
  REPLACE *item-name*|*index-number*;

Editing Statements
  AN=Y|N;
  UN=Y|N;
  DROP *item-name*|*index-number*... ;
  FORMAT *item-name*|*index-number <=> format*... ;
  LIST ALL|VIEW|*index-number*|*item-name <blanks*|DB|DESC>;
  QUIT;
  RENAME *item-name*|*index-number <=> SAS-name*... ;
  RESET ALL|*item-name*|*index-number* ... ;
  SELECT ALL|*item-name*|*index-number*... ;
  SUBSET *selection-criteria*;

RUN;

Description

The ACCESS procedure is used to create and edit access descriptors and view descriptors, and to create SAS data files. Descriptor files describe DBMS data so that

you can read, update, or extract the DBMS data directly from within a SAS session or in a SAS program.

The following sections provide more information about the syntax of the PROC ACCESS statement.

PROC ACCESS Statement Options

To create and edit access and view descriptor files, you must issue the PROC ACCESS statement with options and procedure statements. The statement has this format:

PROC ACCESS *<options>*;

> *required-procedure-statements;*

> *optional-procedure-statements;*

This section describes PROC ACCESS options. For information about the procedure statements, see "Invoking the ACCESS Procedure" on page 100.

Depending on which options you choose, the ACCESS procedure performs several tasks. To create and edit access and view descriptors, use the following options:

DBMS=IMS
> specifies the name of the database management system that the access descriptor will access. Specify DBMS=IMS since you are using the SAS/ACCESS interface to IMS.

ACCDESC=*libref.access-descriptor*
> specifies the name of an access descriptor.
> ACCDESC= is used with the DBMS= option to create a view descriptor that is based on the specified access descriptor. You specify the view descriptor's name in the CREATE statement. You can also use a data set option on the ACCDESC= option to specify any passwords that have been assigned to the access descriptor. The access descriptor that you name must exist.
> The ACCDESC= option has two aliases: AD= and ACCESS=.

The following options enable you to extract IMS data with a view descriptor:

VIEWDESC=*<libref.>view-descriptor*
> specifies the name of the view descriptor from which to extract the IMS data.

OUT=*<libref.>member-name*
> specifies the SAS data file to which DBMS data is written. OUT= is used only with the VIEWDESC= option.

SAS Passwords for SAS/ACCESS Descriptors

SAS enables you to control access to SAS data sets and access descriptors by associating one or more SAS passwords with them. You must first create the descriptor files before assigning SAS passwords to them, as described in "Assigning Passwords" on page 98.

The following table summarizes the levels of protection that SAS passwords have and their effects on access descriptors and view descriptors.

Table 6.1   Password and Descriptor Interaction

	READ=	WRITE=	ALTER=
access descriptor	no effect on descriptor	no effect on descriptor	protects descriptor from being read or edited
view descriptor	protects DBMS data from being read or edited	protects DBMS data from being edited	protects descriptor from being read or edited

When you create view descriptors, you can use a data set option after the ACCDESC= option to specify the access descriptor's password (if one exists). In this case, you are *not* assigning a password to the view descriptor that is being created. Rather, using the password grants you permission to use the access descriptor to create the view descriptor. For example:

```
proc access dbms=ims accdesc=mylib.account(alter=rouge);
  create vlib.customer.view;
  select all;
run;
```

By specifying the ALTER-level password, you can read the MYLIB.ACCOUNT access descriptor and therefore create the VLIB.CUSTOMER view descriptor.

For detailed information about the levels of protection and the types of passwords you can use, refer to *SAS Language Reference: Dictionary*. The following section describes how you assign SAS passwords to descriptors.

Assigning Passwords

You can assign, change, or clear a password for an access descriptor, a view descriptor, or another SAS file by using the DATASETS procedure's MODIFY statement. Here is the basic syntax for using PROC DATASETS to assign a password to an access descriptor, a view descriptor, or a SAS data file:

PROC DATASETS LIBRARY= *libref* MEMTYPE= *member-type* ;
   MODIFY *member-name* (*password-level* = *password-modification*);

RUN;

The *password-level* argument can have one or more of the following values: READ=, WRITE=, ALTER=, or PW=. PW= assigns read, write, and alter privileges to a descriptor or data file. The *password-modification* argument enables you to assign a new password or to change or delete an existing password.

For example, this PROC DATASETS statement assigns the password REWARD with the ALTER level of protection to the access descriptor MYLIB.EMPLOYEE:

```
proc datasets library=mylib memtype=access;
   modify employee (alter=reward);
run;
```

In this case, users are prompted for the password whenever they try to browse or edit the access descriptor or to create view descriptors that are based on MYLIB.EMPLOYEE.

You can assign multiple levels of protection to a descriptor or SAS data file. See "Ensuring IMS Data Security" on page 133 for more information about how to prevent unauthorized access to the data in your IMS databases.

In the next example, the PROC DATASETS statement assigns the passwords MYPW and MYDEPT with READ and ALTER levels of protection to the view descriptor VLIB.CUSTACCT:

```
proc datasets library=vlib memtype=view;
   modify custacct (read=mypw alter=mydept);
run;
```

In this case, users are prompted for the SAS password when they try to read the DBMS data, or try to browse or edit the view descriptor VLIB.CUSTACCT itself. You need both levels to protect the data and descriptor from being read. However, a user could still update the data that is accessed by VLIB.CUSTACCT, for example, by using a PROC SQL UPDATE. Assign a WRITE level of protection to prevent data updates.

To delete a password on an access descriptor or any SAS data set, put a slash after the password:

```
proc datasets library=vlib memtype=view;
   modify custacct (read=mypw/ alter=mydept/);
run;
```

In the following example, PROC DATASETS sets a READ and ALTER password for view descriptor VLIB.CUSTINFO. PROC PRINT tries to use the view descriptor with both an invalid and valid password. PROC ACCESS tries to update the view descriptor with and without a password.

```
/* Assign passwords */
proc datasets library=vlib memtype=view;
  modify custinfo (read=r2d2 alter=c3po);
run;


/* Invalid password given */
proc print data=vlib.custinfo (pw=r2dq);
  where soc_sec_number = '178-42-6534';
  title2 'Data for 178-42-6534';
run;


/* Valid password given */
proc print data=vlib.custinfo (pw=r2d2);
  where soc_sec_number = '178-42-6534';
  title2 'Data for 178-42-6534';
run;


/* Missing password */
proc access dbms=ims;
   update vlib.custinfo.view;
     drop country;
     list all;
run;


/* Valid password given */
proc access dbms=ims;
   update vlib.custinfo.view (alter=c3po);
     drop country;
     list all;
run;
```

Refer to *SAS Language Reference: Dictionary* for more examples of assigning, changing, deleting, and using SAS passwords.

Invoking the ACCESS Procedure

To invoke the ACCESS procedure you use the options described in "PROC ACCESS Statement Options" on page 97 and certain procedure statements. The options and statements you choose are defined by your task.

☐ To create an access descriptor:

> **PROC ACCESS** DBMS=IMS;
>
> > **CREATE** *libref.member-name*.ACCESS;
> > > *required database-description statements;*
> > > *optional editing statements;*
>
> > **RUN**;

☐ To create an access descriptor and a view descriptor in the same procedure:

> **PROC ACCESS** DBMS=IMS;
>
> > **CREATE** *libref.member-name*.ACCESS;
> > > *required database-description statements;*
> > > *optional editing statements;*
> > **CREATE** *libref.member-name*.VIEW;
> > > **SELECT** *item-list;*
> > > *optional editing statements;*
>
> > **RUN**;

☐ To create a view descriptor from an existing access descriptor:

> **PROC ACCESS** DBMS=IMS ACCDESC=*libref.access-descriptor*;
>
> > **CREATE** *libref.member-name*.VIEW;
> > > **SELECT** *item-list;*
> > > *optional editing statements;*
>
> > **RUN**;

☐ To update an access descriptor:

> **PROC ACCESS** DBMS=IMS;
>
> > **UPDATE** *libref.member-name*.ACCESS;
> > > *procedure statements;*
>
> > **RUN**;

☐ To update a view descriptor:

> **PROC ACCESS** DBMS=IMS;
>
> > **UPDATE** *libref.member-name*.VIEW;
> > > *procedure statements;*
>
> > **RUN**;

Note that when you update an access descriptor (for example, drop another field from the display), the view descriptors based on this access descriptor are not updated

automatically. You must re-create or modify any view descriptors that you want to reflect the changes made to the access descriptor. Altering a DBMS table can invalidate both access descriptors and view descriptors.

Database-Description Statements

The following statements define the IMS database in an access descriptor.

DATABASE=*database-name* DBTYPE=*database-type*;

RECORD=*record-name* SEGMENT=*segment-name*
    SEGLNG=*segment-length*;

GROUP=*group-name* LEVEL=*level-number*
    KEY=Y|N|U OCCURS=*number-of-repeats*
    SEARCH=*search-name*;

ITEM=*item-name* LEVEL=*level-number*

  DBFORMAT=*database-format*

  FORMAT=*SAS-format* SEARCH=*search-name*

  KEY=Y|N|U OCCURS=*number-of-repeats*

  DBCONTENT=*database-content;*

DELETE *item-name|index-number;*

INSERT *item-name|index-number;*

REPLACE *item-name|index-number;*

The DATABASE=, RECORD=, and ITEM= statements are required to create an access descriptor with the CREATE statement; the GROUP= statement is optional. The INSERT, DELETE, and REPLACE statements are used with the UPDATE statement to change an existing access descriptor. At least one of the GROUP=, RECORD=, or ITEM= statements must be used with the INSERT, DELETE, and REPLACE statements to change an access descriptor. The DATABASE= statement cannot be used in an UPDATE statement.

Whether you are creating or changing an access descriptor, the RECORD=, ITEM=, and GROUP= statements must be used in the same order as they appear in the database.

Because IMS does not have a dictionary or store descriptive information about the database, you need to provide the DBD information. To provide this information, you need to have a COBOL copybook or layout of the database.

For logical databases, the access descriptor definitions are mapped to the logical DBD and not to one or more physical DBDs. This enables the IMS engine to build correct calls and for the SSAs (segmented search arguments) to navigate the logical structure of the database.

Note: See "Tools for Creating IMS Access Descriptors" on page 123 for tools that SAS supplies to automate the database definition process. △

Editing Statements

SAS/ACCESS *editing statements* enable you to drop or rename items, list items, reset names, and so on in a descriptor. All of the statements can be used when you are creating a descriptor. The ASSIGN=, SELECT, RESET, and UNIQUE= statements cannot be used when you are changing a descriptor.

When creating or changing an access descriptor, place editing statements after the last database definition statement. All editing statements are optional.

The following list shows the basic syntax of each editing statement:

ASSIGN=Y | N;

UNIQUE=Y | N;

DROP *item-name | index-number*...;

FORMAT *item-name | index-number <=> format*...;

LIST ALL | VIEW | *index-number* | *item-name <blanks* | DB | DESC>;

QUIT | EXIT;

RENAME *item-name | index-number <=> SAS-name*...;

RESET ALL | *item-name | index-number*...;

SELECT ALL | *item-name | index-number*...;

SUBSET *selection-criteria*;

These statements are described in detail in the following sections.

ACCESS Procedure Statements for IMS

ASSIGN= Statement

Generates SAS names and formats that are based on item names and DB Formats

Optional statement

Applies to: access descriptor

Syntax

ASSIGN=Y | N;

Details

The ASSIGN= statement causes view descriptors to inherit the SAS variable names and formats of the parent access descriptor at the time that the descriptor is created. That is, if ASSIGN=Y, the variable names generated for the access descriptor will be used in all derived view descriptors, regardless of the naming conventions used.

If ASSIGN=N, which is the default value, you specify the SAS variable names and formats when you create a view descriptor from this access descriptor. The naming conventions used by the view descriptors are determined by examining the VALIDVARNAME SAS option. The VALIDVARNAME SAS option lets users specify what naming conventions will be allowed in a SAS session, and enforces them by converting variable names that do not conform to the necessary format. For more information about the VALIDVARNAME system option, see "Using the SAS/ACCESS Interface to IMS" on page 4 and *SAS Language Reference: Dictionary*.

If you enter a value of Y for this statement, you cannot specify the RENAME, FORMAT, and UN= statements when creating view descriptors that are based on this access descriptor.

When a new CREATE statement is entered, the ASSIGN= statement is reset to the default value, N.

AN= is a valid alias for this statement.

CREATE (Access Descriptor) Statement

Creates an access descriptor

Required statement

Applies to: access descriptor

Syntax

CREATE *libref.member.*ACCESS;

Details

The CREATE statement specifies a one- or two-level name for the access descriptor you want to create. The suffix specifies the member type ACCESS. You can use the CREATE statement in one procedure execution as many times as necessary.

To create an access descriptor, the CREATE statement must follow the PROC ACCESS statement. It is specified before any of the database description or editing statements , which are described later in this section.

When you submit a CREATE statement for processing, the statement is checked for errors and, if none are found, the access descriptor specified in the previous CREATE statement (if there is one) is saved. If errors are found, error messages are written to the SAS log and processing is terminated. After you correct the error, resubmit the statements or batch job for processing.

CREATE (View Descriptor) Statement

Creates a view descriptor

Required statement

Applies to: view descriptor

Syntax

CREATE *libref.member*.VIEW PSBNAME=*psb-name* <PCBINDEX=*pcb-index*>
   <GSAM>;

Details

The CREATE statement specifies a one- or two-level name for the view descriptor you want to create. The suffix specifies the member type VIEW. This statement is required to create and save a view descriptor.

   To create a view descriptor, add the CREATE statement after the procedure statements that create the access descriptor on which this view descriptor is based. If you are creating a view based on an existing access descriptor, specify the access descriptor's name in the ACCDESC= option in the PROC ACCESS statement.

   Place any editing statement and view-descriptor-specific statements, such as the SELECT and SUBSET statements, after the view descriptor's CREATE statement. You can submit more than one CREATE statement in one execution of the PROC ACCESS statement. As with other SAS procedures, end the ACCESS procedure with a RUN statement.

   When you submit a CREATE statement for processing, the statement is checked for errors and, if none are found, the view descriptor specified in the previous CREATE statement (if there is one) is saved. If errors are found, error messages are written to the SAS log and processing is terminated. After you correct the error, resubmit the statements or batch job for processing.

Arguments

The following list explains the arguments that can appear in a CREATE statement for a view descriptor:

PSBNAME= |PSB=
   specifies the name of the PSB that references the IMS database on which this view descriptor is based. This is a required argument.

PCBINDEX= |PCB=
   specifies the PCB in the PSB that references the database. This argument is optional; you need to specify a PCB index only if the PSB references the database more than once. If you do not specify a PCB index and the PSB references the database more than once, the first PCB in the PSB that references the database is used.

GSAM
   specifies that the database on which this descriptor is based is a GSAM database. Specify this argument only if you have a GSAM database.

DATABASE= Statement

Specifies the DBD name of the IMS database on which the access descriptor is based

Required statement

Applies to: access descriptor

Syntax

DATABASE=*database-name* DBTYPE=*database-type*;

Details

The DATABASE= statement specifies the DBD name of the IMS database on which the access descriptor is based. DBD= is an alias for the DATABASE= statement. If you are creating an access descriptor, the DATABASE= statement must be the first statement after the CREATE= statement.

For logical databases, the access descriptor definitions are mapped to the logical DBD (database description) and not to one or more physical DBDs. This enables the IMS engine to build correct database calls and for the SSAs (segmented search arguments) to navigate the logical structure of the database.

Note: See "Tools for Creating IMS Access Descriptors" on page 123 for tools that SAS supplies to automate the database definition process. △

Arguments

The following list explains the argument that can appear in a DATABASE= statement for an access descriptor:

DBTYPE= |DBT=
   specifies the type of database and is required with the DATABASE= statement. Valid database types are HDAM, HIDAM, HSAM, HISAM, GSAM, SHSAM, and SHISAM. See "IMS Database Types" on page 21 for a description of each database type. You can use DBT= as an alias for DBTYPE=.

   DBTYPE= tells the IMS engine how to handle WHERE clauses that generate SSAs for database calls. If you omit DBTYPE= from your DATABASE= statement, you receive the following error:

```
ERROR 22-322: Expecting one of the following:
DBTYPE = NAME. The statement is being ignored.

ERROR: Must enter database name first.
```

An example of the DATABASE= statement is as follows:

```
database=acctdbd dbtype=hisam;
```

DELETE Statement

Removes records, groups, or items from an existing access descriptor

Optional statement

Applies to: access descriptor

Interacts with: UPDATE statement

Syntax

DELETE | DEL *numeric-list*;

DELETE | DEL *item-name <... item-name-n>*;

Details

The DELETE statement deletes the specified record, group, or item from an access descriptor. You can specify as many records, groups, or items as you want in one DELETE statement. When you delete a group or record, all of the items in that group or record are deleted as well.

Note that if the first record of a descriptor is deleted, then the first item in the descriptor must still be a RECORD.

Arguments

The following arguments can appear in a DELETE statement. You can mix item names and quoted strings in the same DELETE statement, but you cannot mix index numbers and names. Referencing a list of index numbers is an efficient way to delete items like OCCURS clauses, which by definition are not unique.

numeric-list
   is a list of index numbers, optionally separated by logical operators, that represent the item's place in the access descriptor. You can obtain the index number of an item using the LIST statement described later in this section.

item-name
   is the name of the IMS group, record, or item to be deleted. This field can also contain a quoted string.

Examples

The following are examples of DELETE statements:

```
DELETE 15 2 8 TO 12;        /* deletes a numeric list */
DELETE 1 TO 23 BY 2;        /* deletes a numeric list */
DELETE CITY STATE ZIP;      /* deletes by name         */
DELETE CITY 'FIRST-ORDER-DATE'; /* deletes a name and quoted string */
```

DROP Statement

Drops the specified item so that it is no longer available for selection

Optional statement

Applies to: access descriptor or view descriptor

Syntax

DROP *numeric-list*;

DROP *item-name <... item-name-n>*;

Details

The DROP statement drops the specified item so that the item is no longer available for selection. When used in an access descriptor, it prevents the specified item from being available to a view descriptor. The DROP statement is used with the UPDATE statement in a view descriptor.

You can specify as many items to be dropped as necessary by using one DROP statement. You can identify items by their index number or by their name or a quoted string, but you cannot mix index numbers and names. If you drop a record or group, all the items in that record or group are dropped.

Arguments

The following arguments can appear in the DROP statement:

numeric-list
is a list of index numbers, optionally separated by logical operators, that represent the item's place in the descriptor. You can get the index number of an item by using the LIST statement described later in this section.

item-name
is the name of the IMS item to be dropped or a quoted string.

FORMAT Statement

Assigns a SAS format to an IMS item

Optional statement

Applies to: access descriptor or view descriptor

Syntax

FORMAT *item-name | index-number <=> format <... item-name-n | index-number-n <=> format-n>*;

Details

The FORMAT statement assigns a SAS format to an IMS item. You can assign formats to as many items as necessary using one FORMAT statement. Note that the equal sign (=) between arguments is optional. You cannot use the FORMAT statement for a record or group.

Arguments

The following list explains the arguments that can appear in the FORMAT statement:

item-name
   is the name of the IMS item for which you want to assign or change the SAS format.

index-number
   is the index number of the IMS item for which you want to assign or change the SAS format. The index number represents the item's place in the access descriptor. You can get the index number of an item using the LIST statement described later in this section.

format
   is the SAS format that you want to assign to the specified IMS item.

GROUP= Statement

Defines the groups within the record

Optional statement

Applies to: access descriptor

Syntax

GROUP= *group-name* LEVEL=*level-number* <KEY=Y|N|U>
    <OCCURS=*number-of-repeats*> <SEARCH=*search-name*>;

Details

The group name is the name that you want to assign to the group item in an IMS database. This name can be a maximum of 32 characters. If any special characters or blanks are included in the name, enclose the entire name in quotation marks. This is a required argument.

   The GROUP= statement defines the groups within the record. This statement is optional.

   See "Handling GROUP Keys in Descriptor Files" on page 137 for information about how to reference GROUP keys in a view descriptor's WHERE statement.

Arguments

In the GROUP= statement, you must enter the group name and level number. The other arguments are used to further define the group and are not required. The following list explains each argument that can appear in a GROUP= statement:

   LEVEL=
   LV=
      specifies the two-character numeric level of the IMS item. This level number is similar to the COBOL level number. Groups have levels greater than 01, and their

level numbers are less than the level numbers of the items within the group. This is a required argument.

KEY=
K=

indicates with an Y, N, or a U whether this item is defined in the DBD as a sequence or key field and whether the key sequence field is unique. The default setting, N, indicates the field is not a key sequence field. You must assign one key sequence field per segment if you plan to use the view descriptors that are created from this access descriptor to update the IMS database. Keys are recommended, but not required, for all segments except the lowest hierarchical level if the view descriptors will be used only for data retrieval. When KEY=U, retrieval calls to IMS will be reduced because the IMS engine will know that there is only one segment in the database for this key.

OCCURS=
O=

indicates the number of times a repeating group occurs. This is an optional argument.

SEARCH=
SE=

specifies the search field name defined for the group item in the DBMS DBD. If you want the IMS engine to create SSAs directly from a WHERE statement or command, you must enter the search field names. Otherwise, the WHERE statement is passed to SAS and all of the segments in the database that are referenced in the view descriptor are read. SEARCH= is an optional argument, but it is recommended where applicable.

   Note: See "Handling GROUP Keys in Descriptor Files" on page 137 for important information about searching at the GROUP level. Also see "Performance and Efficient View Descriptors" on page 124 for more information about SSAs and WHERE statements. △

INSERT Statement

Adds new records, groups, or items to an existing access descriptor

Optional statement

Applies to: access descriptor

Interacts with: UPDATE statement

Syntax

INSERT|INS *index-number*;

INSERT|INS *item-name <… item-name-n>*;

Details

The INSERT statement is a positioning statement; it inserts the RECORD=, GROUP=, or ITEM= statements following it after the item it references. The syntax and use of

the RECORD=, GROUP=, and ITEM= statements are the same in update mode as they are in create mode.

Although the INSERT statement can reference only one item, more than one RECORD=, GROUP=, or ITEM= statement can follow an INSERT statement. The INSERT statement retains control until it encounters an editing, LIST, DELETE, or REPLACE statement, or the ACCESS procedure ends through a QUIT, RUN, or other procedure statement. Multiple INSERT statements can be used in one UPDATE statement. When more than one INSERT statement references the same item, the most recent update displays as first.

Arguments

The following arguments can appear in a INSERT statement:

item-number
> is an index number that represents the item's place in the access descriptor. You can get the index number of an item by using the LIST statement described later in this section.

item-name
> is the name of the IMS group, record, or item after which subsequent groups, records, or items will be inserted. This field can also contain a quoted string.

Example

The following is an example of an INSERT statement. A new record and item are inserted at the beginning of the access descriptor ADLIB.CUSTINS. "INSERT 0" inserts items at the beginning of the descriptor. The first item in an access descriptor must always be a record. Also in the example, note that the first LIST statement prints a pre-update listing of the database as defined by the access descriptor, while the second prints a post-update listing.

```
proc access dbms=ims;
   update adlib.custins.access;
       list all db;
       insert 0;
         record=newfrec sg=newrecsg sl=400;
         item=newfitem lv=3 dbf=$12. se=custfsti;
       list all db;
run;
```

ITEM= Statement

Defines the fields within the record

Required statement

Applies to: access descriptor

Syntax

ITEM= *item-name* LEVEL=*level-number*
   DBFORMAT=*database-format* <SASNAME=*SAS-name*>

<FORMAT=*SAS-format* > <SEARCH=*search-name*>
<KEY=Y | N | U> <OCCURS=*number-of-repeats*>
<DBCONTENT=*database-content* >;

Details

The ITEM= statement defines the fields within the record. The item name is the name that you assign to the field in an IMS database segment and which SAS/ACCESS software uses to generate a SAS variable name. This name can be a maximum of 32 characters. If any special characters or blanks are included in the name, enclose the entire name in single quotation marks. This is a required argument.

The generated SAS variable name will use the naming conventions specified by the VALIDVARNAME system option. For information about VALIDVARNAME, see the *SAS Language Reference: Dictionary*.

If you specified the AN= statement with a value of Y, you will not be able to change the SAS variable names when you create a view descriptor from this access descriptor after the access descriptor is created.

If you specified the UN= statement with a value of Y, the variable names will be unique. Any duplicate names will be resolved as follows: the name will be truncated to the legal length and a number appended to the end to identify it as unique. For example, two instances of CUSTOMER_ADDRESS would be changed to CUSTOMER_ADDRESS and CUSTOMER_ADDRESS0.

Sites commonly refer to undesired portions of the data buffer by using the FILLER notation in the ITEM= statement and by defining the DBC (DB Content) as $CHAR. See "Using Filler Notation in ITEM=" on page 139 for information.

Arguments

In the ITEM= statement, you must enter the item name, level number, and the DBFORMAT= argument. The other arguments define the item further and are not required. The following list explains each argument in the ITEM= statement:

LEVEL=
LV=

specifies the two-character numeric level of the IMS field. This level number is similar to the COBOL level number. To indicate that a field is in a group, the field's level number must be greater than the group's level number. This is a required argument.

DBFORMAT=
DBF=

specifies how the IMS field is stored in the database. See "IMS Data Types in SAS/ACCESS Descriptors" on page 22 for a table of recommended DB formats to use for COBOL and PL/I data types. This table also shows the SAS variable formats that the SAS/ACCESS interface to IMS generates for the DB Formats.

You must specify one of the following SAS informats in this argument. For character data, the SAS informats are as follows:

$*w*.	$HEX*w*.
$CHAR*w*.	$PHEX*w*.
$CHARZB*w*.	

For numeric data, the SAS informats are as follows:

w.d	ZDB*w.d*	RB*w.d*
F*w.d*	IB*w.d*	PD*w.d*
BZ*w.d*	PIB*w.d*	PK*w.d*
ZD*w.d*	HEX*w.*	

SASNAME=
SN=
  is supported for Version 6 compatibility only. It assigns a SAS variable name to
  the IMS field. When VALIDVARNAME=V6, the name assigned to this argument is
  also used as input to the subsetting WHERE statement.

FORMAT=
FMT=
  assigns a SAS format to the SAS variable. This is an optional argument.
     If you specified the AN= statement with a value of Y, SAS assigns default
  formats (based on the field's database format) to the variables when the access
  descriptor is created. If you want, you can enter formats using the FORMAT=
  argument in the ITEM= statement at that time. However, you will not be able to
  change these formats when you create a view descriptor from this access
  descriptor after the access descriptor is created.

SEARCH=
SE=
  specifies the search field name defined for the field in the DBMS DBD. If you want
  the IMS engine to create SSAs directly from a WHERE statement or command
  that references the named item, then you must assign search field names.
  Otherwise, the WHERE statement is passed to SAS, and all occurrences of the
  segments referenced in the view descriptor in the database are read and passed to
  SAS for further processing. See "Performance and Efficient View Descriptors" on
  page 124 for more information about SSAs and WHERE statements. This is an
  optional argument.

KEY=
K=
  indicates with a Y, N, or U whether this field is defined in the DBD as a sequence
  or key field and whether the key sequence field is unique. The default setting, N,
  indicates the field is not a key sequence field. You must assign one key sequence
  field per segment if you use the view descriptors created from this access descriptor
  to update the IMS database. Keys are recommended, but not required, for all
  segments except the lowest hierarchical level if the view descriptors are used only
  for data retrieval. When KEY=U, retrieval calls to IMS are reduced because the
  IMS engine knows that there is only one segment in the database for this key.

OCCURS=
O=
  indicates the number of times a repeating field occurs. This is an optional
  argument.

DBCONTENT=
DBC=
  indicates that the values for this field need special handling by the IMS engine.
  This is an optional argument. You can use this argument to specify a SAS format

that indicates the way date values are represented internally in the IMS database, or to indicate how a field is initialized or stored in the database. This is not the same as the value you entered in the DBFORMAT= argument.

For example, you would use the DBFORMAT= argument to specify that a date is stored as a packed decimal. You would then use the DBCONTENT= argument to indicate where the month, day, and year are stored in that packed decimal. The following are valid parameters for date values:

Table 6.2 Valid Parameters for Date Values

YYMMDD6.	DDMMYY6.	JULIAN5.
YYMMDD8.	DDMMYY8.	JULIAN7.
MMDDYY6.	TFGY2KD4.*	
MMDDYY8.	TFGY2KN4.*	

* The TFGY2KD4. and TFGY2KN4. values indicate a 4-byte packed decimal value that is stored in the IMS database in the form 'CYYMMDDS'x, where C=century (0=1900, 1=2000), YY=year, MM=month, DD=day, and S=sign (C). TFGY2KD4. interprets the packed decimal value and converts it to a SAS date value, which is represented as the number of days since January 1, 1960. For example, '0990101C'x is interpreted as January 01, 1999, and is converted to the value 14245. You can then use the FORMAT= statement to apply a SAS format to the value. TFGY2KN4. interprets the packed decimal value as an 8-byte number and converts it to a numeric value. For example, '1990101C'x is interpreted as January 01, 2099, and is stored as 20990101. When the database is updated, the SAS values are converted back to the packed decimal format. When TFGY2KD4. or TFGY2KN4. are entered for DBC=, a DBFORMAT= of PD4. or PD4.0. must also be specified or SAS issues an error message.

The following are valid parameters for special formats values that indicate how a field is initialized:

B   when values are blanks for zero.

H   for high values.

L   for low values.
These special formats affect how SAS displays and updates the fields in the database. Use special format B to indicate to the IMS engine that a numeric variable has blanks when its value is zero. Use the special codes H and L to indicate that a variable is initialized to high or low values, respectively. For example, if you specify L for a variable, SAS displays a missing value when it finds low values (hexadecimal zeroes) in the variable. If you update that variable with a missing value, the IMS engine writes low values to the variable in the database. If you specify H for a variable, SAS displays a missing value when it finds high values (hexadecimal Fs) in the variable. If you update that variable with a missing value, the IMS engine writes high values to the variable in the database.

You can also use the special formats values when a date is initialized in a special way. For example, if you have a date initialized to low values, enter, enclosed in single quotation marks, the date format followed by a slash (/) and an initialization code. For example, for an eight-digit date in the MMDDYY8. form initialized to low values, you would enter the following value for the DBCONTENT argument:

```
'MMDDYY8./L'
```

Do not specify a DBCONTENT for records and groups.

LIST Statement

Lists all or selected items in the descriptor and information about the items

Optional statement

Applies to: access descriptor or view descriptor

Syntax

LIST <ALL | VIEW | *index-number* | *item-name*> <*blanks* | DB | DESC>;

Details

The LIST statement lists all or selected items in the descriptor and information about the items.

Arguments

The LIST statement consists of two sets of arguments. Select one argument from the first set to select the items to be displayed, and select one argument from the second set to specify the type of information to be displayed about the selected items.

The first set includes the following arguments:

ALL

lists all the items in the access descriptor that are available for selection. If an item is dropped, **NON-DISPLAY** is displayed next to the item's description when listing an access descriptor. When listing a view descriptor, dropped items are not displayed.

VIEW

lists all the items in the access descriptor that are selected for the view descriptor.

index-number

specifies the index number that corresponds to the IMS item for which you want to display the current status. The index number represents the item's place in the descriptor.

item-name

specifies the name of an IMS item for which you want to display the current status.

The second set includes the following arguments:

blanks	lists the SAS information, including the DB Format and SAS format information, for the specified items. To use this argument, include only the ALL, VIEW, *item-name*, or *index-number* argument from the first set to specify the items.
DB	lists the database information, including the DB Content, segment name, search field, segment length, key field, and occur field information, for the specified items. Use the ALL, VIEW, *item-name* or *index-number* argument before this argument to specify which items to list.
DESC	lists both SAS and database information for the specified items. Use the ALL, VIEW, *item-name* or *index-number* argument before this argument to specify which items to list.

Note: The LIST statement output is written to the SAS log. △

QUIT Statement

Terminates the procedure without any further descriptor creation

Optional statement

Syntax

QUIT | **EXIT**;

Details

The QUIT statement terminates the procedure without any further descriptor creation. EXIT is an alias for the QUIT statement.

RECORD= Statement

Defines an IMS segment

Required statement

Applies to: access descriptor

Syntax

RECORD=*record-name* SEGMENT=*segment-name* SEGLNG=*segment-length*;

Details

The RECORD= statement defines an IMS segment. A value of 01 is automatically assigned as the level number of a record, so the RECORD= statement does not include a level number argument. You should begin your database definition with a RECORD= statement immediately after the DATABASE= statement.

Arguments

The following list explains each of the arguments that can appear in a RECORD= statement.

RECORD= | RE=
    specifies an arbitrary name for the segment. A record name can be a maximum of 32 characters. If special characters or blanks are included in the name, enclose the entire name in single quotation marks. This is a required argument.

SEGMENT= | SG=
> specifies the name of the segment as defined in the DBD. A segment name can be a maximum of eight characters. If your database is a GSAM database, enter `GSAM` as the segment name. This is a required argument.

SEGLNG= |SL=
> specifies the segment length as defined in the DBD. This is a required argument. See "Handling Segments of Varying Length" on page 137 for more information.

RENAME Statement

Enters or modifies the SAS name for an item

Optional statement

Applies to: access descriptor or view descriptor

Syntax

RENAME *item-name*|*index-number* <=> *SAS-name*
  <... *item-name-n*|*index-number-n* <=> *SAS-name-n*>;

Details

The RENAME statement enters or modifies the SAS variable name for an item. If you are creating a view descriptor from an existing access descriptor with an ASSIGN= value of Y, you cannot use the RENAME= statement. You can rename as many items as necessary using one RENAME= statement.

Note: If the VALIDVARNAME system option is set to V6, this statement affects the SAS name parameter; if VALIDVARNAME is set to V7 or one of the other values, it affects the item name. △

Arguments

The following list explains the arguments that appear in the RENAME= statement:

item-name
> is the name of the IMS item that you want to rename.

index-number
> is the index number of the IMS item that you want to rename. The index number represents the item's place in the descriptor. You can get the index number of an item using the LIST statement described earlier in this section.

SAS-name
> is the new SAS variable name that you want to assign to the specified item.

REPLACE Statement

Modifies record, group, and item definitions in an existing access descriptor

Optional statement

Applies to: access descriptor

Interacts with: UPDATE statement

Syntax

REPLACE | **REPL** *index-number*

  <GROUP= *new-group-name* ITEM=*new-item-name*
      LEVEL=*level-number*
      DBFORMAT=*database-format*
      FORMAT=*SAS-format* SEARCH=*search-name*
      KEY=Y|N|U DBCONTENT=*database-content*>;

   |

  <RECORD=*new-record-name*
      SEGMENT= *segment-name*
      SEGLNG= *segment-length*>;

REPLACE | **REPL** *item-name*

  <LEVEL=*level-number* DBFORMAT=*database-format*
      FORMAT=*SAS-format* SEARCH=*search-name*
      KEY=Y|N|U DBCONTENT=*database-content*>;

   |

  <SEGMENT=*segment-name* SEGLNG=*segment-length*>;

Details

The REPLACE statement replaces or modifies existing records, groups, and items in existing access descriptors. Any item that can be entered on RECORD, GROUP=, and ITEM= statements can be modified, except the OCCURS option.

   Unlike the INSERT and DELETE statements, each data item to be modified needs a separate REPLACE statement, although any number of REPLACE statements can occur in any order with INSERT and DELETE statements within an UPDATE statement

Arguments

The only required item on the REPLACE statement is the index number, name, or quoted string used to identify it. However, the optional arguments are recommended for data definition. Except for the following optional arguments, the arguments follow the same editing rules as they would in create mode or in an update insert situation.

  □ KEY=N will remove an item as a designated key field.

  □ Specifying blanks on a SEARCH or DBCONTENT parameter removes their value, effectively dropping the parameters.

  □ The FORMAT parameter currently cannot be reset to its default value.

Examples

The following are examples of replacement statements:

```
replace shipped dbc=mmddyy6.; /* modifies dbcontent */

replace 5 se=' '    /* drops search field parameter */

replace 'old-record-name' record='new-record-name';
   sg='new-ims-segname';       /* replaces record    */

replace 2 item='cust-item';  /* renames item          */
```

RESET Statement

Resets specified or all items to their default settings

Optional statement

Applies to: access descriptor or view descriptor

Syntax

RESET ALL | *item-name* | *index-number*
   <... *item-name-n* | *index-number-n*>;

Details

The RESET statement resets specified or all items to their default settings. You can reset as many items as necessary using one RESET statement or the ALL option to reset all the items. The RESET statement has different effects on access and view descriptors.

Access descriptors   In access descriptors, the default setting for a SAS variable name is a blank unless you included the AN= statement. If you used the AN= statement, the names are reset to those generated. The default setting for SAS formats in access descriptors is determined by the DB Formats of the items. Any dropped items will be included again.

View descriptors   In view descriptors, the RESET statement deselects items and resets the SAS name and format values to those defined in the access descriptor on which the view descriptor is based. The SAS names and formats are unaffected by the RESET statement if you specified the AN= statement with a value of Y when you created the access descriptor on which this view descriptor is based.

Arguments

The following list explains the arguments that appear in the RESET statement:

ALL
   resets all the items defined in the access descriptor to their default setting. For a view descriptor, the ALL option resets only the items that are selected.

item-name
   specifies the name of the item that you want to reset. If you specify a record or group name, all the items in that record or group are reset.

index-number
> specifies the index number of the item that you want to reset. The index number represents the item's place in the access descriptor. You can get the index number of an item using the LIST statement described earlier in this section. If you specify a record or group index number, all the items in that record or group are reset.

SELECT Statement

Selects the items in the access descriptor that are to be included in the view descriptor

Optional statement

Applies to: view descriptor

Syntax

SELECT ALL | *item-name* | *index-number* <... *item-name-n*|*index-number-n*>;

Details

The SELECT statement selects the items in the access descriptor that are to be included in the view descriptor. Use the SELECT statement only when you are defining view descriptors. You can select as many items as necessary using one SELECT statement.

Arguments

The following list explains the arguments that appear in the SELECT statement:

ALL
> includes in the view descriptor all of the items that are defined in the access descriptor that were not dropped.
>
> ***CAUTION:***
> > **If the access descriptor contains segments representing more than one path, using ALL will create an invalid view descriptor.** △

item-name
> specifies the name of the item you want to select to be included in the view descriptor. If you specify a record or group name, all the items in that record or group are selected.

index-number
> specifies the index number of the item you want to select. The index number represents the item's place in the access descriptor. You can get the index number of an item using the LIST statement described earlier in this section. If you specify a record or group index number, all the items in that record or group are selected.

SUBSET Statement

Adds or modifies selection criteria defined for a view descriptor

Optional statement

Applies to: view descriptor

Syntax

SUBSET <*selection-criteria*>;

Details

The SUBSET statement specifies the selection criteria for the view descriptor. If you do not use the SUBSET statement, the view will include all occurrences of the segments included in the view descriptor.

Arguments

The *selection-criteria* argument can be new or modified selection criteria that you want to define for the view descriptor. Only a WHERE statement can be used with the SUBSET statement.

Use SAS variable names in the SAS WHERE statement to specify selection criteria. Any variable specified in the WHERE statement must also be selected in the view descriptor. If your statement includes a date or time representation, use the SAS date or time constant representation, such as '01JAN91'D.

To improve performance, use WHERE statements from which the IMS engine can generate SSAs. For more information about creating efficient view descriptors, see "Performance and Efficient View Descriptors" on page 124. For more information about the WHERE statement and the expressions it allows, see *SAS Language Reference: Dictionary*.

You can delete the current selection criteria by issuing the SUBSET statement without an argument.

UNIQUE = Statement

Generates unique SAS names based on item names

Optional statement

Applies to: view descriptor

Syntax

UNIQUE | **UN** = Y | N;

Details

The UNIQUE= statement specifies whether unique SAS variable names should be generated for items. The UNIQUE= statement can be used only when creating a view descriptor.

The default value, N, enables you to enter duplicate SAS variable names. You must resolve these duplicate names before you create view descriptors based on the access descriptor.

If you specify a value of Y and duplicate SAS variable names exist, numbers are appended to any SAS names that are duplicated as the result of truncation. For example, if you enter a value of Y for the UNIQUE= statement, two instances of the item ADDRESS would be changed to ADDRESS and ADDRESS0.

Note: If you specified a value of Y for the ASSIGN= statement when you created the access descriptor on which this view descriptor is based, you cannot specify a UNIQUE= statement. △

UN= is a valid alias for this option.

UPDATE Statement

Updates a SAS/ACCESS descriptor file.

Optional statement

Applies to: access descriptor or view descriptor

Syntax

UPDATE *libref.member.*ACCESS | VIEW;

Details

The UPDATE statement identifies an existing access descriptor or view descriptor that you want to change. The descriptor can exist in a temporary (WORK) or permanent SAS data library. If the descriptor has been protected with a SAS password that prohibits editing of the access or view descriptor, then the password must be specified on the UPDATE statement.

To update a descriptor, use its three-level name. The first level identifies the libref of the data library where you stored the descriptor. The second level is the descriptor's member name. The third level is the type of SAS file: ACCESS or VIEW. For a view descriptor, you can optionally specify the PSBNAME and PCBINDEX arguments.

You can use the UPDATE statement as many times as necessary in one procedure. Use these guidelines to write the UPDATE statement:

□ Like the CREATE statement, the UPDATE statement should immediately follow PROC ACCESS and precede any database definition and editing statements. Also, all database definition statements should precede any editing statements.

□ Within the database definition group, the DELETE, INSERT, and REPLACE statements can be specified in any order and can occur multiple times with an UPDATE sequence. The order has no effect on processing.

□ When using index numbers, the numbers specified with the UPDATE statement refer to the original pre-update order. Index numbers used with editing statements always apply to the post-update, "ready to rewrite" order.

□ Use the LIST statement after the UPDATE statement and avoid using intermediate LIST statements, particularly in batch mode. The LIST statement

forces a reorganization of the in-memory layout of the access or view descriptor. Intermediate list statements change the index numbering at each invocation and can cause an error.

☐ Do not attempt to create a view descriptor after you have updated a view descriptor in the same procedure execution. You can create a view descriptor after updating or creating an access descriptor or after creating a view descriptor.

The following examples edit the access descriptor IMSLIB.CUSTS. Despite the order of the INSERT, DELETE, and REPLACE statements in the update sequence, the examples produce identical results.

```
/* ----example 1------ */
proc access dbms=ims;
  update imslib.custs.access;
   insert address;
      item=address2      lv=3 dbf=$12     se=custadd2;
   delete contact;
   repl 23 se=custphon;
   ins 23;
      item=newitem      lv=3 dbf=$30     se=custlsti;
run;


/* ---example 2--- */
proc access dbms=ims;
  update imslib.custs.access;
   delete contact;
   repl 23 se=custphon;
   ins 23;
      item=newitem      lv=3 dbf=$30     se=custlsti;
   insert address;
      item=address2      lv=3 dbf=$12     se=custadd2;
run;
```

The following example shows how index numbers are interpreted by different parts of an UPDATE statement. In the example, the DELETE statement processes the third item in the original descriptor. The DROP statement, however, processes the fourth item in the post-update order, which in this case would have been the fifth item in the original sequence.

```
proc access dbms=ims;
  update imslib.custs.access;
    delete 3;  /* pre-update item 3 */
    drop 4;    /* post-update item 4 */
  list all;
run;
```

Pre-update and post-update listings are shown below.

```
/* ---prior to UPDATE --- */
IMS Database: CUSTOMER
Function: Create Descriptors-access: CUSTS1  view:
   L# Item Name                 DBFormat    Format
1  01 CUSTOMER                  *RECORD*    *RECORD*
2  02 CUSTOMER-INFO             *GROUP*     *GROUP*
3  03 CUSTOMER-CODE             $8.         $8.
4  03 STATE                     $2.         $2.
5  03 ZIP                       10.0        12.0
```

```
6  03 COUNTRY                      $20.       $20.


/* ---after UPDATE --- */
IMS Database: CUSTOMER
Function: Create Descriptors-access: CUSTS1  view:
   L# Item Name                 DBFormat    Format
1  01 CUSTOMER                   *RECORD*    *RECORD*
2  02 CUSTOMER-INFO              *GROUP*     *GROUP*
3  03 STATE                      $2.         $2.
4  03 ZIP   *NON-DISPLAY*        10.0        12.0
5  03 COUNTRY                    $20.        $20.
```

Tools for Creating IMS Access Descriptors

The SAS/ACCESS interface to IMS is different from other SAS/ACCESS interfaces in that it requires you to define the database in your access descriptor. Other SAS/ACCESS interfaces are able to query a data dictionary or another information repository to acquire detailed information about the database object that is being accessed.

Defining access descriptors for IMS databases can be time consuming because the data has to be entered manually. To automate this process, especially in cases where many access descriptors must be defined, there are several tools available for your use.

COB2SAS Tool

The COB2SAS tool uses the COB2SAS utility to process COBOL copybook database definitions and to store them in a permanent SAS data file. This data file is then processed by a DATA step program that is supplied in the installed *prefix*.SAMPLE PDS, called IMSS2A. The IMSS2A program processes the observations in the data file and generates most of the syntax required by the PROC ACCESS procedure statements that create an access descriptor for the database.

The generated statements are written to a host file (physical sequential or PDS member) where they can be edited. The statements written to the host file require some editing because the copybook file does not contain all the information that is necessary to create the access descriptor. You need to add DBD-specific information such as segment lengths, search and sequence field names, DBD name, DBTYPE, and segment names, in order to complete the code. You can then either submit the generated statements with JCL in a batch execution, or submit them from the SAS Program Editor window.

The COB2SAS tool is available from SAS free of charge for download from the World Wide Web, from an FTP site, or in the form of a mailer tape. This tool was originally designed to aid in converting COBOL file copybooks to INPUT statements for SAS DATA steps. For access descriptor creation, it is not necessary to complete all of the steps outlined in the COB2SAS usage instructions. Typically, after the copybook is processed, the results are stored in a temporary SAS file, which is then used to generate the INPUT statement. For IMS access descriptor creation, only the steps up to and including creation of the SAS file (dictionary file) are necessary. A modification is made to make the dictionary file permanent, and from there the IMSS2A program is used to complete the process.

Note that only steps R2COB1-R2COB5 are needed to create the dictionary file. Member R2MVS is the file to edit to make the dictionary a permanent file. R2MVS is also the main program that drives all of the other steps. It is well documented, and comments provide information about what each step does.

For more information about using the COB2SAS tool and about the IMSS2A sample program, look in the sample PDS for OS/390.

SAS Macro and DATA Step Code

The second tool was donated by a SAS user.* The tool consists of SAS macro and DATA step code that processes the database DBD directly. The benefit of this tool is that the file of generated PROC ACCESS code does not need further editing before being submitted for execution. This tool is available in the sample PDS for OS/390.

Performance and Efficient View Descriptors

When you create and use view descriptors, follow these guidelines to minimize the use of IMS and OS/390 system resources and to reduce the time IMS takes to access data.

General Information

Select only the items your program needs. Selecting unnecessary fields adds extra processing time.

Sorting data can be resource-intensive, even if it is done using the SORT procedure. You should sort data only when sorted data is needed for your program. Note that IMS does not support the ORDER BY clause or a BY statement in an application, such as **PROC PRINT ... BY** *variable*...;. If you have an IMS database that does not have an index and you want to use a SAS procedure that requires the data to be sorted, you must first extract the data to sort it. If you have an IMS database that does have an index and you want to use a BY variable other than an index key, you must also extract the data to sort it before executing the SAS procedure.

Where possible, specify selection criteria that can be converted into SSAs to subset the amount of data IMS returns to SAS.

Extracting Data Using a View

If a view descriptor describes a large IMS database and you will use the temporary or permanent view descriptor many times, it might be more efficient to extract the data and place it in a SAS data file. Under the following circumstances you should probably extract data:

- □ If you plan to use the same IMS data in several procedures, you might improve performance by extracting it. Placing the data into a SAS data file requires disk space to store the data and I/O to write the data. However, SAS data files are organized to provide optimal I/O performance with PROC and DATA steps. Programs using SAS data files often use less CPU time than programs that directly read IMS data.

- □ If you plan to read a large amount of data from a large IMS database and the database is being shared by several users, your direct reading of the data could adversely affect all users' response time. Extracting data can improve response time.

---

* Bruce Babbitt of New England Power Service Company

☐ If you think directly reading this data would present a security risk, you might want to extract the data and not distribute information about either the access descriptor or view descriptor.

Deciding How to Subset Your Data

There are many reasons why you might want to subset or filter the data that is being returned from a database path that is defined by a view descriptor. The main benefit is performance. Retrieving a portion of the data in the database path is more efficient than retrieving all of the data in the path. Another reason is to enforce security measures, such as restricting users of view descriptors to certain subsets of data.

Once you determine that your application can benefit from using a subset of data, there are several ways that you can subset data in SAS. Use the following guidelines to determine when to use a view descriptor WHERE expression, an application WHERE expression, or a DATA step subsetting IF statement, and when to use a combination of the methods.

Note: Regardless of the method that you choose, for performance reasons you should always attempt to choose selection criteria that can be converted by the engine into SSAs. If the engine cannot build SSAs for your data request, then a sequential access method is used to retrieve all path data that is defined by the view descriptor. △

View Descriptor WHERE Expression

Include a WHERE expression in your view descriptor by using a SUBSET statement when you want to do the following tasks:

☐ have selection criteria that you want to always apply, regardless of the application that references the view descriptor.

☐ restrict access to data in a way that the selection criteria cannot be viewed, modified, or deleted.

Selection criteria stored in a view descriptor can be protected with a password as well as with operating system security. If an application specifies additional subset criteria, it is combined with the view descriptor selection criteria and treated as an AND search argument.

Application WHERE Expression

Use an application WHERE expression (SAS WHERE statement, clause, or data set option) when the guidelines specified in the previous section do not apply and you meet the following criteria:

☐ you want to use the same view descriptor for various tasks (includes DATA steps, procedures, and SCL), where each requires a different subset of data

☐ you need to generate dynamic selection criteria for the data that is defined by the view descriptor.

For a more detailed description of how the WHERE expressions work, see "WHERE Statement Processing" on page 145.

DATA Step IF Statement

Use a subsetting IF statement in a DATA step execution when you meet the following criteria:

☐ you need to impose selection criteria that would result in a sequential retrieval of the data that is defined by the view descriptor. This type of criteria does not meet SSA eligibility requirements.

The IMS engine generates SSAs only when all of the conditions in a WHERE expression meet eligibility requirements. The DATA step IF statement enables you to perform filtering that does not meet SSA eligibility requirements, while using a view descriptor WHERE expression or application WHERE expression to obtain the performance benefits from SSAs.

Combination of Methods

There are some comparison operators in SAS that cannot be incorporated into SSAs for DL/I function calls and that cannot be used with the DATA step IF statement. In these cases, you will have to evaluate the impact of a sequential retrieval to see if that method is acceptable. If it is not, then you can extract a subset of view descriptor data into a SAS data set (or define a DATA step view) using eligible selection criteria, then subset the data set using an application task to achieve the desired performance gains.

If needed, you can mix all of the filtering methods. For example,

```
data work.subset;
  set vlib.imsview; /*View can contain subset criteria*/

  where (additional eligible conditions for IMS SSAs);
  if (ineligible criteria that would not generate SSAs);
run;
```

For all methods, it is possible that a change in criteria can cause an application that once produced SSAs to no longer produce them and resort to using a sequential access method. You can prevent this from happening with the SAS system option IMSWHST=Y. IMSWHST= is an invocation option that can be placed in the restricted options table so that it cannot be changed or overridden. Should the engine detect that no SSAs can be generated when this option is in effect, it will issue a message to the SAS log and terminate the executing task.

Writing Efficient WHERE Statements

Specifying a WHERE statement from which the IMS engine can generate SSAs improves performance. The IMS engine returns to SAS only those database segments that meet your selection criteria. If the IMS engine cannot generate SSAs, all segment occurrences for each IMS record (as defined by the path of segments in the view descriptor) are returned to SAS for further processing.

To determine whether SSAs are being generated by your WHERE statement, set the option IMSDEBUG=Y or set the number of calls for which you want debugging information.

To ensure that your WHERE statements generate SSAs, do the following:

☐ When creating descriptors, specify a search field name for all variables you plan to include in your application's WHERE statements, when possible.

☐ Use one of the eight operators supported by IMS in your WHERE statements. The eight operators supported by IMS are listed in the following table, along with their alternate forms.

Table 6.3   IMS Supported Operators

Operator	Alternate Form
=	EQ
>	GT

Operator	Alternate Form
<	LT
>=	=> or GE
<=	=< or LE
¬=	=¬ or NE
&	* or AND (dependent AND)
|	+ or OR (logical OR)

* Pad the =, >, and < operators with blanks on the right or left.

The ability of the IMS engine to generate SSAs also depends on the database type and on the operators that you use in your WHERE expression.

- □ For GSAM databases, no SSAs can be generated.
- □ For other database types, the following rules apply:
  - □ SSAs are generated only for WHERE expressions that involve a variable, an operator, and a literal value. Multiple expressions that use Boolean operators are also allowed. For example:

    ```
    where partnum > 1000
    where partnum > 1000 and
                    orddate = '31JAN94'd
    ```

  - □ The following operators generate SSAs: = (EQ), > (GT), < (LT), >= (GE), <= (LE), IN, BETWEEN, IS NULL, and IS MISSING. For HDAM databases, only the equals (=), IS MISSING, and IN operators generate SSAs.
  - □ Compound expressions generate SSAs, except when the expressions are joined by OR and the fields involved are in different segments.

For a more detailed description of how WHERE statements work, see "WHERE Statement Processing" on page 145.

Identifying Inefficient SAS WHERE Conditions

When your view descriptor uses WHERE clauses that have multiple values for a search field, and specifies a path that does not originate from the root segment in the IMS database, it forces the IMS engine to reposition itself to the beginning of the IMS database for each value.

In this example, the WHERE statement tries to find two checking account records in the ACCTDBD database.

```
where chckacct = '345620145345'
   or chckacct = '345620134663';
```

Because the CUSTOMER segment is the root segment and the CHCKACCT segment is a child of CUSTOMER, the IMS engine must issue a GU call for each checking account number that it wants to find. It does this in order to reposition itself at the start of the database. If it used GN calls, it might pass by one of the records because they are not in sequential order.

Specifying multiple values for a search field in a WHERE statement for HDAM IMS databases permits the IMS engine to create a WHERE key list. The IMS engine issues calls that use, at a minimum, the first segment level SSA with a WHERE key list value. When no more data is retrieved from the IMS database for a WHERE key list value, a

GU call is used to reposition to the beginning of the database and the next WHERE key list value is used. Processing stops when all WHERE key list values have been used.

The following conditions do not allow the IMS engine to generate SSAs. They cause all data from the IMS database as defined by the view descriptor to be returned to SAS for further processing:

- HDAM WHERE statements that use a WHERE key list and an OR operator with another search field or key list in the first segment level of the view descriptor, for example,

```
where custcode in ('24589689' '29834248')
    | state in ('CA' 'VA');
```

- an OR between two segment levels.

Identifying SAS WHERE Conditions That Are Not Acceptable to IMS

The following examples are SAS WHERE conditions that are passed to SAS for further processing.

- arithmetic expressions, for example:

```
where c1=c4*3
where c4−c5
```

- expressions in which a variable or combination of variables assumes a value of 1 or 0 to signify true or false, for example:

```
where c1
where (c1=c2)*20
```

- concatenation of character variables, for example, **where c2=D2||D3**.
- LIKE, BETWEEN, CONTAINS, SOUNDS LIKE operators, for example:

```
where lastname=*'SMITH'
where lastname like 'D_A%'
```

- truncated comparison, for example, **where c1=:abc**.
- DATETIME and TIME formats, for example:

```
where ctime= '12:00't
where ctime= '01jan60:12:00'dt
```

- comparisons using operators other than equivalence (=) for character variables, for example:

```
where name>'A'
where ssn<='251-09-7384'
```

- comparisons using operators other than equivalence (=) for date variables not in the YYMMDD format, for example, **where stmtdate>'01JAN01'D**. STMTDATE has a DB Content of MMDDYY6.
- references to missing values. This includes the period (.) for numeric variables, and the IS MISSING and IS NULL operators.

```
where stmtdate = .(numeric)

where name = (character)
```

☐ OR requests for conditions in two hierarchical levels of the database, for example,
   where name='Smith' or stmtamt>0. In this example, the NAME field is in the
   root segment, and the STMTAMT field is in a child segment.

☐ any WHERE statement for a GSAM database, for example, **where var1<200**.

☐ Any reference to a variable that does not have a SEARCH or SEQ field assigned to
   it in the access descriptor.
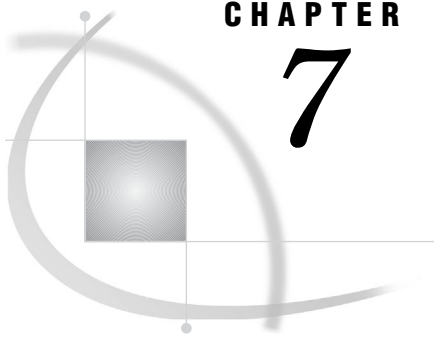
Advanced User Topics for the SAS/ACCESS Interface View Engine for IMS

Introduction to Advanced Topics for the Interface View Engine

This section includes some considerations for administering the SAS/ACCESS interface view engine for IMS (hereafter, referred to as the *IMS engine*). It provides additional technical detail on how the engine interface and engine calls work.

Changing an IMS Database and the Effects on Descriptors

Changes to an IMS database can affect descriptor files. You must modify or re-create the descriptors if changes to the IMS database invalidate them. You use the ACCESS procedure to edit the affected access descriptors and view descriptors, or to create new descriptors.

If a view descriptor differs from the access descriptor, you receive a message. Re-create or edit the view descriptor as required. If you do not change your descriptor files, IMS might return incorrect data to you. If the changes to the database involve numeric data, the procedure that uses the view descriptor could terminate abnormally. See "UPDATE Statement" on page 121 for information about editing descriptors.

Changing an item name has *no* effect on existing view descriptors. However, before you make other changes to IMS databases, consider the guidelines described in the next section.

Changes That Cause Existing View Descriptors to Fail

The following changes to an IMS database cause existing view descriptors to fail:

☐ inserting or deleting segments in the middle of the hierarchy if you are updating the database

☐ inserting or deleting a level in multiple occurring fields

☐ changing the attributes of a field

☐ deleting fields that are referenced in a view descriptor

☐ inserting a field in the middle of a segment

☐ adding fields to the end of database segments because longer segments might not be reflected in the RECORD statement's SEGLNG= argument.

Understanding Character Set Encoding

IMS does not use character sets or code pages and does not transcode data, so the interpretation of the data is done by SAS. Therefore, the IMS engine must transcode all character data going into an IMS database and all character data returned from an IMS database.

The default encoding behavior is as follows:

☐ for output processing to a new IMS database (did not previously exist), the data is written to the new database using the current SAS session encoding.

☐ for output processing to an existing IMS database, the new data inherits the encoding of the existing data in the database.

☐ for input (read) processing, if the SAS session encoding and the encoding on the IMS database are incompatible, the data is transcoded to the session encoding. If the database does not have an encoding, SAS transcodes the data only if the host platform is different.

The SAS/ACCESS interface to IMS supports the ENCODING= data set option in order to override the encoding for processing a specific input or output file. For example, when you are reading an IMS database using an IMS view descriptor, the ENCODING= data set option enables you to specify an encoding that is different from the session encoding. The data is transcoded from the database encoding to the session encoding as the data is read from the IMS database.

```
proc print data=imsview (encoding=latin2);
 run;
```

Some of the reasons that you might want to override encoding behavior by using the ENCODING= data set option are as follows:

- □ to create output in an encoding that is different from the current session encoding or that is the encoding for an existing IMS database.
- □ to create output that contains mixed encodings.
- □ to request that no transcoding occurs.

For more information about the ENCODING= data set option, see the *SAS Language Reference: Dictionary*.

Ensuring IMS Data Security

This section describes how to prevent unauthorized access to the data in your IMS databases.

IMS Security

SAS preserves the data security that is provided by IMS and the operating system. The Database Administrator (DBA) has control over who has access to an IMS database. A user cannot use IMS facilities through the ACCESS procedure or the SAS/ACCESS interface view engine unless the PSB specified provides that user with the appropriate IMS authority. The PSB determines if a user can access an IMS database and, if so, the kind of access the user has to the database (Get, Insert, Replace, Delete, or All).

In addition to controlling access to a database, the PSB can also control access to specific segments and fields in the database. To control access to a specific database, the DBA can create several view descriptors that describe the same data in the database, and assign each view descriptor a different PSB. Each PSB should define a different type of access to the database. For example, one PSB would enable a user to insert data in the database and another PSB would enable a user only to read the data in that same database. This enables the DBA to provide each user with a PSB that defines the type of database access the DBA wants to allow that user to have. Each segment in a view descriptor must be specified in the PSB that is referenced in the view.

SAS Security

To secure data from accidental update or deletion, you can do the following on the SAS side of the interface:

- □ Set up all SAS/ACCESS access descriptors yourself, dropping items that contain sensitive data so they cannot be referenced in view descriptors. Give users either read-only or no access to the SAS data library where you store the access descriptors. Read-only access prevents users from editing access descriptors and allows them to see only the items selected for each view descriptor.
- □ Set the IMSDLUPD= or IMSBPUPD= SAS system options to N to disable all updates from SAS for a particular region type.
- □ Assign SAS passwords (READ, WRITE, ALTER, or PW) to a view descriptor, access descriptor, PROC SQL view, DATA step view, or data file.

  Using passwords adds an extra measure of security if you use view descriptors that include sensitive or confidential data in a shared environment (that is, where

SAS/SHARE software is in use). For more information about assigning passwords, see "SAS Passwords for SAS/ACCESS Descriptors" on page 97.

Maximizing IMS Performance

Among the factors that affect IMS performance is the size of the database that is accessed. If the database being accessed is very large, you should evaluate all SAS programs that you want to access the database directly. When evaluating the programs, ask the following questions:

- □ Does the program need all the items included in the view descriptor?
- □ Does the view descriptor's WHERE statement retrieve only those records or segments that are needed for subsequent analysis?
- □ Does your WHERE statement directly generate SSAs so that only a subset of the data is passed to SAS for processing? To determine whether a WHERE statement is generating SSAs, set the SAS system option IMSDEBUG=Y or set the number of calls for which you want debugging information.

    For HDAM, avoid non-equality conditions in a WHERE statement. See "Identifying Inefficient SAS WHERE Conditions" on page 127 for more information.

- □ Can you use the DATA step's MODIFY statement to join view descriptors (where each view represents one path in the database) when conditions for a MODIFY statement's use apply?
- □ Is the data going to be used by more than one procedure? If so, consider requiring the data to be extracted and placed in a SAS data file rather than accessed directly by each procedure. (See the VIEWDESC= and OUT= options in "PROC ACCESS Statement Options" on page 97 for information about extracting IMS data.)

Understanding the IMS Interface

This section describes concepts that are exclusive to the SAS/ACCESS interface to the IMS engine. You must understand these concepts in order to successfully use the interface. This section describes the following concepts:

- □ the flattened file concept
- □ missing values
- □ special fields
- □ BY variables.

Understanding the Flattened File Concept

When the IMS engine creates SAS observations from a hierarchical database, it must flatten out the data. The *flattened file concept* means that SAS flattens the hierarchical levels and treats one path of data, including the root segment, parent segments, and child segments, as one SAS observation. If the root segment or any parent segment has children, the parent segment is repeated for each child segment's data. Therefore, each observation contains *all* the parent segments above the child segment.

For example, if you access the data in the database shown in Figure 7.1 on page 135, the IMS engine will return data from the segments in the following table as SAS observations. Therefore, the view descriptor would have to define four segment types.

Table 7.1 Flattened File Segments

Observation	Segments returned
1	1 2 3 4
2	1 2 3 5
3	1 2 3 6
4	1 2 3 7
5	1 8 9 .
6	1 8 10 .

Figure 7.1 Flattened File Concept
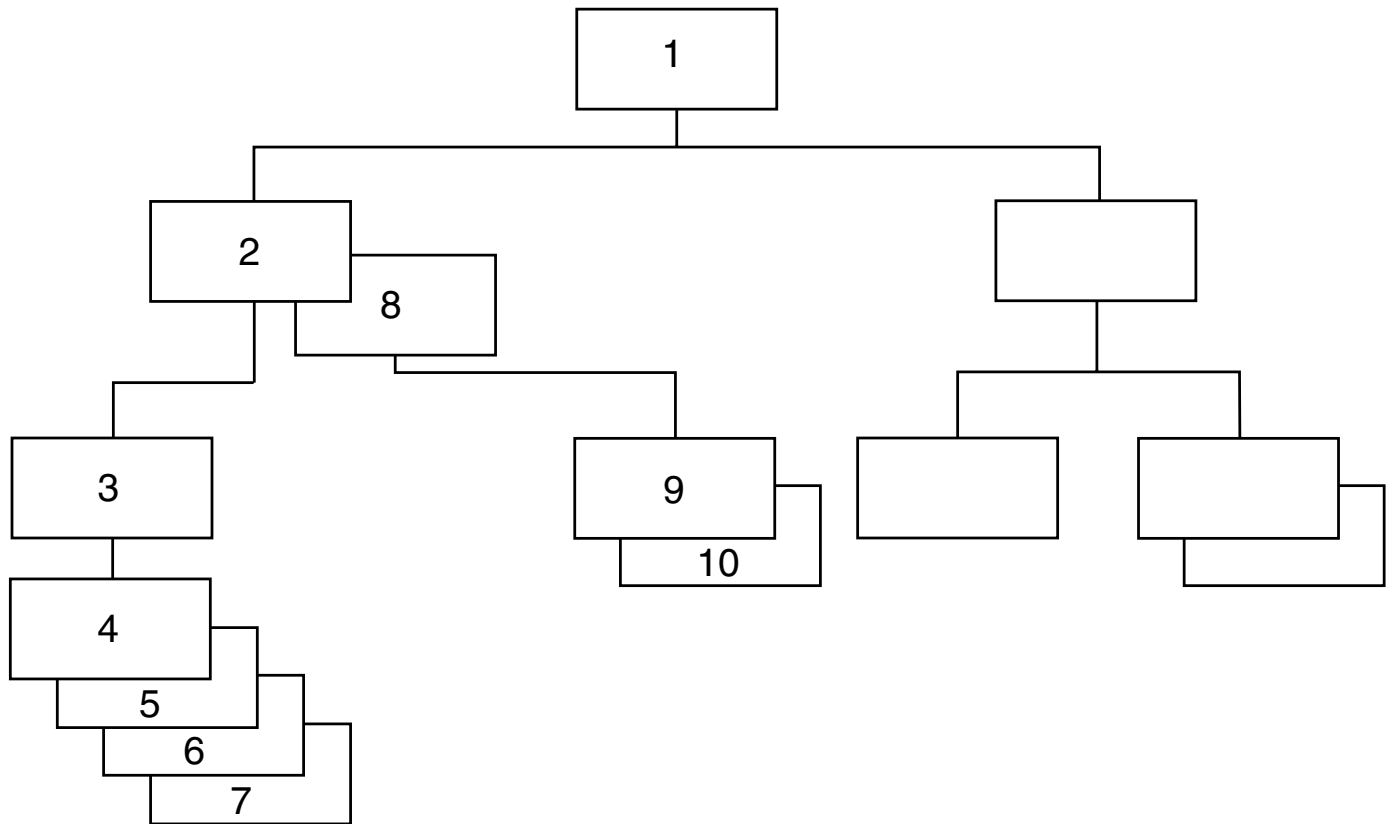
If you use the data from these observations in a SAS procedure, it appears that the data in segment 1 occur six times rather than only once. This can result in misleading statistics when you use such procedures as the MEANS procedure that involves any segment except the child segment in a database with more than two hierarchical levels. It can also be a problem in second-level data because root data repeats. To avoid misleading statistics that can result from flattened files, create view descriptors that describe data in only one hierarchical level. Or perform statistical operations using data from only the lowest level that is accessed by the view descriptor.

Using the *U Command Code

The IMS engine generates navigational SSAs to traverse and flatten the database hierarchy. Because sequential calls perform this task, the database's current position is an important issue. (See "Database Position" on page 28 for more information.)

Using a *U command code ensures the current database position on the proper parent segment as a DL/I call moves down the hierarchy to the next target segment (the segment named in the last SSA). *U on the immediate parent of the target means that even if the parent is unqualified, the position indicator remains there and does not move to a child (target segment) that belongs to a different parent occurrence.

For example, when DL/I processes a Get or ISRT call, it establishes a position on the segment occurrence that satisfies the call at each level in the path of the segment (target) that you are retrieving or inserting. A *U command code on an SSA in a Get or ISRT call tells DL/I not to move from the established database position at the level of the SSA when trying to satisfy the call. You use an ISRT call and I/O or TP PCBs to insert messages to the IMS/ESA control region message queues when a SAS program is executing in a BMP region. See "ISRT Calls to Message Queues" on page 214.

Handling Missing Values

This section describes how the SAS/ACCESS interface to IMS handles missing data values. It also describes how the DB Content field affects how data is displayed and stored in the database.

If you create a view descriptor to add an IMS-DL/I database segment and fields in that segment are not defined, the IMS engine writes low values to the database fields that are not included in the descriptor. The engine does so because it does not know that the fields exist.

If there are missing values in a SAS data set that you use to add or update an IMS database, the IMS engine writes zeroes to the database for numeric fields and blank spaces to the database for character fields unless you specify a special format (B, L, or H) for the DBCONTENT= argument of the ITEM= statement. DBCONTENT= affects how the engine updates the fields. (See "ITEM= Statement" on page 110 for more information about special formats.)

Conversely, if a field is defined with a DBCONTENT= value and the database retrieves that value (blanks, low values, or high values) in the field, then the IMS engine passes missing values to SAS. In addition, if a view descriptor describes more than one level in a database, and not all the levels exist for one database record, the IMS engine fills the missing segment occurrence with missing values in the SAS observation.

Handling Special Fields

Handling Fields That Occur Multiple Times

An item or a group in an IMS database segment can occur more than one time. For example, in the example database ACCTDBD, the two phone number fields, home phone and office phone, could be defined in your access descriptor as one field that occurs two times. To do this, specify OCCURS=2 in the ITEM= statement for the phone number field when you create the access descriptor. When you save the access descriptor, the descriptor is expanded to show fields for two phone numbers. When the IMS engine reads the database, it retrieves two phone numbers for each customer.

Fields that occur multiple times in the database can be nested only three levels deep, which creates a three-dimensional table. The following example shows the definition of a record with fields that occur multiple times, nested three levels deep:

```
01 Automatic Teller Record
   02 ATM Information
      03 ATM Location (occurs 20 times)

         04 Location
         04 ATM Transaction Information (occurs 7 times)

            05 Account Type
            05 Transaction Time
            05 Transaction (occurs 2 times)

               06 Transaction Type
               06 Transaction Amount
```

After you have saved an access descriptor, you cannot change the number in the OCCURS= argument. Instead, you have to delete an item and re-enter it with the correct number in OCCURS=.

Handling Redefined Fields

Redefined fields are fields that have been defined with more than one data type. For example, some records in a database might have character values stored in a certain field, and other records in the same database might have numeric values stored in that same field. You could handle this by defining the field as **$11.** in one access descriptor and **11.** in another access descriptor based on the same database. When you create view descriptors for the database, use a WHERE statement to retrieve only the appropriate values for the field. This can often be done by specifying a particular record type or other code in the WHERE statement.

Handling Segments of Varying Length

If you work with a segment that contains a field that varies in length, specify the maximum length of the varying field for SEGLNG= when you define the segment in the access descriptor. When IMS retrieves the entire segment, it fills in the varying portion with missing values if it did not retrieve any data for that portion of the segment.

Handling GROUP Keys in Descriptor Files

To support a definition of a GROUP field as a key and to be able to have access to the GROUP items, you need to define a dummy field for this key.

In IMS, GROUPs enable the same portion of data in a buffer to be assigned different logical names. For example, a field that begins at offset 1 for a length of 15 can be named FIELD1. Other fields can be defined within FIELD1, such as in FIELD2, FIELD3, and FIELD4 that begin at offsets 1, 6, and 11, respectively (where each has a length of 5).

Because no SAS variable name can be specified in the GROUP= statement, no single reference can be made to the group in the WHERE criteria. Therefore, even if a valid SEARCH or SEQ name exists for the GROUP in the DBD, the IMS engine cannot qualify calls that are based on the group itself.

A simple solution is to define the entire group as an item and to assign the SAS variable name and SEARCH name appropriately. Then you can specify a WHERE statement in your view descriptor or application and the IMS engine will build qualified

SSAs. A problem remains if the application wants access to the components of the GROUP. In this case, you must reference the view descriptor in a DATA step to SUBSTR out the parts and store them in separate SAS variables.

Using Dummy Fields for GROUP Keys

You can define a dummy field in the segment for a GROUP key in order to permit a WHERE clause reference for qualified SSAs and to access the composite fields. The GROUP statement defines the group but you can take it a step further. You add a dummy field to the end of the segment definition as an ITEM with a length that is equal to the entire GROUP and a SEARCH= value equal to the DBD SEARCH or SEQ field name from the DBD (the GROUP SEARCH= also has this value). The SEGLNG value is increased to allow for this field.

By using a dummy field for the GROUP, you can specify in your view descriptor a WHERE clause as follows:

```
WHERE sas-dummy-name EQ value
```

In this case, the IMS engine locates the dummy field in the view descriptor through the SAS variable name in the WHERE clause. It uses its SEARCH= value to qualify the SSA. When the data comes back to the buffer, the true data is in the GROUP portion of the segment definition and its component values are stored in the SAS variables that are associated with the items that are defined for the GROUP.

Also, by marking the GROUP itself as the key (with the KEY= argument), navigational SSAs that are generated by the IMS engine for sequential GN calls will refer to the correct buffer location for the data. The navigational SSAs will use the correct SEARCH= value in the SSA.

CAUTION:
You must never refer to the dummy field as the key (with KEY=) because doing so would force the IMS engine to use the dummy buffer location to qualify navigational SSAs for GN calls. This would cause problems. △

Below is an example of an access descriptor and a view descriptor based on the ACCTDBD. The GROUP key is on home phone, which has a dummy field (GROUP STUFF) defined for it.

```
proc access dbms=ims;
  create work.account.access;
    dbd=acctdbd dbtype=hdam;
    record='customer_record' sg=customer sl=225;
      item=soc_sec_number   lv=2 dbf=$11. key=u
                            se=ssnumber;
      item=customer_name    lv=2   dbf=$40.
                            se=custname;
      item=addr_line_1      lv=2   dbf=$30.
                            se=custadd1;
      item=addr_line_2      lv=2   dbf=$30.
                            se=custadd2;
      item=city             lv=2   dbf=$28.
                            se=custcity;
      item=state            lv=2   dbf=$2.
                            se=custstat;
      item=country          lv=2   dbf=$20.
                            se=custland;
      item=zip_code         lv=2   dbf=$10.
                            se=custzip;
```

```
      group=home_phone        lv=2
                              se=custhphn;
       item='area code'       lv=3    dbf=$3.
       item=filler1           lv=3    dbf=$1.
       item=phone_number      lv=3    dbf=$8.
       item=office_phone      lv=2    dbf=$12.
                              se=custophn;
       item='group stuff'     lv=2    dbf=$12.
                              se=custhphn;
   list all;

   create work.phone.view psbname=acctsam pcbindex=2;
     select soc_sec_number customer_name 'area code'
         'phone number' 'group stuff';
   list view;
 run;


 proc print data=work.phone;
   var soc_sec_number customer_name 'area code'
       'phone number';
   where 'group stuff' = '803-657-1346' or
         'group stuff' = '803-657-1687';
 run;
```

The following output shows the results.

Output 7.1   Results of a Dummy Field for a GROUP Key

```
                            The SAS System

 OBS    soc_sec_number            customer_name         'area code'  'phone number'

  1     436-42-6394               BOOKER, APRIL M.          803         657-1346
  2     178-42-6534               PATTILLO, RODRIGUES       803         657-1346
  3     434-62-1234               SUMMERS, MARY T.          803         657-1687
```

Using Filler Notation in ITEM=

It is important that access descriptor segment definitions not omit ITEM and GROUP references for fields that are embedded in the segment. Database segments might contain fields (contiguous or discontiguous) that applications might not need to access. In these cases, it is correct not to define them in SAS/ACCESS view descriptors. For performance reasons, it is recommended that applications not define them so that the IMS engine does not invoke conversion routines to convert data that will not be used.

Sites commonly refer to undesired portions of the data buffer by using the FILLER notation in the ITEM= statement, and by defining the DBC (DB Content) as $CHAR. If the undesired portion of the segment lays beyond all the desired segment fields, applications do not have to define these portions of the segment. However, you must be sure that the SEGLNG value for the segment is equal to the length of the entire segment and not just to the portion of the segment that they are interested in defining.

When the undesired fields are embedded between desired fields, you must use the FILLER notation or something similar (FILLER is a reserved word in COBOL but not in SAS). SAS uses relative offsets to locate defined fields in the buffer when converting data from the IMS buffer to the SAS program data vector (PDV). By using the field

lengths from the DBC, SAS determines the offset and length in the IMS buffer for the current field as needed to map to the PDV. If a field or series of fields is undesired, information must be supplied about placement and length so that SAS can move correctly to the next valid field to be mapped.

FILLER fields can be coded as DBC of $CHAR, which requires no conversion if selected for a view descriptor. In most cases FILLER fields are not selected. By preserving the relative offsets of fields within the buffer using FILLER definitions, the IMS engine can correctly map data that is requested by the application or view descriptor to the PDV.

Below is an example of a root segment for the ACCOUNT database with all of the fields defined from the DBD.

```
record='customer_record'    segment=customer
                            seglng=225;
  item=soc_sec_number       lv=2  dbf=$11.
                            search=ssnumber key=y;
  item=customer_name        lv=2   dbf=$40.
                            search=custname;
  item='address info'       lv=2;
  item=addr_line_1          lv=3   dbf=$30.;
  item=addr_line_2          lv=3   dbf=$30.;
  item=city                 lv=3   dbf=$28.;
  item=state                lv=3   dbf=$2. ;
  item=country              lv=3   dbf=$20.;
  item=zip_code             lv=3   dbf=$10.;
  item=home_phone           lv=2   dbf=$12.;
  item=office_phone      lv=2   dbf=$12.;
```

Assuming that none of your view descriptors would ever require phone information, you could code the following:

```
record='customer_record'    segment=customer
                            seglng=225;
  item=soc_sec_number       lv=2  dbf=$11.
                            search=ssnumber key=y;
  item=customer_name        lv=2   dbf=$40.
                            search=custname;
  item='address info'       lv=2;
  item=addr_line_1          lv=3   dbf=$30.;
  item=addr_line_2          lv=3   dbf=$30.;
  item=city                 lv=3   dbf=$28.;
  item=state                lv=3   dbf=$2. ;
  item=country              lv=3   dbf=$20.;
  item=zip_code             lv=3   dbf=$10.;
```

Note that the SEGLNG= value does not change even though two fields at the end are dropped.

By comparison, assume that the application needs everything except the address information:

```
record='customer_record'    segment=customer
                            seglng=225;
  item=soc_sec_number       lv=2   dbf=$11
                       .  search=ssnumber key=y;
  item=customer_name        lv=2   dbf=$40.
                            search=custname;
  item='filler'             lv=2   dbf=$char60.;
```

```
item=city              lv=3   dbf=$28.;
item=state             lv=3   dbf=$2. ;
item=country           lv=3   dbf=$20.;
item=zip_code          lv=3   dbf=$10.;
item=home_phone        lv=2   dbf=$12.;
item=office_phone      lv=2   dbf=$12.;
```

Here, the FILLER preserves 60 bytes so that view descriptors that reference fields past the filler can get data mapped correctly from the IMS buffer to the PDV variables based on the relative offset information. Once again, SEGLNG= does not change.

Using BY Variables

If you specify an IMS view descriptor as input to a SAS procedure that uses a BY variable, you must either

- □ create a SAS data file from the IMS data (that is, extract the data) and sort the data using that variable. You then specify the newly created data file in your procedure.
- □ reference a SAS variable associated with a database index in the BY statement; that is, the BY variable must be defined as the index key.

IMS Engine Calls to the Database

To create an access descriptor using the ACCESS procedure, you must first enter the database definition. IMS does not store descriptive information about databases in a dictionary or database. After you have created an access descriptor, you can select variables from one path of data when you create a view descriptor. The IMS engine is designed to get its information to build its own SSAs from the view descriptors and any supplied WHERE clause; these views are based on access descriptors that define the DL/I databases. The IMS engine uses the information stored in the view descriptor to generate DL/I calls and to format the results of those calls into SAS observations. By design, view descriptors cannot access IMS/ESA control region message queues. Therefore, the IMS engine interface is not able to access the message queues if it is executing in a BMP region.

Data Retrieval

The IMS engine sequentially processes database data in order to flatten IMS records when no WHERE criteria exist. All data in the path specified by the view descriptor is returned in the order in which it was stored in the database when you use unqualified Get-Next (GN) processing. Therefore, the IMS engine uses qualified segment search arguments (SSAs) to navigate the database path and maintain proper positioning, basing all qualified Get calls on the results of the previous call. You can use SAS WHERE statements to perform some level of direct access to a database.

You can see an example of this process by using the view descriptor VLIB.CHKDEB, which describes the CUSTOMER, CHCKACCT, and CHCKDEBT segments in the ACCTDBD database. First, the IMS engine issues an unqualified Get Unique (GU) call to position itself at the beginning of the database. If the CUSTOMER segment were the only segment in the ACCTDBD database, the IMS engine would then issue qualified Get Next (GN) calls for CUSTOMER until it reached the end of the database. However, because the ACCTDBD database is a multilevel database and the view descriptor defines more than the root segment, the processing is more difficult. To obtain the

dependent segment, the IMS engine must use the value returned in the I/O area for the field designated as the key in order to build a qualified SSA for the parent segment (in this case the root segment).

Next, the IMS engine issues a Get Next Within Parent (GNP) call by concatenating the qualified SSA for the root segment with an unqualified SSA for the next level down in the hierarchy. The engine then takes the value of the field designated as the key field of that segment (as defined originally in the access descriptor) from the I/O area to generate a qualified SSA for that level. The next database call is a GNP with the two qualified SSAs concatenated with an unqualified SSA for the next level down in the hierarchy. The engine continues to combine qualified SSAs with an unqualified SSA for the next lowest level down the hierarchy until the lowest level (as defined in the view descriptor) is retrieved, or until a status of GE is returned; GE indicates no segment occurrence.

The following figure shows the segments that are described by the view descriptor, VLIB.CHKDEB, and the order in which the segments are accessed by IMS. The calls that are generated by the IMS engine to navigate the database are also described. Note that one SAS observation is made up of one complete path of data. If there is no child segment, the IMS engine passes missing values in the fields for that segment to SAS.

Figure 7.2   ACCTDBD Segments That are Described by VLIB.CHKDEB
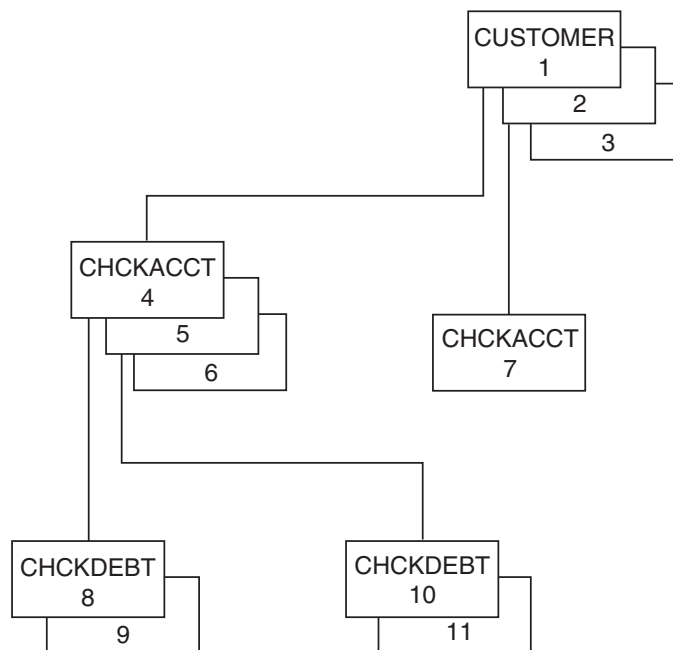
Shown below is the call output that is generated by the IMS engine when it navigates the database (based on the preceding figure). It is printed to the SAS log by using SAS IMSDEBUG=Y. It shows how the IMS engine uses the *U command code to maintain parentage in cases where no key field has been defined for one or more hierarchical levels in the view descriptor. See "Using the *U Command Code" on page 136 for more information.

```
GU                                      gets CUSTOMER 1
Status Code=

GNP
CUSTOMER*U-(SSNUMBEREQ667-73-8275)  gets CHCKACCT 4
```

```
CHCKACCT*--
Status Code=

GNP
CUSTOMER*U-(SSNUMBEREQ667-73-8275)  gets CHCKDEBT 8
CHCKACCT*U-(ACNUMBEREQ345620145345)
CHCKDEBT*--

Status Code=

GNP
CUSTOMER*U-(SSNUMBEREQ667-73-8275) gets CHCKDEBT 9
CHCKACCT*U-(ACNUMBEREQ345620145345)
CHCKDEBT*--

Status Code=

GNP
CUSTOMER*U-(SSNUMBEREQ667-73-8275)
CHCKACCT*U-(ACNUMBEREQ345620145345)
CHCKDEBT*--

Status Code=GE

GNP
CUSTOMER*U-(SSNUMBEREQ667-73-8275)  gets CHCKACCT 5
CHCKACCT*--
Status Code=

GNP
CUSTOMER*U-(SSNUMBEREQ667-73-8275) gets CHCKDEBT 10
CHCKACCT*U-(ACNUMBEREQ345620154633)
CHCKDEBT*--

Status Code=

GNP
CUSTOMER*U-(SSNUMBEREQ667-73-8275) gets CHCKDEBT 11
CHCKACCT*U-(ACNUMBEREQ345620154633)
CHCKDEBT*--

Status Code=

GNP
CUSTOMER*U-(SSNUMBEREQ667-73-8275)
CHCKACCT*U-(ACNUMBEREQ345620145345)
CHCKDEBT*--

Status Code=GE

GNP
CUSTOMER*U-(SSNUMBEREQ667-73-8275) gets CHCKACCT 6
CHCKACCT*--
Status Code=
```

```
GNP
CUSTOMER*U-(SSNUMBEREQ667-73-8275)
CHCKACCT*U-(ACNUMBEREQ345620180723)
CHCKDEBT*--

Status Code=GE

GNP
CUSTOMER*U-(SSNUMBEREQ667-73-8275)
CHCKACCT*--

Status Code=GE

GN
CUSTOMER*--                              gets CUSTOMER 2
Status Code=

GNP
CUSTOMER*U-(SSNUMBEREQ434-62-1234)   gets CHCKACCT 7
CHCKACCT*--
Status Code=

GNP
CUSTOMER*U-(SSNUMBEREQ434-62-1234)
CHCKACCT*U-(ACNUMBEREQ345620104732)
CHCKDEBT*--

Status Code=GE

GNP
CUSTOMER*U-(SSNUMBEREQ434-62-1234)
CHCKACCT*--

Status Code=GE
GN
CUSTOMER*--                            gets CUSTOMER 3
Status Code=

GNP
CUSTOMER*U-(SSNUMBEREQ436-42-6394)
CHCKACCT*--

Status Code=GE

GN
CUSTOMER*--

Status Code=GB
```

Note: The data retrieval process for GSAM databases is somewhat different. After issuing an initial close call (CLSE) to establish position at the beginning of the database, the IMS engine uses unqualified GN calls to retrieve all the data in the database. △

WHERE Statement Processing

There are many ways to subset data in SAS by using the following tools:

- ☐ a WHERE statement in a view descriptor
- ☐ a SAS WHERE statement in a PROC or DATA step
- ☐ a PROC SQL SELECT statement's WHERE clause
- ☐ a WHERE command in the SAS/FSP procedures
- ☐ a SAS data set WHERE option.

These all use SAS WHERE statement syntax. You do not have to use IMS SSA syntax with the IMS engine that runs under Version 7 and later of SAS.

The IMSI engine attempts to build SSAs from the WHERE conditions that you enter; *condition* refers to the expression(s) in the WHERE statement, clause, command, or option. The engine uses these SSAs to qualify each call to the database. Therefore, IMS returns to SAS only those observations that meet your conditions. However, if the IMS engine cannot convert the WHERE condition into SSAs, it passes all database segments referenced by the view descriptor to SAS, which then subsets and processes the data. Because it uses more resources to have SAS process WHERE conditions, you should try to use WHERE conditions that can be turned into valid SSAs when resources are a concern.

To specify WHERE conditions that the IMS engine can use to generate SSAs, use one of the operators supported by IMS. In the access descriptor, define search field names from the DBD for all the variables included in your WHERE condition when possible. See "Writing Efficient WHERE Statements" on page 126 for a list of the operators IMS supports.

Note: IMS SSAs do not support conditions that use OR and combine elements from two different segments. △

The engine uses the search field names that are entered in the view descriptor for the field names in the SSAs. Therefore, if you use a SAS variable in a WHERE condition for which you do not have a search field defined, the IMS engine cannot generate SSAs for that WHERE condition.

If the WHERE statement or clause contains multiple conditions and any one of the conditions cannot generate a qualified SSA, then no qualified SSA is generated from the statement or clause.

If the IMS engine can handle a WHERE condition, it uses the SEARCH= argument in the ITEM= statement to generate a qualified SSA. If possible, the engine combines the qualified SSAs that it generated to navigate the database with any WHERE condition SSAs. If both SSAs involve the same field, only the WHERE SSA is used to avoid a mutually exclusive situation. The engine then issues a path call to obtain the segments in the hierarchy down to the lowest level with an item specified in the WHERE condition. All segments in the path are retrieved and passed to SAS. Therefore, if you use a WHERE condition from which the IMS engine can generate SSAs, the Program Specification Block (PSB) specified in that view descriptor must allow path calls for the segments in the hierarchy above and including segments with variables in the WHERE condition.

For example, if you enter the WHERE condition

```
WHERE CHCKACCT = 345620145345
```

the IMS engine passes the following SSAs to IMS:

```
CUSTOMER*D-
CHCKACCT*--(ACNUMBERREQ345620145345)
```

The IMS engine uses the results of this call to generate SSAs to navigate the database further and to flatten out the IMS record as defined in the view descriptor.

The engine combines these navigational SSAs with the SSA that it generated from the WHERE condition for the CHCKACCT segment. The engine continues processing and retrieves the view descriptor's lowest level segment (CHCKDEBT), which is a child of the CHCKACCT segment. CHCKACCT has an ACNUMBER value that is equal to 345620145345 until the engine does not find another CHCKDEBT segment (status code GE).

To improve the efficiency of using a WHERE condition to subset your data, use the operators supported by IMS. Enter the search field names of all variables in the WHERE condition so that the IMS engine can pass only a subset of data to SAS for further processing. Use the SAS system option IMSDEBUG=Y to see whether your WHERE condition is generating SSAs directly.

Note: For GSAM databases, the IMS engine always passes WHERE clauses to SAS for processing. △

Data Retrieval by Using a Secondary Index

The SAS/ACCESS interface enables you to take advantage of secondary indexes in IMS. A secondary index enables a SAS application to complete the following tasks:

□ access a segment type in the database in a sequence other than the sequence that is specified by the key field. For example, the application might need to access a database by phone numbers—a field in the root segment of the database—rather than by the social security number, which is the segment's key.

□ change the view of the database data based on a condition in a dependent segment in the database. For example, a banking application might need to access the database (normally accessed by the social security number, the root segment's key field) by the checking account number or the savings account number, which are key fields in dependent segments to the root segment.

Because IMS stores root segments in the sequence of their key fields, an application that accesses the data in another order would be inefficient. A database administrator (DBA), in conjunction with the SAS application user analyst or programmer, determines if a secondary index is needed and assists in laying out the secondary index. By using secondary indexing, IMS can go directly to a segment based on a field that is not the key field.

You can define your access descriptors and view descriptors so that they can access secondary indexes, as described in this section.

In IMS, when an application requests that a segment be returned based on the database call and SSA combination, the segment that is returned is called the *target segment*. If an application requests only one segment, that segment becomes the target segment. If a sequence of SSAs is used, then the lowest level segment retrieved in the hierarchy is the target segment. If you issue a path call for multiple segments, all the segments are returned to the I/O area.

To use secondary indexes with SAS applications, you have to assign certain IMS parameters and use certain arguments when you create an access descriptor. The PCB that you use must specify the PROCSEQ parameter, which causes IMS to use the secondary index. You also might need to use the PCBINDX= argument when you create a view descriptor so that the correct PCB is used by the engine. The secondary index is transparent to the application because it is automatically accessed when these parameters are assigned.

To create a secondary index, the DBD for the database must contain XDFLD statements that do the following:

□ define the name of an indexed field that is associated with an index target segment type

□ identify the index source segment type

□ identify the index source segment fields that are used in creating a secondary index.

One XDFLD statement is required for each secondary index relationship.

Combining Segments to Define Descriptors

This section lists ways in which target and source segments can be related and, therefore, how you should define your descriptors in order to access the IMS data through a secondary index.

□ When the source segment and the target segment are the same, and the target segment is the root segment the following is true:

The *source segment* supplies the field(s) values that comprise the secondary index. The secondary index stores these values in order with other information that specifies where the target segment is located for any value of the secondary index.

The XDFLD statement contains the NAME= value that is used in the SEARCH= argument, because doing so gives the secondary index the same name as will be used in the application's SSAs.

□ When the source segment and the target segment are the same, and the target segment is *not* the root segment the following is true:

The database is conceptually restructured. The DBA and the SAS applications analyst or programmer lay out how the database will look conceptually. Physically, the database is still the same. This causes the SAS application to access the data by using the secondary index data structure of the database. For this case, in addition to the scenario described in the first item of this list, the entire access descriptor definition must describe the secondary index data structure and not the primary structure.

□ When the source segment and the target segment are *not* the same, and the target segment is the root segment the following is true:

There is no secondary index data structure because the target segment is the root segment. However, the target and source segments are separate segments in the database.

In order for you to create an access descriptor using separate segments, you must add a dummy field to the end of the root segment. This dummy field must contain a length that matches the field(s) length for the target segment key value. In addition, the SEGLNG value for the entire root segment must be increased in the DBD to allow for this additional field. Any valid SAS name can be assigned to this dummy field, but the SEARCH= value must be the XDFLD name for the field from the DBD.

In essence, the dummy field is a virtual field in the access descriptor definition for the root. It does not physically exist there, but a SAS application can submit SSA references for the target (in this case the root) that is qualified on this field.

For example, consider a SAS application that uses the following WHERE statement:

```
WHERE sasname EQ value
```

Sasname is the SAS variable name for the virtual field and *value* is a value for the field in the source segment. The IMS engine properly builds a SSA for the target (root) that is qualified by using the XDFLD name for the field and the value from the WHERE clause.

    ☐ When the source segment and the target segment are *not* the same, and the target segment is *not* the root the following is true:

        This is the most complicated. It combines the scenarios described in the previous two list items. The same dummy field must be added to the target segment as in the previous list item. In addition, the entire access descriptor must map to or define the secondary data structure that results from the target not being the root.

Data Modification Processing

    Modifying a hierarchical database such as an IMS database can be complicated. Therefore, you need to know how the IMS engine operates in order to perform database modifications.

    If you plan to use a view descriptor to update the database, the IMS engine requires that you designate one search field as a key (that is, one key field) for each hierarchical level in the database. You designate the key field when you create the access descriptor on which the view descriptor is based. The key fields must be selected in the view descriptor.

    Note: The search field that you designate as the key *must* be defined in the DBD as a key field; otherwise, updating results might be unpredictable. In addition, you cannot skip hierarchical levels in a view descriptor that you want to use to update the database. Because the IMS engine uses path calls to perform most updates, no ROLB (ROLLBACK) calls are required. If a path call fails, the engine returns an error to SAS and no update is performed. △

    The engine, by default, issues checkpoints at the beginning and end of the update process. You can use the AUTOSAVE option with SAS/FSP software to increase the frequency of issuing checkpoints. Your update PSB must allow path call processing, and an I/O PCB must be included for checkpoint calls.

    The only time an update is performed with multiple DL/I calls is when you request both an update and an insert. For example, you could use the FSEDIT procedure to update a CUSTOMER segment and, on the same display, enter information to insert a new CHCKACCT segment under the CUSTOMER parent segment you just modified. In this case, if the insert call fails after the engine has processed the modification, the IMS engine issues another update call that replaces the modified parent segment with the original data in that segment. This process uses fewer resources than a ROLB call. (See Chapter 9, "How to Use the IMS DATA Step Interface," on page 197 for information about ROLB and other non-database access calls.)

Delete Processing

    You can delete only the lowest existing segment defined in the view descriptor.

CAUTION:
    If you delete a segment that has children that are not defined in the view descriptor, the children are also deleted. △

    For example, if your view includes the CUSTOMER and CHCKACCT segments only and you delete a CHCKACCT segment, any CHCKDEBT segments under CHCKACCT are also deleted even though they are not defined in the view descriptor.

    If your view descriptor includes all the hierarchical levels but a particular segment has no children, the lowest existing segment is deleted. For example, if a CUSTOMER segment occurrence has no CHCKDEBT segments under a CHCKACCT segment, issuing the DELETE command deletes the CHCKACCT segment. If you then have only a CUSTOMER segment and you issue the DELETE command, the CUSTOMER segment is deleted.

Note: You cannot delete segments in a GSAM database. △

Add Processing

SAS/FSP software provides three ways to insert new data into an IMS database:

□ To add a path of data to your database, enter the new data using the FSVIEW procedure (with the MODIFY command) or the FSEDIT procedure, and issue the ADD command. The IMS engine adds all the data that you entered as a new path of data in your database.

□ To add a new child segment under an existing root segment that does not have any children, use PROC FSVIEW (with the MODIFY command) or PROC FSEDIT to display the existing segment. Enter the child data on the screen below the existing parent segment.

□ To add a twin segment to an existing child segment, you must first use PROC FSVIEW (with the MODIFY command) or PROC FSEDIT to display the segment to which you want to add a twin. Enter the new data by typing over the existing child, making sure you change the key field of the segment to which you want to add a twin. The IMS engine then inserts a twin segment. Any segments which appear on the screen under the changed segment are also added under the new twin segment in a path call.

Note that you can add segments only at the end of a GSAM database.

You can also use the APPEND procedure, DATA step MODIFY statement, or an INSERT statement in the SQL procedure to add data to an IMS database. To insert a path of data, use a view descriptor that describes the entire path to be inserted. To insert child segments under a parent segment, enter the key field value of the parent segment. The new data will be inserted under the existing parent.

Update Processing

The IMS engine compares the data you entered to the data that is stored in the I/O area from the last call. If you change any data in a path of data, the engine replaces only the segments that have changed in the path. If you change the key field defined in the view descriptor, the IMS engine inserts a twin segment occurrence under the current parent segment.

Note: You cannot update segments in a GSAM database. △

P A R T *4*
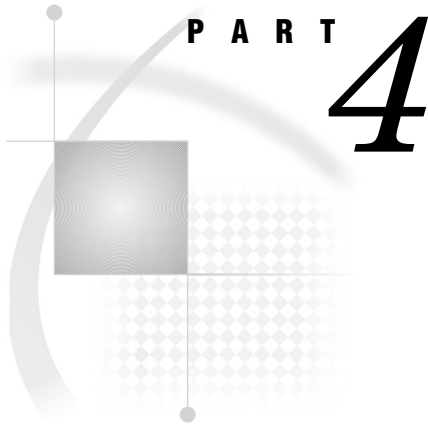
The IMS DATA Step Interface: Reference

CHAPTER

8

Overview of the IMS DATA Step Interface

Introduction to the IMS DATA Step Interface

Special SAS extensions for the standard SAS INFILE and FILE statements enable you to format DL/I calls in a SAS DATA step. These extended SAS statements and their corresponding INPUT and PUT statements are called DL/I INFILE, DL/I INPUT, DL/I FILE, and DL/I PUT to distinguish them from the standard SAS statements. An IMS DATA step can contain standard SAS statements as well as the SAS statements that are used with the SAS/ACCESS interface to IMS.

The beginning of this section describes the syntax of the SAS statement extensions that are used with the SAS/ACCESS interface to IMS. The next section describes basic DATA step programming techniques and considerations for this IMS interface. The last

section consists of sample DATA step programs that access DL/I databases. The sample programs integrate many of the concepts that are discussed throughout the section.

This section assumes that you understand the SAS DATA step and the statements used in the DATA step. See *SAS Language Reference: Dictionary* for details about the statements, options, and syntax in SAS DATA steps.

There are many references to DL/I processing in this description, such as DL/I calls and status codes. If you are not familiar with the DL/I information, be sure to refer to the appropriate IBM documentation for complete descriptions. You should also read this document's Chapter 2, "IMS Essentials," on page 11 which gives an overview of DL/I concepts that are important in writing DATA step programs for the DATA step interface to IMS.

Overview of the DATA Step Statement Extensions

In a DATA step, the SAS/ACCESS interface to IMS uses special extensions of standard SAS INFILE and FILE statements to access DL/I resources. These extended statements are referred to as the DL/I INFILE and DL/I FILE statements, and their corresponding INPUT and PUT statements are referred to as DL/I INPUT and DL/I PUT statements.

DL/I INFILE and DL/I INPUT statements work together to issue DL/I get calls. The DL/I INFILE, DL/I FILE, and DL/I PUT statements work together to issue DL/I update calls.

The DL/I INFILE statement tells SAS where to find the parameters needed to build DL/I calls. Special DL/I INFILE statement extensions perform the following tasks:

- □ name the PSB
- □ specify a SAS variable or a number that selects the appropriate PCB in the PSB
- □ specify a SAS variable that contains DL/I call functions (for example, GN or REPL)
- □ specify SAS variables that contain SSAs for the DL/I call
- □ name SAS variables to contain information returned by the call, for example, the status code and retrieved segment name.

The DL/I INFILE statement is necessary to identify the parameters for a call. However, the call is not actually formatted and issued until a DL/I INPUT statement is executed for get calls or DL/I FILE and DL/I PUT statements are executed for update calls.

The DL/I INFILE statement is required in any DATA step that accesses a DL/I database because the special extensions of the DL/I INFILE statement specify variables that set up the DL/I calls. When a DL/I INFILE statement is used with a DL/I INPUT statement, get calls are issued. When a DL/I INFILE statement is used with DL/I FILE and DL/I PUT statements, update calls are issued. Both get and update calls can be issued in one DATA step.

The syntax and use of the DL/I INFILE, DL/I FILE, DL/I INPUT, and DL/I PUT statements are described in detail later in this section.

DL/I Input and Output Buffers

Two separate buffers are allocated by SAS as I/O areas for data transfer. The *input buffer* is for DL/I segments retrieved by get calls. The *output buffer* is for data written by an update call. The length of each buffer is specified by the LRECL= option in the DL/I INFILE statement. The default length for each buffer is 1,000 bytes.

The input buffer is formatted by DL/I in the same way an I/O area for any DL/I program is formatted. If a fixed-length segment is retrieved, the fixed-length segment

begins in column 1 of the input buffer. If a segment of varying length is retrieved, the length field (LL field) in IB2. format (half-word binary) begins in column 1 and the varying-length segment data follows immediately. If a path of segments is retrieved, the buffer contains the concatenated segments.

The format of the output buffer is like that of the input buffer. If a fixed-length segment is written, the fixed-length segment begins in column 1 of the output buffer. If a varying-length segment is written, the length field in IB2. format (half-word binary) begins in column 1. The varying-length segment data immediately follows the length field. If a path of segments is written, the buffer contains the concatenated segments.

The segment data format in the output buffer is determined by the DL/I PUT statement and must match the original segment data format. See "Using the DL/I PUT Statement" on page 180 for more information about how to format segment data in the output buffer.

The format of the data in a segment is determined by the application program that wrote the data segment originally, just as the data format in any other record is determined by the program that writes the record. When you write an IMS DATA step program you must know the segment's format in order to read data from the segment with a DL/I INPUT statement or to write data to the segment with a DL/I PUT statement.

In most cases, you are probably not the person who originally determined the segment data format. Segment data format information is stored in different ways at different installations. For example, the information can be obtained from a data dictionary, COBOL or Assembler copy libraries, source programs, a SAS macro library, or other documentation sources. DBA staff at your installation can help you find the segment data formats you need.

An Introductory Example of a DATA Step Program

The following example is a simple IMS DATA step program that reads segments from a DL/I database and creates a SAS data set from data in the retrieved segments. Next, the program processes the SAS data set with PROC SORT and PROC PRINT.

The example accesses the ACCTDBD database with a PSB called ACCTSAM. ACCTSAM contains five PCBs; the second PCB contains a view of the ACCTDBD database in which the CUSTOMER segment is the only sensitive segment. See Appendix 2, "Example Data," on page 255 for information about the databases, PSBs, segments, and fields used in this example and other examples in this document. This example uses the DLI option of the INFILE statement, which tells SAS that the INFILE statement refers to a DL/I database. Other nondefault region and execution parameters in effect include these:

□ The second PCB in the specified PSB is used.

□ Status codes are examined.

Defaults for other region and execution parameters in this example include these:

□ A DL/I region is used.

□ The DL/I calls issued are all GN (get-next) calls.

□ No SSAs are used.

□ Program access is sequential.

□ PCB feedback mask data is not examined.

If you do not want to use these defaults, the special statement and product options that you can specify for IMS are described later in this section.

The numbered comments following this program correspond to the numbered statements in the program:

```
❶  data work.custlist;
❷     infile acctsam dli status=st pcbno=2;
❸     input @1   soc_sec_number  $char11.
            @12  customer_name    $char40.
            @52  addr_line_1      $char30.
            @82  addr_line_2      $char30.
            @112 city             $char28.
            @140 state            $char2.
            @142 country          $char20.
            @162 zip_code         $char10.
            @172 home_phone       $char12.
            @184 office_phone     $char12.;

❹    if st ⌐= '  ' then
       do;
          file log;
          put _all_;
          abort;
       end;
     run;

❺    proc sort data=work.custlist;
       by customer_name;

❻    options linesize=132;
     proc print data=work.custlist;
       var home_phone office_phone;
       id customer_name;
       title2 'Customer Phone List';

❼    proc print data=work.custlist;
       var addr_line_1 addr_line_2 city
           state country zip_code;
       id customer_name;
       title2 'Customer Address List';
     run;
```

❶   The DATA statement references a temporary SAS data set called CUSTLIST, which is to be opened for output.

❷   The INFILE statement tells SAS to use a PSB called ACCTSAM. The DLI option tells SAS that ACCTSAM is a DL/I PSB instead of a fileref. The statement also tells the IMS interface to use the second PCB and to return the DL/I STATUS code in the ST variable.

❸   The INPUT statement causes a GN (get-next) call to be issued. The PCB being used is sensitive only to the CUSTOMER segment, so the get-next calls retrieve only CUSTOMER segments. When the INPUT statement executes, data is retrieved from a CUSTOMER segment and placed in the input buffer. The data is then moved to the specified SAS variables in the program data vector (SOC_SEC_NUMBER, CUSTOMER_NAME, and so on).

           As the DATA step executes, CUSTOMER segments are retrieved from ACCTDBD, and SAS observations that contain the

CUSTOMER data are written to the CUSTLIST data set. Because program access is sequential, the DATA step stops executing when the DL/I STATUS code indicates an end-of-file condition.

❹ The status code is checked for non-blank values. For any non-blank status code except **GB**, all values from the program data vector are written to the SAS log, and the DATA step aborts. If the status code variable value is **GB**, the DATA step will terminate with an end-of-file condition if the processing was sequential (using non-qualified SSAs). Since this example uses no SSA, the database is processed sequentially and no check for a status code of **GB** is required.

❺ The SORT procedure sorts the CUSTLIST data set alphabetically by customer name.

❻ The PRINT procedure first prints a Customer Phone List.

❼ The procedure is invoked again to print a Customer Address List.

The following output shows the SAS log for this example.

Output 8.1   SAS LOG for Introductory IMS Data Step Example

```
12          data work.custlist;
13            infile acctsam dli status=st pcbno=2;
14            input @1    soc_sec_number $char11.
15                  @12  customer_name   $char40.
16                  @52  addr_line_1     $char30.
17                  @82  addr_line_2     $char30.
18                  @112 city           $char28.
19                  @140 state          $char2.
20                  @142 country        $char20.
21                  @162 zip_code       $char10.
22                  @172 home_phone     $char12.
23                  @184 office_phone   $char12.;
24            if st ^= ' ' then
25              do;
26                file log;
27                put _all_;
28                abort;
29              end;
30

NOTE: The infile ACCTSAM is:
      (system-specific pathname),
      (system-specific file attributes)

NOTE: GB -End of database encountered
NOTE: 10 records were read from the infile (system-specific pathname).
      The minimum record length was 225.
      The maximum record length was 225.
NOTE: The data set WORK.CUSTLIST has 10 observations and 10 variables.

31          proc sort data=work.custlist;
32            by customer_name;
33
34          options linesize=132;

NOTE: The data set WORK.CUSTLIST has 10 observations and 10 variables.

35          proc print data=work.custlist;
36            var home_phone office_phone;
37            id customer_name;
38            title2 'Customer Phone List';
39

NOTE: The PROCEDURE PRINT printed page 1.

40          proc print data=work.custlist;
41            var addr_line_1 addr_line_2 city state country zip_code;
42            id customer_name;
43            title2 'Customer Address List';
44          run;

NOTE: The PROCEDURE PRINT printed page 2.
```

The following two outputs show the results of this example.

Output 8.2   Customer Phone List — Results of Introductory Example

```
                    Customer Phone List

        customer_name           home_phone      office_phone

        BARNHARDT, PAMELA S.     803-345-4346    803-355-2543
        BOOKER, APRIL M.         803-657-1346
        COHEN, ABRAHAM           803-657-7435    803-645-4234
        LITTLE, NANCY M.         803-657-3566
        O'CONNOR, JOSEPH         803-657-5656    803-623-4257
        PATTILLO, RODRIGUES      803-657-1346    803-657-1345
        SMITH, JAMES MARTIN      803-657-3437
        SUMMERS, MARY T.         803-657-1687
        WALLS, HOOPER J.         803-657-3098    803-645-4418
        WIKOWSKI, JONATHAN S.    803-467-4587    803-654-7238
```

Output 8.3   Customer Address List — Results of Introductory Example

```
                         Customer Address List

                      addr_
    customer_name      line_1   addr_line_2        city             state   country   zip_code

    BARNHARDT, PAMELA S.        RT 2 BOX 324       CHARLOTTESVILLE   VA      USA       25804-0997
    BOOKER, APRIL M.            9712 WALLINGFORD PL. GORDONSVILLE    VA      USA       26001-0670
    COHEN, ABRAHAM              2345 DUKE ST.      CHARLOTTESVILLE   VA      USA       25804-0997
    LITTLE, NANCY M.            4543 ELGIN AVE.    RICHMOND          VA      USA       26502-3317
    O'CONNOR, JOSEPH            235 MAIN ST.       ORANGE            VA      USA       26042-1650
    PATTILLO, RODRIGUES         9712 COOK RD.      ORANGE            VA      USA       26042-1650
    SMITH, JAMES MARTIN         133 TOWNSEND ST.   GORDONSVILLE      VA      USA       26001-0670
    SUMMERS, MARY T.            4322 LEON ST.      GORDONSVILLE      VA      USA       26001-0670
    WALLS, HOOPER J.            4525 CLARENDON RD  RAPIDAN           VA      USA       22215-5600
    WIKOWSKI, JONATHAN S.       4356 CAMPUS DRIVE  RICHMOND          VA      USA       26502-5317
```

Example of Using DATA Step Views

The preceding introductory DATA step example can also be made into a DATA step view. A DATA step view is a SAS data set of type VIEW. It contains only a definition of data that is stored elsewhere, in this case, in a DL/I database; the view does not contain the physical data.

A DATA step view is a stored, named DATA step program that you can specify in other SAS procedures to access IMS data directly. A view's input data can come from one or more sources, including external files and other SAS data sets.

The following DATA step code is contained in a macro that is invoked twice to create two distinct DATA step views. When the DATA step views are executed, CUSTOMER segments are read from the ACCTDBD database and selected data values are placed in two SAS data sets. Then each SAS data set is processed with PROC SORT and PROC PRINT to produce the same outputs as the introductory example in "An Introductory Example of a DATA Step Program" on page 155.

The numbered comments following this program correspond to the numbered statements in the program:

❶ %macro custview(viewname=,p1=,p2=,p3=,p4=,p5=,
        p6=,p7=,p8=,p9=,p10=);
❷ data &viewname / view=&viewname;
❸ keep &p1 &p2 &p3 &p4 &p5 &p6 &p7 &p8 &p9 &p10;
❹ infile acctsam dli status=st pcbno=2;
    input @1   soc_sec_number $char11.
          @12   customer_name $char40.
          @52   addr_line_1 $char30.
          @82   addr_line_2 $char30.
          @112 city $char28.
          @140 state $char2.
          @142 country $char20.
          @162 zip_code $char10.
          @172 home_phone $char12.
          @184 office_phone $char12.;

    if st ¬= ' ' then
       do;
          file log;
          put _all_;
          abort;
       end;
❺ %mend;

❻ %custview(viewname=work.phone,
        p1=customer_name,
        p2=home_phone,
        p3=office_phone);

❼ %custview(viewname=work.address,
        p1=customer_name,
        p2=addr_line_1,
        p3=addr_line_2,
        p4=city,
        p5=state,
        p6=country,
        p7=zip_code);

 options linesize=132;

❽ data work.phonlist;
    set work.phone;
 run;

❾ proc sort data=work.phonlist;
    by customer_name;
 run;

 proc print data=work.phonlist;
    title2 'Customer Phone List';
 run;

❿ data work.addrlist;
    set work.address;

```
   run;

⓫ proc sort data=work.addrlist;
      by customer_name;
   run;

proc print data=work.addrlist;
      title2 'Customer Address List';
   run;
```

❶   %MACRO defines the start of the macro CUSTVIEW which allows 11 input overrides. VIEWNAME is the name of the DATA step view to be created. The following are the other 10 overrides:

P1	name of the 1st data item name to keep.
P2	name of the 2nd data item name to keep.
P3	name of the 3rd data item name to keep.
P4	name of the 4th data item name to keep.
P5	name of the 5th data item name to keep.
P6	name of the 6th data item name to keep.
P7	name of the 7th data item name to keep.
P8	name of the 8th data item name to keep.
P9	name of the 9th data item name to keep.
P10	name of the 10th data item name to keep.

    Ten data items are allowed because there are 10 input fields in the INPUT statement for the database.

❷   The DATA statement names the DATA step view as specified by the macro variable &VIEWNAME.

❸   The KEEP statement identifies the variables that will comprise the observations in the output data set. In this case, there will be as many as 10.

❹   This is the same code that was executed in the introductory example in "An Introductory Example of a DATA Step Program" on page 155.

❺   %MEND defines the end of macro CUSTVIEW.

❻   %CUSTVIEW generates a DATA step view named WORK.PHONE, which when executed produces observations containing the data items CUSTOMER_NAME, HOME_PHONE, and OFFICE_PHONE.

❼   %CUSTVIEW generates a DATA step view named WORK.ADDRESS, which when executed produces observations containing the data items CUSTOMER_NAME, ADDR_LINE_1, ADDR_LINE_2, CITY, STATE, COUNTRY, and ZIP_CODE.

❽   Data set WORK.PHONLIST is created by obtaining data using the DATA step view WORK.PHONE.

❾   PROC SORT sorts WORK.PHONLIST and PROC PRINT prints it out.

❿   Data set WORK.ADDRLIST is created by obtaining data using the
               DATA step view WORK.ADDRESS.

⓫   PROC SORT sorts WORK.ADDRLIST and PROC PRINT prints it
               out.

The DL/I INFILE Statement

If you are unfamiliar with the standard INFILE statement, refer to *SAS Language
Reference: Dictionary* for more information.

A standard INFILE statement specifies an external file to be read by an INPUT
statement. A DL/I INFILE statement specifies a PSB, which in turn identifies DL/I
databases or message queues to be accessed with DL/I calls. Special extensions in the
DL/I INFILE statement specify SAS variables and constants that are used to build a
DL/I call and to handle the data returned by the call. A limited selection of the
standard INFILE statement options can also be specified in a DL/I INFILE statement.

To issue get calls, use the DL/I INFILE statement with the DL/I INPUT statement.
To issue update calls, use the DL/I FILE and DL/I PUT statements with the DL/I
INFILE statement.

Note that there is an important difference between the standard INFILE statement
and the DL/I INFILE statement: you must use a corresponding INPUT statement with
a standard INFILE statement, but you can use a DL/I INFILE statement without a
DL/I INPUT statement. The standard INFILE statement has no effect without a
corresponding INPUT statement because the standard INFILE statement points to a
file to be read with INPUT statements. However, a DL/I INFILE statement does not
always have an accompanying DL/I INPUT statement. Instead, it can be grouped with
DL/I FILE and DL/I PUT statements. When combined with DL/I FILE and DL/I PUT
statements, the DL/I INFILE statement points to a PSB and specifies SAS variables
and constants that are used to build update calls. In other words, a DL/I INFILE does
not always imply that you are reading from a DL/I database; it is also used if you are
writing to the database.

Use the following syntax when issuing a DL/I INFILE statement:

```
INFILE PSBname DLI options;
```

PSBname
   specifies the name of the PSB used to communicate with DL/I in the current DATA
   step. A *PSBname* must be specified in a DL/I INFILE statement and must
   immediately follow the keyword INFILE. (A standard INFILE statement would
   specify a fileref in this position.)

   All DL/I INFILE statements in the same DATA step must specify the same PSB
   name. You cannot use more than one PSB in a DATA step. Therefore, the PSB
   must be sensitive to all DL/I databases or message queues that you want to access.
   Different PSBs can be used in different DATA steps.

   Note: The PSB name cannot be the same name as a fileref on a JCL
   statement. △

DLI
   tells SAS that this INFILE statement refers to DL/I databases or message queues.
   DLI must be specified immediately following the PSB name in a DL/I INFILE
   statement.

The options described in the next two sections can appear in the DL/I INFILE
statement but are not required. Many of these options identify a SAS variable that

contains DL/I information. These variables are not added automatically to a SAS output data set (that is, they have the status of variables that are dropped with the DROP option). If you want to include the variables in an output SAS data set, you will need to create separate variables and assign values to them. Most of the variables do not need to be predefined before specification in the DL/I INFILE statement. SAS allocates them automatically with the correct type and length. However, the SSA variables are an exception.

PCB Selection Options

PCBNO=*number*
> defines the first eligible PCB in the PSB (specified by *PSBname*). For example, if you specify PCBNO=3, the first eligible PCB is the third PCB in the PSB. This option enables you to bypass PCBs that are inappropriate for your program. You can combine PCBNO= with the DBNAME= option or the PCB= option (described later in this section) to select a particular PCB for your program.
>
> If PCBNO= is not specified, the first eligible PCB is the first PCB in the PSB.

DBNAME=*variable*
> specifies a SAS *variable* that contains a DL/I DBD name. The value of the variable determines which of the eligible PCBs is used for the DL/I call. When DBNAME= is specified, the eligible PCBs are searched sequentially, starting with the first eligible PCB. Refer to the description of the PCBNO= option earlier in this section for more information. The first eligible PCB with a DBD name that matches the value of the DBNAME= variable is used. You must enter the variable in uppercase letters.
>
> For example, if PCBNO=5, DBNAME=DB, and the value of the DB variable is ACCOUNT, SAS searches for a PCB with the DBD name ACCOUNT beginning with the fifth PCB, which is the first eligible PCB.
>
> The DBNAME= variable must be assigned a valid eight-character DBD name (padded with blanks if necessary) or a blank character string prior to execution of a DL/I INPUT or DL/I PUT statement that issues a DL/I call. The value of the variable specified by the DBNAME= option can be changed between calls.
>
> If the DBNAME= option is not specified or the DBNAME= variable contains a blank character string, the PCB= option (described later in this section) is used to select the appropriate PCB, if specified. If neither the DBNAME= option nor the PCB= option is specified, the first PCB in the PSB is used for every DL/I call.
>
> DBNAME= is convenient because you do not have to know which PCB refers to a particular database; you need to know only the DBD name for the database you want to access. However, if more than one eligible PCB refers to the same database, only the first of these PCBs is used. You must specify the PCB= option rather than DBNAME= if more than one eligible PCB refers to the same database and you want to use any PCB other than the first one for the database.

PCB=*variable*
> names a SAS *variable* that is an index for the list of eligible PCBs as defined by the PCBNO= option. The value of the PCB= variable indicates which PCB in the eligible list to use. The specified variable must be numeric and must be assigned a value prior to execution of a DL/I INPUT or DL/I PUT statement. The value of the specified variable can be changed between calls.
>
> Consider an example that uses the PCBNO= and PCB= options. Assume that PCBNO=3, PCB=PCBNDX, and PCBNDX has a value of 2. Since PCBNO=3, the third PCB in the PSB is the first eligible PCB, and since PCBNDX has a value of 2, the second eligible PCB (that is, the fourth PCB in the PSB) is used.

If the DBNAME= option is also specified and the DBNAME= variable's value is not blank, the PCB= variable value is not used. If neither the DBNAME= option nor the PCB= option is specified, the first eligible PCB is used for every DL/I call by default.

Other DL/I INFILE Options

CALL=*variable*
names a SAS *variable* that contains the DL/I call function used when a DL/I INPUT or DL/I PUT statement is executed. *Variable* must be assigned a valid four-character DL/I call function code before a DL/I INPUT or DL/I PUT statement is executed. The value must be entered in capital letters and be a valid get call function for any DL/I INPUT statement execution (for example, `'GU   '`). It must be a valid update call function for any DL/I PUT statement execution (for example, `'REPL'`). The following table shows the calls executed by DL/I INPUT statements and those executed by DL/I PUT statements.

Table 8.1 Calls Executed by DL/I INPUT and DL/I PUT Statements

DL/I INPUT Statement	DL/I PUT Statement
GU	ISRT
GHU	REPL
GN	DLET
GHN	CHKP
GNP	ROLL
GHNP	ROLB
GCMD	CHNG
STAT	LOG
POS	PURG
	CMD
	DEQ
	FLD
	OPEN
	CLSE

The value of the CALL= variable can be changed between calls.
If CALL= is not specified, the call function defaults to GN (get next). In this case, a DL/I PUT statement would not have a valid call function because DL/I PUT statements execute update calls, and should not be used.

FSARC=*variable*
specifies a SAS *variable* that contains the concatenated status code bytes of each field search argument (FSA) of an MVS IMS/VS Fast Path FLD call. The first character of *variable* contains the first FSA status code value, the second character

contains the second FSA status code value, and so forth. The specified variable is a character variable with a default length of 200. Since each status code is one byte in length, as many as 200 FSA status codes can be stored.

If FSARC= is not specified, the FSA status codes are not returned.

LENGTH=*variable*

specifies a SAS *variable* that contains the length of the segment or path of segments retrieved when a DL/I get call is executed. The variable that is specified must be numeric.

You can find the length of fixed-length segments in the DBD for the database. If a segment has a varying length, the length information is contained in the first two bytes of the segment, that is, in the LL field. To obtain the length data from the LL field of the segment, simply specify the LL field in the DL/I INPUT statement:

```
input @1 ll pib2.
      @3 loan_num
      @10 terms;
```

Be aware that in some cases the value that is returned for the LENGTH= variable or INFILE notes might not represent the length of the segment data correctly. This is due to the method SAS uses to determine the length. The entire input buffer is filled with the hex characters X'2E' before the call is executed. When DL/I executes the get call, segment data overwrites the X'2E' characters until the segment data ends. SAS scans the buffer, looking for the first occurrence of the X'2E' sequence. If the remainder of the buffer is filled with X'2E' or if there are 256 consecutive X'2E's, SAS assumes that the sequence indicates the end of the returned data and calculates the segment length. However, if the segment data happens to contain 256 consecutive bytes of X'2E' or end with one or more bytes with this value, the returned length value is incorrect.

LRECL=*length*

specifies the *length* of the SAS buffers used as I/O areas when DL/I calls are executed. The length must be greater than or equal to the length of the longest segment or path of segments accessed. If LRECL= is not specified, the default buffer length is 1000 bytes.

If a retrieved segment or path of segments is longer than the value of LRECL=, DL/I overlays other data or instruction storage areas. Unpredictable results can occur if this happens.

PCBF=*variable*

names a SAS *variable* that contains feedback values from the PCB mask data that is generated by each DL/I call. The specified variable is a character variable with a default length of 200.

Some of the data returned in the PCBF= variable is the same as that returned in the SEGMENT= variable and STATUS= variable described below. Separate options are available for segment and status data because they are more commonly used in controlling the program flow.

If the DL/I call uses a database PCB, the mask data returned in the PCBF= variable is formatted as shown in Table 8.2 on page 166. The format of the PCBF= variable is different when a non-database PCB (an I/O PCB or TP PCB) is used in the DL/I call. See Chapter 10, "Advanced Topics for the IMS DATA Step Interface," on page 219 for information about the format of the mask data for a non-database PCB.

If PCBF= is not specified, the mask data is not returned (except segment and status information if the SEGMENT= and STATUS= options are specified).

Particular data can be extracted from the mask data using the SAS function SUBSTR. For example, this assignment statement extracts the value of the first eight bytes, the DBD name. PCBMASK is the PCBF= variable:

```
dbdname=substr(pcbmask,1,8);
```

To extract data that is stored in a nonstandard format, use the INPUT and SUBSTR functions. For example, this assignment statement extracts the value of bytes 9 and 10, the segment level number:

```
seglev=input(substr(pcbmask,9,2),ib2.);
```

Table 8.2 Format of Data Returned in the PCBF= Variable for a Database PCB

Bytes	Description
1–8	These bytes of the PCBF= variable contain the DBD name.
9–10	The level number of the last segment accessed is contained in bytes 9 and 10 in IB2. format. Level number refers to a segment's level in the hierarchical structure. For example, your program might issue a qualified GN call with these SSAs: **CUSTOMER*D-(SSNUMBER =667-73-8275)** **CHCKACCT*D- (ACNUMBER =345620145345)** **CHCKCRDT (CRDTDATE =033195)** If segments exist to satisfy the CUSTOMER and CHCKACCT SSAs but there is no CHCKCRDT segment with a CRDTDATE field value of 033195, the last segment accessed is the CHCKACCT segment. CHCKACCT is at the second level of the hierarchy; therefore, the level number is 2.
11–12	The DL/I status code is contained in these bytes of the PCBF= variable. The status code can also be obtained by specifying the STATUS= option.
13–16	Bytes 13–16 contain the DL/I processing options defined for this PCB in the PSBGEN with the PROCOPT= parameter.
17–24	These bytes contain the name of the last segment accessed. (Normally, the reserved area of the PCB mask occupies bytes 17–20, but the reserved data has been removed.) Consider the example for the level number of data in bytes 9–10. In that example there are SSAs for CUSTOMER, CHCKACCT, and CHCKCRDT segments; however, only the SSAs for CUSTOMER and CHCKACCT are satisfied. Since CHCKACCT is the last segment accessed, these bytes contain a value of CHCKACCT. The name of the last segment accessed can also be obtained from the variable specified by the SEGMENT= option.
25–28	The length of the key feedback data is contained in these bytes in IB4. format. The key feedback data is described in this table under bytes 33–200.
29-32	The number of sensitive segments in the PCB is contained in these bytes in IB4. format. For example, if you use a PCB that defines CUSTOMER and SAVEACCT as sensitive segments, these bytes contain a value of 2.
33–200	The *key feedback data* is contained in bytes 33–200. Key feedback data consists of the key field of the last segment accessed and the key field of each segment along the path to the last segment. This is also called the *concatenated key*. For example, if you issue a GN call qualified with SSAs for the CUSTOMER and CHCKACCT segments, the concatenated key consists of the values from the SSNUMBER field of the CUSTOMER segment and the ACNUMBER field of the CHCKACCT segment. The maximum length of the PCBF= variable is 200. Since 32 of the 200 bytes are used by other data from the PCB mask, the maximum length of the key feedback data in the PCBF= variable is 168 bytes. If the length of the concatenated key is greater than 168 bytes, the data is truncated. (However, the value in bytes 25–28 reflects the actual length, not the truncated length.)

SEGMENT=*variable*

specifies a SAS *variable* that contains the name of the last segment accessed by the DL/I call. The specified variable is a character variable with a default length of 8.

If the DL/I call is qualified (that is, if one or more SSAs are used), the name of the lowest-level segment encountered that satisfied a qualification of the call is returned. For example, assume that a GN call is issued with these two SSAs:

```
SAVEACCT*D-(ACNUMBER =345620145345)
SAVECRDT(CRDTDATE =033195)
```

If a SAVEACCT segment is encountered with the correct value for ACNUMBER but there is no segment with the correct CRDTDATE, then the value SAVEACCT is returned to the SEGMENT= variable.

If the call is unqualified (no SSAs used), the name of the retrieved segment is returned. This information can be useful in sequential-access programs with more than one sensitive segment type. For example, assume that a program employs a PCB that is sensitive to the CUSTOMER, CHCKACCT, and CHKCRDT segments and issues unqualified calls. You can specify the SEGMENT= option so that the name of the returned segment is available.

If SEGMENT= is not specified, the last segment's name is not returned to the program unless the PCBF= option is used.

SSA=*variable*
SSA=(*variable, variable,...*)

specifies from 1 to 15 SAS *variables* that contain values used as DL/I SSAs for the calls executed by DL/I INPUT or DL/I PUT statements. Each SSA= variable value must be entered in capital letters and must be assigned a complete DL/I SSA value (qualified or unqualified) or be set to blanks prior to the execution of the DL/I INPUT or DL/I PUT statement. Each SSA= variable value must be character and must be assigned a length (for example, with a LENGTH statement) prior to execution of the DL/I INFILE statement. The minimum length of an SSA variable is 9 bytes, and the maximum length is 200 bytes.

The value of an SSA= variable can be changed between calls.

SSA= variables must be character variables, but you can qualify an SSA with data from a numeric field in a segment. In this case, you can use the PUT function to insert a numeric value into an SSA= variable. See "SSAs in IMS DATA Step Programs" on page 235 for more information.

If SSA= is not specified, SSAs are not used in any DL/I call in the DATA step.

STATUS=*variable*

names a SAS *variable* to which the DL/I status code is assigned after each DL/I call. The variable is a character variable with a length of 2. This option provides a convenient way to check status codes, for example, when you are writing a random-access program and need to check for the end-of-file condition. (See "Checking Status Codes" on page 174 for more information about checking status codes in IMS DATA step programs.)

If STATUS= is not specified, status codes are not returned to the program unless the PCBF= option is used.

The following standard INFILE statement options can also be specified in a DL/I INFILE statement:

EOF=*label*

specifies a statement *label* that is the object of an implicit GO TO when the input file reaches an end-of-file condition in a sequential-access IMS DATA step program. Random-access programs do not cause the end-of-file condition to be set and, thus, do not execute this option. In random-access programs, you must check

the status code variable for a value of **GB** (end-of-file) and explicitly branch to the labeled statements.

OBS=*n*

specifies the last line to be read from the INFILE. In an IMS DATA step program, *n* specifies the maximum number of DL/I get calls to execute.

START=*variable*

defines the starting column of the input buffer when you use the _INFILE_ specification in a DL/I PUT statement.

STOPOVER

stops processing if the segment returned to the input buffer does not contain values for all variables that are specified in the DL/I INPUT statement.

Refer to *SAS Language Reference: Dictionary* for complete descriptions of these options. Note that EOF=, OBS=, START=, and STOPOVER are the only standard INFILE options that can be specified in a DL/I INFILE statement.

One other standard INFILE statement option, the MISSOVER option, is the default for DL/I INFILE statements and does not have to be specified. The MISSOVER option prevents SAS from reading past the current segment data in the input buffer if values for all variables specified by the DL/I INPUT statement are not found. Variables for which data is not found are assigned missing values. Without the default action of the MISSOVER option, SAS would issue another get call when values for some variables are missing.

Table 8.3 on page 169 summarizes the DLI INFILE statement options and other options that affect the DATA step interface to IMS, and it also describes the purpose of each option along with its default value and any additional comments.

Using the DL/I INFILE Statement

You can have more than one input source in a DATA step; for example, you can read from a DL/I database and a SAS data set in the same DATA step. If you want to use several external files (data sets other than SAS data sets) in a DATA step, use separate INFILE statements for each source. The input source is set (or reset) whenever an INFILE statement is executed. The file or DL/I PSB referenced in the most recently executed INFILE statement is the *current input source* for INPUT statements. The current input source does not change until a different INFILE statement executes, regardless of the number of INPUT statements executed.

When you change input sources by executing multiple INFILE statements and you want to return to an earlier input source, it is not necessary to repeat all options specified in the original INFILE statement. SAS remembers options from the first INFILE statement with the same fileref or PSB name. In a standard INFILE statement it is sufficient to specify only the fileref; in a DL/I INFILE, specify DLI and the PSB. Options specified in a previous INFILE statement with the same fileref or PSB name cannot be altered.

Note: The PSB name cannot be the same name as a fileref on a JCL DD statement or TSO ALLOC, or a filename's fileref. △

Table 8.3 Summary of DL/I INFILE Statement Specifications and Options

Option	Purpose	Default	Comments
CALL= *variable*	specifies variable containing call function	**GN** (get-next)	required to change call function from default
DBNAME= *variable*	specifies which eligible database PCB to use	n/a	overrides PCB= option if variable value is nonblank
DLI	indicates DL/I resource is data source	n/a	required; must follow PSB name
FSARC= *variable*	specifies variable containing FSA status codes	n/a	MVS IMS/VS Fast Path FLD calls only
LENGTH= *variable*	specifies variable containing length of returned segment(s)	n/a	
LRECL= *length*	specifies length of I/O buffers	1000 bytes	if too short, unpredictable results might occur
PCB= *variable*	specifies variable containing numeric index to choose eligible PCB	n/a	
PCBF= *variable*	specifies variable containing PCB feedback data	n/a	
PCBNO=*n*	defines first eligible PCB	1	
PSBname	specifies PSB to use	n/a	required; must follow INFILE keyword; cannot match active fileref or DDname
SEGMENT= *variable*	specifies variable containing last segment accessed	n/a	segment name also available through PCBF= variable
SSA= *variable* or (*variable*, *variable*,. . .)	specifies 1 to 15 variables containing SSAs	n/a	must have length defined prior to INFILE execution
EOF= *label*	specifies label for subroutine executed at end-of-file	n/a	for sequential access only
MISSOVER	assigns missing values for missing data	yes	forced for DL/I INFILE, does not have to be specified
OBS=*n*	specifies maximum number of get calls	n/a	
START= *variable*	specifies variable containing start column for _INFILE_	n/a	
STOPOVER	stops processing if some variable values missing	n/a	

Consider this DATA step:

```
filename employ '<your.sas.employ>' disp=shr;
data test (drop = socsec);
   ssa1 = 'CUSTOMER ';
   func = 'GN  ';
   infile acctsam dli call=func
      ssa=ssa1 pcbno=3 status=st;
   input @1   soc_sec_number $char11.
         @12  customer_name  $char40.
```

```
              @82  addr_line_2     $char30.
              @112 city           $char28.
              @140 state          $char2.
              @162 zip_code       $char10.
              @172 home_phone     $char12.;
          if st ¬= ' ' then
              link abendit;

      prt = 0;
      do until (soc_sec_number = socsec);
          infile employ ls=53 ;
          input @1  socsec    $11.
                @13 employer $3.;
          if soc_sec_number = socsec then
              do until (st = 'GE');
                  infile acctsam dli;
                  func = 'GNP ';
                  ssa1 = 'SAVEACCT ';
                  input @1  savings_account_number 12.
                        @13 savings_amount      pd5.2
                        @18 savings_date        mmddyy6.
                        @26 savings_balance     pd5.2;
                  if st = ' ' then
                     do;
                         output test;
                         prt = 1;
                     end;
                  else
                  if st = 'GE' then
                     do;
                         _error_ = 0;
                         if prt = 0 then
                            output test;
                     end;
                  else
                     link abendit;
              end;
          end;
      return;

      abendit:
          file log;
          put _all_;
          abort;
run;

proc print data=test;
   title2 '2 Files Combined';
run;

filename employ clear;
```

The input source for the first INPUT statement is the DL/I PSB called ACCTSAM. When the second INFILE statement is executed, an external file referenced by the fileref EMPLOY becomes the current input source for the next INPUT statement. Then,

the input source switches back to the ACCTSAM PSB after **soc_sec_number = socsec**. Notice the entire DL/I INFILE statement is not repeated; only the PSBname and DLI are specified.

Remember that only one PSB can be used in a given DATA step, although that PSB can be referenced in multiple INFILE statements.

Since the IMS database is being processed sequentially, the DATA step will terminate as soon as either a **GB** status is returned from IMS or an end-of-file is encountered when processing file EMPLOY.

Note: For the purposes of this example, the data in the EMPLOY file is in the same order as the HDAM database used in the example and there is a one-to-one correspondence between the values of SOC_SEC_NUMBER and SOCSEC. △

The following output shows the results of this example.

Output 8.4 Results of Using Multiple Input Sources in an IMS DATA Step

```
                              The SAS System
                             2 Files Combined


          soc_sec_
   OBS     number     customer_name         addr_line_2          city              state

    1    667-73-8275  WALLS, HOOPER J.     4525 CLARENDON RD    RAPIDAN            VA
    2    434-62-1234  SUMMERS, MARY T.     4322 LEON ST.        GORDONSVILLE       VA
    3    436-42-6394  BOOKER, APRIL M.     9712 WALLINGFORD PL. GORDONSVILLE       VA
    4    434-62-1224  SMITH, JAMES MARTIN  133 TOWNSEND ST.     GORDONSVILLE       VA
    5    434-62-1224  SMITH, JAMES MARTIN  133 TOWNSEND ST.     GORDONSVILLE       VA
    6    178-42-6534  PATTILLO, RODRIGUES  9712 COOK RD.        ORANGE             VA
    7    156-45-5672  O'CONNOR, JOSEPH     235 MAIN ST.         ORANGE             VA
    8    657-34-3245  BARNHARDT, PAMELA S. RT 2 BOX 324         CHARLOTTESVILLE    VA
    9    667-82-8275  COHEN, ABRAHAM       2345 DUKE ST.        CHARLOTTESVILLE    VA
   10    456-45-3462  LITTLE, NANCY M.     4543 ELGIN AVE.      RICHMOND           VA
   11    234-74-4612  WIKOWSKI, JONATHAN S. 4356 CAMPUS DRIVE   RICHMOND           VA


                                                    savings_
                                                    account_    savings_    savings_    savings_
   OBS    zip_code    home_phone     prt   employer  number      amount       date       balance

    1    22215-5600  803-657-3098    0     AAA      459923888253  784.29      12870       672.63
    2    26001-0670  803-657-1687    0     NBC      345689404732 8406.00      12869      8364.24
    3    26001-0670  803-657-1346    0     CTG      144256844728  809.45      12863      1032.23
    4    26001-0670  803-657-3437    0     CBS      345689473762  130.64      12857       261.64
    5    26001-0670  803-657-3437    1     CBS      345689498217 9421.79      12858      9374.92
    6    26042-1650  803-657-1346    0     UMW      345689462413  950.96      12857       946.23
    7    26042-1650  803-657-5656    0     AFL      345689435776  136.40      12869       284.97
    8    25804-0997  803-345-4346    0     ITT      859993641223  845.35      12860      2553.45
    9    25804-0997  803-657-7435    0     IBM      884672297126  945.25      12868       793.25
   10    26502-3317  803-657-3566    0     SAS      345689463822  929.24      12867       924.62
   11    26502-5317  803-467-4587    0     UNC             .         .           .            .
```

The DL/I INPUT Statement

If you are unfamiliar with the INPUT statement, refer to *SAS Language Reference: Dictionary* for more information.

An INPUT statement reads from the file that is specified by the most recently executed INFILE statement. If the INFILE statement is a DL/I INFILE statement, the INPUT statement issues a DL/I get call and retrieves a segment or segments.

There are no special options for the DL/I INPUT statement as there are for the DL/I INFILE statement. The form of the DL/I INPUT statement is the same as that of the standard INPUT statement:

```
input variable optional-specifications;
```

For example, suppose you are issuing a qualified get call for the CUSTOMER segment. The DL/I INPUT statement might be coded like this:

```
input @1    soc_sec_number  $char11.
      @12   customer_name   $char40.
      @52   addr_line_1     $char30.
      @82   addr_line_2     $char30.
      @112 city             $char28.
      @140 state            $char2.
      @142 country          $char20.
      @162 zip_code         $char10.
      @172 home_phone       $char12.
      @184 office_phone     $char12.;
```

When this DL/I INPUT statement executes, DL/I retrieves a CUSTOMER segment and places it in the input buffer. Data for the variables specified in the DL/I INPUT statement is then moved from the input buffer to SAS variables in the program data vector by SAS.

Different forms of the INPUT statement can have different results:

□ When an INPUT statement specifies variable names (as in the previous example), the segment is usually retrieved and placed in the input buffer and the values are moved immediately to SAS variables in the program data vector unless this form of the INPUT statement is preceded by an INPUT statement with a trailing @ sign, for example, **input@**. The INPUT statement with a trailing @ sign is described below.

□ If the INPUT statement does not specify any variable names or options, as in this example:

```
input;
```

a segment or segments are retrieved by the call and placed in the input buffer but no data is mapped to the program data vector. Or, if the previous INPUT statement was **input@**, this clears the hold.

□ If the INPUT statement does not specify variable names but does have a trailing @:

```
input @;
```

a call is issued and one or more segments are retrieved and placed in the input buffer. The trailing @ tells SAS to use the data just placed in the input buffer when the next DL/I INPUT statement in that execution of the DATA step is executed. In other words, the trailing @ tells SAS *not* to issue another call the next time a DL/I INPUT statement is executed. Instead, SAS uses the data that is already in the input buffer. This form of the INPUT statement is very useful in IMS DATA step programs. Refer to "Using the DL/I INPUT Statement" on page 174 for more information.

□ You can combine the form that names variables with the form that uses a trailing @. In this example, a call is issued, a segment is retrieved and placed in the input buffer, and values for the named variables are moved to SAS variables in the program data vector:

```
input soc_sec_number $char11. @;
```

Because of the trailing @, SAS holds the segment in the input buffer for the next INPUT statement.

Although the syntax of the DL/I INPUT statement and the standard INPUT statement are the same, your use of the DL/I INPUT statement is often different. Suggested uses of the DL/I INPUT statement are discussed in "Using the DL/I INPUT Statement" on page 174.

Example 1: A Get Call

The following DATA step illustrates how to issue get calls using the DL/I INFILE and DL/I INPUT statements:

```
data custchck;
   retain ssa1 'CUSTOMER*D '
          ssa2 'CHCKACCT ';
   infile acctsam dli ssa=(ssa1,ssa2) status=st
          pcbno=3;
   input @1   soc_sec_number        $char11.
         @12  customer_name         $char40.
         @52  addr_line_1           $char30.
         @82  addr_line_2           $char30.
         @112 city                  $char28.
         @140 state                 $char2.
         @142 country               $char20.
         @162 zip_code              $char10.
         @172 home_phone            $char12.
         @184 office_phone          $char12.
         @226 check_account_number  $char12.
         @238 check_amount          pd5.2
         @243 check_date            mmddyy6.
         @251 check_balance         pd5.2;
   if st ¬= '  ' then
      do;
         file log;
         put _all_;
         abort;
      end;
run;

proc print data=custchck;
   title2 'Customer Checking Accounts';
run;
```

This DATA step creates a SAS data set, CUSTCHCK, with one observation for each checking account in the ACCTDBD database. To build the data set, the program issues qualified get-next path calls using unqualified SSAs for the CUSTOMER and CHCKACCT segments. The path call is indicated by the *D command code in the CUSTOMER SSA, SSA1. The PCBNO= option specifies the first eligible PCB that permits path calls for the CUSTOMER segment of the ACCTDBD database.

The DL/I INFILE statement points to the ACCTSAM PSB and specifies two SSA variables, SSA1 and SSA2. The SSA variables have already been assigned values and lengths by the preceding RETAIN statement. Since these SSAs are not qualified, the program access is sequential. In this get call, the status code is checked and the third PCB is specified. Defaults are in effect for the other DL/I INFILE options: only get-next

calls are issued, the input buffer length is 1000 bytes, and segment names and PCB mask data are not returned.

When the DL/I INPUT statement executes and **status = ' '**, the qualified GN call is issued, the concatenated CUSTOMER and CHCKACCT segments are placed in the input buffer, and data from both segments are moved to SAS variables in the program data vector.

The following output shows the results of this example.

Output 8.5   Results of Issuing Get Calls

```
                                    The SAS System
                              Customer Checking Accounts


         soc_sec_                             addr_
   OBS    number       customer_name         line_1      addr_line_2         city              state

    1   667-73-8275   WALLS, HOOPER J.                   4525 CLARENDON RD    RAPIDAN           VA
    2   667-73-8275   WALLS, HOOPER J.                   4525 CLARENDON RD    RAPIDAN           VA
    3   434-62-1234   SUMMERS, MARY T.                   4322 LEON ST.        GORDONSVILLE      VA
    4   436-42-6394   BOOKER, APRIL M.                   9712 WALLINGFORD PL. GORDONSVILLE      VA
    5   434-62-1224   SMITH, JAMES MARTIN               133 TOWNSEND ST.     GORDONSVILLE      VA
    6   434-62-1224   SMITH, JAMES MARTIN               133 TOWNSEND ST.     GORDONSVILLE      VA
    7   178-42-6534   PATTILLO, RODRIGUES               9712 COOK RD.        ORANGE            VA
    8   156-45-5672   O'CONNOR, JOSEPH                  235 MAIN ST.         ORANGE            VA
    9   657-34-3245   BARNHARDT, PAMELA S.              RT 2 BOX 324         CHARLOTTESVILLE   VA
   10   667-82-8275   COHEN, ABRAHAM                    2345 DUKE ST.        CHARLOTTESVILLE   VA
   11   456-45-3462   LITTLE, NANCY M.                  4543 ELGIN AVE.      RICHMOND          VA
   12   234-74-4612   WIKOWSKI, JONATHAN S.             4356 CAMPUS DRIVE    RICHMOND          VA


                                                        check_
                                                        account_       check_     check_    check_
   OBS   country   zip_code     home_phone    office_phone   number     amount     date     balance

    1    USA     22215-5600   803-657-3098  803-645-4418  345620145345  1702.19   12857     1266.34
    2    USA     22215-5600   803-657-3098  803-645-4418  345620154633  1303.41   12870     1298.04
    3    USA     26001-0670   803-657-1687                345620104732   826.05   12869      825.45
    4    USA     26001-0670   803-657-1346                345620135872   220.11   12868      234.89
    5    USA     26001-0670   803-657-3437                345620134564  2392.93   12858     2645.34
    6    USA     26001-0670   803-657-3437                345620134663     0.00   12866      143.78
    7    USA     26042-1650   803-657-1346  803-657-1345  745920057114  1404.90   12944     1502.78
    8    USA     26042-1650   803-657-5656  803-623-4257  345620123456   353.65   12869      463.23
    9    USA     25804-0997   803-345-4346  803-355-2543  345620131455  1243.25   12871     1243.25
   10    USA     25804-0997   803-657-7435  803-645-4234  382957492811  7462.51   12876     7302.06
   11    USA     26502-3317   803-657-3566                345620134522   608.24   12867      831.65
   12    USA     26502-5317   803-467-4587  803-654-7238  345620113263   672.32   12870       13.28
```

Refer to "Example 6: Issuing Path Calls" on page 184 later in this section for a detailed explanation of a sample IMS DATA step program that includes a similar DATA step.

Using the DL/I INPUT Statement

When a DL/I INPUT statement is executed, a DL/I get call is issued as formatted by variables specified in the DL/I INFILE statement.

Checking Status Codes

A get call might or might not successfully retrieve the requested segments. For each call issued, DL/I returns a status code that indicates whether or not the call was successful. Since the success of a call can affect the remainder of the program, it is a good idea to check status codes, especially in programs that use random access. You can

obtain the status code returned by DL/I with the STATUS= option or the PCBF= option of the DL/I INFILE statement. Refer to your IBM documentation for explanations of DL/I status codes.

In general, a call has been successful and the segment(s) has been obtained if the automatic SAS variable _ERROR_ has a value of zero. This corresponds to a blank DL/I return code, or codes of **CC**, **GA**, or **GK**. SAS sets _ERROR_ to 1 if any other DL/I status code is returned or if the special SAS status code **SE** is returned. (The **SE** code is generated when SAS cannot format a proper DL/I call from the options specified.) If _ERROR_ is set to 1, the contents of the input buffer and the program data vector are printed on the SAS log when another INPUT statement is executed or when control returns to the beginning of the DATA step, whichever comes first.

Some of the DL/I status codes that set _ERROR_ might not be errors to your SAS program. When this is the case, you should check the actual return code as well as the value of _ERROR_. For example, suppose you are writing a program that looks for a segment with a particular value for a sequence field. If the segment is found, a replace call (REPL) is issued to update the segment. If the segment is not found, _ERROR_ is set to 1, but you do not consider the status code to be an error. Instead, you issue an insert call (ISRT) to add a new segment.

If a status code sets _ERROR_ but you do not consider the status code to be an error, you should reset _ERROR_ to zero prior to executing another INPUT or PUT statement or returning to the beginning of the DATA step. Otherwise, the contents of the input buffer and program data vector are printed on the SAS log.

Use of the Trailing @

You can use different forms of the DL/I INPUT statement to perform these general functions:

- □ issue a DL/I get call
- □ place the retrieved segment in the input buffer
- □ move data from the input buffer to SAS variables in the program vector if variables are named in the INPUT statement.

In some programs, it is important to check the values of the _ERROR_ or STATUS= variables *before* moving data from the input buffer to SAS variables in the program data vector. For example, if a get call fails to retrieve the expected segment, the input buffer might still contain data from a previous get call or be filled with missing values. You might not want to move these values to SAS variables. By checking the STATUS= or _ERROR_ variable, you determine whether or not the call was successful and can decide whether or not to move the input buffer data to SAS variables.

Similarly, if you issue unqualified get calls with a PCB that is sensitive to more than one segment type, you might need to know what type of segment was retrieved in order to move data to the appropriate SAS variables.

When you want to issue a get call but you need to check _ERROR_ or STATUS= variable values before moving data to SAS variables, use a DL/I INPUT statement with a trailing @ to issue the call:

```
input @;
```

The trailing @ pointer control causes SAS to hold the current record (segment) in the input buffer for the next DL/I INPUT statement. The next DL/I INPUT statement to be executed does not issue another call and does not place a new segment in the input buffer. Instead, the second INPUT statement uses the data placed in the input buffer by the first INPUT statement.

If no variables are named in the first DL/I INPUT statement (as in the statement shown in the previous paragraph), data is not moved from the buffer to SAS variables until another DL/I INPUT statement specifying the variables is executed. Therefore,

before executing a second INPUT statement, you can check the value of the STATUS= or PCBF= variable to determine whether or not the call was successful. You can also check the _ERROR_ automatic variable and the SEGMENT= variable. After checking these values, execute a second DL/I INPUT statement to move data to SAS variables, if appropriate.

Example 2: Using the Trailing @

This example demonstrates the use of the trailing @. This DATA step creates two SAS data sets, CHECKING and SAVINGS, from data in the CHCKACCT and SAVEACCT segments of the ACCTDBD database. The PCB that is used defines CUSTOMER, CHCKACCT, and SAVEACCT as sensitive segments. Since no CALL= or SSA= variables are specified, all calls are unqualified get-next calls, and access is sequential.

Each call is issued by a DL/I INPUT statement with a trailing @, so the retrieved segment is placed in the buffer and held there. Two variables are checked: ST and SEG (the SEGMENT= variable). If a call results in an error, the job terminates. If a call is successful, the program checks SEG to determine the type of the retrieved segment. Because this is a sequential access program, a **GB** (end-of-file) status code is not treated as an error by the program. Therefore, the program resets _ERROR_ to 0.

When SEG='CUSTOMER', execution returns to the beginning of the DATA step. When the SEG value is CHCKACCT or SAVEACCT, another DL/I INPUT statement moves the data to SAS variables in the program data vector, and the observation is written to the appropriate SAS data set.

```
data checking savings;
   infile acctsam dli segment=seg status=st
       pcbno=3;
   input @;
   if st ¬= '  ' and
      st ¬= 'CC' and
      st ¬= 'GA' and
      st ¬= 'GK' then
         do;
            file log;
            put _all_;
            abort;
         end;
   if seg = 'CUSTOMER' then
     return;
   else
     do;
       input @1  account_number $char12.
             @13 amount          pd5.2
             @18 date            mmddyy6.
             @26 balance         pd5.2;
       if seg = 'CHCKACCT' then
         output checking;
       else
         output savings;
     end;
run;

proc print data=checking;
   title2 'Checking Accounts';
```

```
    run;

    proc print data=savings;
        title2 'Savings Accounts';
    run;
```

The following output shows the results of this example:

Output 8.6   Results of Using the Trailing @

```
                        The SAS System
                      Checking Accounts

                    account_
            OBS      number        amount     date     balance

             1     345620145345    1702.19    12857    1266.34
             2     345620154633    1303.41    12870    1298.04
             3     345620104732     826.05    12869     825.45
             4     345620135872     220.11    12868     234.89
             5     345620134564    2392.93    12858    2645.34
             6     345620134663       0.00    12866     143.78
             7     745920057114    1404.90    12944    1502.78
             8     345620123456     353.65    12869     463.23
             9     345620131455    1243.25    12871    1243.25
            10     382957492811    7462.51    12876    7302.06
            11     345620134522     608.24    12867     831.65
            12     345620113263     672.32    12870      13.28


                        The SAS System
                      Savings Accounts

                    account_
            OBS      number        amount     date     balance

             1     459923888253     784.29    12870     672.63
             2     345689404732    8406.00    12869    8364.24
             3     144256844728     809.45    12863    1032.23
             4     345689473762     130.64    12857     261.64
             5     345689498217    9421.79    12858    9374.92
             6     345689462413     950.96    12857     946.23
             7     345689435776     136.40    12869     284.97
             8     859993641223     845.35    12860    2553.45
             9     884672297126     945.25    12868     793.25
            10     345689463822     929.24    12867     924.62
```

Note: If the DL/I call is issued by a DL/I INPUT statement with a trailing @ and the status code sets _ERROR_, but you do not consider the status code to be an error and you want to issue another get call in the same execution of the DATA step, then you must first execute a blank DL/I statement:**input;** △

The blank DL/I INPUT statement clears the input buffer. If the buffer is not cleared by issuing a blank INPUT statement, the next DL/I INPUT statement assumes that the data to be retrieved is already in the buffer and does not issue a DL/I call. See "Example 8: Using the Blank INPUT Statement" on page 190 for an example that includes a blank INPUT statement.

The DL/I FILE Statement

If you are unfamiliar with the FILE statement, refer to *SAS Language Reference: Dictionary* for more information.

The FILE statement identifies an external file to which information specified by a PUT statement is written. In an IMS DATA step, the DL/I FILE statement specifies a PSB, which in turn identifies a DL/I database or message queue to be accessed by a DL/I update call. The call is formatted using the values and variables specified in the DL/I INFILE statement, which must precede the DL/I FILE statement in the DATA step. The update call is issued when the corresponding DL/I PUT statement is executed. In other words, to issue an update call you use a DL/I INFILE, DL/I FILE, and DL/I PUT statement.

The following is the form of the DL/I FILE statement:

```
FILE PSBname DLI;
```

PSBname
    specifies the same PSB referenced in the DATA step's DL/I INFILE statement. Refer to "The DL/I INFILE Statement" on page 162 for more information. A PSB name must be specified.

DLI
    tells SAS that the output file is a DL/I database or message queue. DL/I must be specified and must be after the PSB name.

No other options (including standard FILE statement options) are recognized in the DL/I FILE statement.

The DL/I FILE statement references a PSB that identifies a database or message queue to which a corresponding DL/I PUT statement writes.

The most recently executed FILE statement determines the *current output file*. If you are using more than one output file in a DATA step, there must be a FILE statement for each file. Change the current output file from one to another by executing a different FILE statement. To return to the original output file, repeat the original FILE statement. The current output file does not change until a new FILE statement executes, regardless of the number of PUT statements executed.

The DL/I PUT Statement

If you are unfamiliar with the PUT statement, refer to *SAS Language Reference: Dictionary* for more information.

A PUT statement writes information to the file specified by the most recently executed FILE statement. If the FILE statement is a DL/I FILE statement, the corresponding PUT statement issues a DL/I update call.

There are no special options for a DL/I PUT statement as there are for the DL/I INFILE and DL/I FILE statements. The form of the DL/I PUT statement is the same as that of the standard PUT statement:

```
PUT variable optional-specifications;
```

For example, assume that you are issuing an insert call for the CUSTOMER segment of the ACCTDBD database. The following DL/I PUT statement (which looks just like a standard PUT statement) formats a CUSTOMER segment and issues the ISRT call:

```
put @1    ssnumber        $char11.
    @12   custname        $char40.
    @52   addr_line_1     $char30.
    @82   addr_line_2     $char30.
    @112 custcity         $char28.
    @140 custstat         $char2.
    @142 custland         $char20.
    @162 custzip          $char10.
    @172 h_phone          $char12.
    @184 o_phone          $char12.;
```

Although the syntax of the DL/I PUT statement is identical to that of the standard PUT statement, your use of the DL/I PUT is often different. Segment format and suggested uses of the DL/I PUT statement are discussed in "Using the DL/I PUT Statement" on page 180.

Example 3: An Update Call

This DATA step reads MYDATA.CUSTOMER, an existing SAS data set containing information about new customers, and updates the ACCTDBD database with the data in the SAS data set:

```
data _null_;
   set mydata.customer;
   length ssa1 $9;
   infile acctsam dli call=func ssa=ssa1
      status=st pcbno=4;
   file acctsam dli;
   func = 'ISRT';
   ssa1 = 'CUSTOMER';
   put @1    ssnumber        $char11.
       @12   custname        $char40.
       @52   addr_line_1     $char30.
       @82   addr_line_2     $char30.
       @112 custcity         $char28.
       @140 custstat         $char2.
       @142 custland         $char20.
       @162 custzip          $char10.
       @172 h_phone          $char12.
       @184 o_phone          $char12.;
   if st ¬= ' ' then
      if st = 'LB' or st = 'II' then
         _error_ = 0;
      else
         do;
            file log;
            put _all_;
            abort;
         end;
run;
```

To update ACCTDBD with new occurrences of the CUSTOMER segment type, this program issues qualified insert calls that add observations from MYDATA.CUSTOMER to the database. The DL/I INFILE statement defines ACCTSAM as the PSB. Options in the INFILE statement specify the following information:

 □ the SAS variable FUNC contains the call function

 □ PCBNO= specifies the database PCB to use

 □ SSA1 contains the SSA that specifies the segment name of the segment to be inserted

 □ STATUS= specifies where the status code is returned.

Defaults are in effect for the other DL/I INFILE options: the output buffer length is 1000 bytes, and segment names and PCB mask data are not returned.

   If the ISRT call is not successful, the status code variable ST is set with the DL/I status code and the automatic variable _ERROR_ is set to 1. After the ISRT call, the status code variable ST is checked for non-blanks. If the variable value is either **LB** or **II**, which indicate that the segment occurrence already exists, the automatic variable _ERROR_ is reset to 0 and processing continues. Otherwise, all values from the program data vector are written to the SAS log, and the DATA step aborts.

Using the DL/I PUT Statement

   A PUT statement writes data to the current output file, which is determined by the most recently executed FILE statement. A DL/I PUT statement writes to a DL/I database or message queue by issuing a DL/I update call. If you are unfamiliar with the PUT statement, refer to *SAS Language Reference: Dictionary* for more information.

   In order for a DL/I update call to be executed, the CALL= option must be specified in the DL/I INFILE statement. The value of the CALL= variable must be set to the appropriate update call before the DL/I PUT statement is executed. If CALL= is not specified, the call function defaults to GN and no update calls can be issued.

   The update call issued by a DL/I PUT statement might or might not be successful. DL/I returns various status codes that indicate whether or not the update call was successful. It is always a good idea to check the status code, but it is especially important in an update program. If you are unfamiliar with DL/I status codes, consult your IBM documentation for descriptions. Your SAS program can obtain the return code if the STATUS= option of the INFILE statement is specified. The _ERROR_ and STATUS= variable checking guidelines discussed in "Using the DL/I INPUT Statement" on page 174 are also applicable to DL/I PUT statements.

REPL Call

   When you replace a segment (REPL call) with a DL/I PUT statement, you must place the entire segment in the output buffer, even if all fields are not being changed.

   One way the buffer can be formatted is by specifying all fields and their locations. For example, this DL/I PUT statement formats the entire CUSTOMER segment of the ACCTDBD database:

```
put @1    ssnumber          $char11.
    @12   custname          $char40.
    @52   addr_line_1       $char30.
    @82   addr_line_2       $char30.
    @112  custcity          $char28.
    @140  custstat          $char2.
    @142  custland          $char20.
    @162  custzip           $char10.
    @172  h_phone           $char12.
    @184  o_phone           $char12.;
```

   Another way to format the output buffer is with the _INFILE_ specification. If the current input source is a DL/I INFILE and the last DL/I INPUT statement retrieved

the DL/I segment to be replaced, then the following DL/I PUT statement formats the output buffer with the contents of the retrieved segment and holds the segment in the output buffer until another DL/I PUT statement is executed:

```
put _infile_ @;
```

A subsequent DL/I PUT statement can modify the data in the output buffer and execute the REPL call.* Example 4 illustrates this technique.

Example 4: Issuing REPL Calls

In this example, CUSTOMER segments are updated with change-of-address information from a Version 6 SAS data set called MYDATA.NEWADDR. The Version 6 DATA step interface works exactly like the Version 7 and later DATA step interfaces, except that the Version 7 and later DATA step interfaces support SAS variable and member names of up to 32 characters. The interface will work as long as the SAS variable names specified in the DL/I INPUT statement match those specified in the DL/I PUT statement. Variables in this SAS data set are SSN (social security number), NEWADDR1, NEWADDR2, NEWCITY, NEWSTATE, and NEWZIP. After the CUSTOMER segment is retrieved, the PUT statement formatting the output buffer is issued. The segment is held in the output buffer until a second PUT statement is issued that executes a REPL call to update the CUSTOMER segment.

Notice that SSA1, a qualified SSA, is constructed by concatenating the SSA specification with the value of the SSN variable in the SAS data set. SSA1 is set to blanks after the GHU call because an SSA is not needed for the REPL call. (Since the program issues get calls with qualified SSAs, access is random.)

```
data _null_;
   set mydata.newaddr;
   length ssa1 $31;
   infile acctsam dli ssa=ssa1 call=func
      status=st pcbno=4;
   ssa1 = 'CUSTOMER(SSNUMBER =' || ssn || ')';
   func = 'GHU ';
   input;
   if st = '  ' then
      do;
         func = 'REPL';
         ssa1 = '  ';
         file acctsam dli;
         put _infile_ @;
         put @52 newaddr1  $char30.
             @82 newaddr2  $char30.
             @112 newcity  $char28.
             @140 newstate $char2.
             @162 newzip   $char10.;
         if st ¬= '  ' then
            link abendit;
      end;
   else
```

---

* The effect of a trailing @ in a DL/I PUT statement is slightly different from the effect of one in a DL/I INPUT statement. A trailing @ in a DL/I PUT statement causes data to be moved to the output buffer but does not issue the update call. Instead, the call is issued by the next DL/I PUT statement that does not terminate with a trailing @. In a DL/I INPUT statement with a trailing @, the get call is issued, and data is moved to the input buffer. The next DL/I INPUT statement can then move data to the program data vector.

```
            if st = 'GE' then
               _error_ = 0;
            else
               link abendit;
      return;

      abendit:
         file log;
         put _all_;
         abort;
   run;
```

Alternatively, the two DL/I PUT statements can be combined into one without the trailing @ sign. For example:

```
data _null_;
   set mydata.newaddr;
   length ssa1 $31;
   infile acctsam dli ssa=ssa1 call=func
      status=st pcbno=4;
   ssa1 = 'CUSTOMER(SSNUMBER ='||ssn||')';
   func = 'GHU  ';
   input;
   if st = '  ' then
      do;
         func = 'REPL';
         ssa1 = '  ';
         file acctsam dli;
         put @1  _infile_
            @52 newaddr1 $char30.
            @82 newaddr2 $char30.
            @112 newcity $char28.
            @140 newstate $char2.
            @162 newzip  $char10.;
         if st ¬= '  ' then
            link abendit;
      end;
   else
      if st = 'GE' then
         _error_ = 0;
      else
         link abendit;
   return;

   abendit:
      file log;
      put _all_;
      abort;
run;
```

DLET Call

When issuing a delete call (DLET), DL/I requires that the sequence field of the segment be formatted in the output buffer. The DL/I PUT statement can explicitly format the sequence field. Alternatively, if the current INFILE is a DL/I INFILE and the last DL/I INPUT statement retrieved the DL/I segment to be deleted, then the

following SAS statement formats the output buffer with the contents of the retrieved segment (including the sequence field) and executes the DLET call:

```
put _infile_;
```

"Example 5: Issuing DLET Calls" on page 183 demonstrates this technique.

Example 5: Issuing DLET Calls

The following example deletes all WIRETRAN segments with a transaction date of 03/31/95:

```
data _null_;
   length ssa1 $30;
   retain db 'WIRETRN '  ;
   infile acctsam dli call=func dbname=db
      ssa=ssa1 status=st;
   func = 'GHN ';
   ssa1 = 'WIRETRAN(WIREDATE =03/31/95) ';
   input;
   if st = '  ' then
      do;
          file acctsam dli;
          func = 'DLET';
          ssa1 = '  ';
          put _infile_;
          if st ¬= '  ' then
             link abendit;
      end;
   else
      if st = 'GB' then
         do;
             _error_ = 0;
             stop;
         end;
      else
         link abendit;
   return;

   abendit:
      file log;
      put _all_;
      abort;
run;
```

Note: A check for a status code of **GB** is required in this DATA step because it uses a qualified SSA and random access processing. The DATA step does not set the end-of-file condition, and the source logic must check for it to stop the DATA step normally. △

IMS DATA Step Examples

Complete IMS DATA step examples are presented in this section. Each example illustrates one or more of the concepts described earlier in this section.

All of these examples are based on the sample databases, DBDs, and PSBs described in Appendix 2. If you have not read the sample database descriptions, you should do so before continuing this section.

It is assumed that the installation default values for IMS DATA step system options are the same as the default values described in Appendix 1. Statement options used in these examples that are not IMS DATA step statement extensions (for example, the HEADER= option in the FILE statement) are described in *SAS Language Reference: Dictionary*.

Example 6: Issuing Path Calls

This example produces a report that shows the distribution of checking account balances by ZIP code in the ACCTDBD database. SAS data set DISTRIBC is created from data in the CUSTOMER and CHCKACCT segments. The segments are retrieved with get-next calls using an unqualified SSA for the CUSTOMER segment with an *D command code and an SSA for the CHCKACCT segment. Thus, both the CUSTOMER and CHCKACCT segments are returned. The new SAS data set contains three variables: CHECK_AMOUNT (from the CHCKACCT segment), ZIPRANGE (created from the CUSTZIP value in the CUSTOMER segment), and BALRANGE (created from the BALANCE variable). The distribution information is produced by the TABULATE procedure from the DISTRIBC data set.

The numbered comments following this program correspond to the numbered statements in the program:

```
❶   data distribc;
❷       length ziprange $11;
❸       keep ziprange
              check_amount
              balrange;
❹       retain ssa1 'CUSTOMER*D '
              ssa2 'CHCKACCT ';
❺       infile acctsam dli ssa=(ssa1,ssa2) status=st
              pcbno=3;

❻       input @162 zip_code      $char10.
              @238 check_amount  pd5.2;
❼       if st ¬= '  ' and
            st ¬= 'CC' and
            st ¬= 'GA' and
            st ¬= 'GK' then
❽         if st = 'GE' then
              do;
                 _error_ = 0;
                  stop;
              end;
❾       else
          do;
              file log;
              put _all_;
❿         abort;
          end;

⓫       balrange=check_amount;
⓬       ziprange=substr(zip_code,1,4)
```

```
               ||'0-'||substr(zip_code,1,4)||'9';
          title 'Checking Account Balance Distribution
                  By ZIP Code';

⓭   proc format;
        value balrang
        low-249.99 = 'under $250'
        250.00-1000.00 = '$250 - $1000'
        1000.01-high = 'over $1000';

⓮   proc tabulate data=distribc;
        class ziprange balrange;
        var check_amount;
        label balrange='balance range';
        label ziprange='ZIP code range';
        format ziprange $char11. balrange balrang.;
        keylabel sum= '$ total' mean ='$ average'
          n='# of accounts';
        table ziprange*(balrange all),
              check_amount*(sum*f=14.2 mean*f=10.2 n*f=4);
     run;
```

❶   The DATA statement specifies DISTRIBC as the name of the SAS
           data set created by this DATA step.

❷   The length of the new variable ZIPRANGE is set.

❸   The new data set will contain only the three variables specified in
           the KEEP statement.

❹   The RETAIN statement specifies values for the two SSA variables,
           SSA1 and SSA2. SSA1 is an unqualified SSA for the CUSTOMER
           segment with the command code for a path call, *D. This command
           code means that the CUSTOMER segment is returned along with
           the CHCKACCT segment that is its child. SSA2 is an unqualified
           SSA for the CHCKACCT segment. Without the *D command code in
           SSA1, only the target segment, CHCKACCT, would be returned.

              These values are retained for each iteration of the DATA step.
           The RETAIN statement, which initializes the variables, satisfies the
           requirement that the length of an SSA variable be specified before
           the DL/I INFILE statement is executed.

❺   The INFILE statement specifies ACCTSAM as the PSB. The DLI
           specification tells SAS that the step will access DL/I resources. Two
           variables containing SSAs are identified by the SSA= option, SSA1
           and SSA2. Their values were set by the earlier RETAIN statement.
           The STATUS= option specifies the ST variable for status codes
           returned by DL/I. The PCBNO= option specifies which PCB to use.

              These defaults are in effect for the other DL/I INFILE options: all
           calls are get-next calls, the input buffer has a length of 1000 bytes,
           and the segment, and PCB mask data are not returned. No qualified
           SSAs are used; therefore, program access is sequential.

❻   The DL/I INPUT statement specifies positions and informats for the
           necessary variables in both the CUSTOMER and CHCKACCT

segments because the path call returns both segments. When this statement executes, the GN call is issued. If successful, CUSTOMER and CHCKACCT segments are placed in the input buffer and the ZIP_CODE and CHECK_AMOUNT values are then moved to SAS variables in the program data vector.

❼ If the qualified GN call issued by the DL/I INPUT statement is not successful (that is, it obtains any return code other than blank, **CC**, **GA**, or **GK**), the automatic SAS variable _ERROR_ is set to 1 and the DO group (statements 8 through 10) is executed.

❽ If the ST variable value is GE (a status code meaning that the segment or segments were not found), SAS stops execution of the DATA step. _ERROR_ is reset to 0 so that the contents of the input buffer and program data vector are not printed on the SAS log. This statement is included because of a DL/I feature. In a program issuing path calls, DL/I sometimes returns a GE status code when it reaches end-of-database. The GB (end-of-database) code is returned if another get call is issued after the GE code. Therefore, in this program, the GE code can be considered the end-of-file signal rather than an error condition.

❾ For any other non-blank status code, all values from the program data vector are written to the SAS log.

❿ The DATA step execution terminates and the job aborts.

⓫ If the qualified GN call is successful, BALRANGE is assigned the value of CHECK_AMOUNT.

⓬ The ZIPRANGE variable is created using the SUBSTR function with the ZIP_CODE variable.

⓭ PROC FORMAT is invoked to create a format for the BALRANGE variable. These formats are used in the PROC TABULATE output.

⓮ PROC TABULATE is invoked to process the DISTRIBC data set.

The following output shows the results of this example.

Output 8.7 Results of Issuing Path Calls

```
              Checking Account Balance Distribution By ZIP code

          ------------------------------------------------------------
          |                            |       check_amount           |
          |                            |------------------------------|
          |                            |          |          | # of   |
          |                            |          |          | acc-   |
          |                            |          |          | oun-   |
          |                            | $ total  | $ average| ts     |
          |----------------------------+----------+----------+----    |
          |ZIP code range |balance range |        |          |        |
          |---------------+------------- |        |          |        |
          |22210-22219    |over $1000    |  4410.50|  1470.17|    3   |
          |               |------------- |        |          |        |
          |               |All           |  4410.50|  1470.17|    3   |
          |---------------+--------------+----------+----------+----   |
          |25800-25809    |balance range |        |          |        |
          |               |------------- |        |          |        |
          |               |over $1000    |  8705.76|  4352.88|    2   |
          |               |------------- +----------+----------+----   |
          |               |All           |  8705.76|  4352.88|    2   |
          |---------------+--------------+----------+----------+----   |
          |26000-26009    |balance range |        |          |        |
          |               |------------- |        |          |        |
          |               |under $250    |   220.11|   110.06|    2   |
          |               |------------- +----------+----------+----   |
          |               |$250 - $1000  |   826.05|   826.05|    1   |
          |               |------------- +----------+----------+----   |
          |               |over $1000    |  2392.93|  2392.93|    1   |
          |               |------------- +----------+----------+----   |
          |               |All           |  3439.09|   859.77|    4   |
          |---------------+--------------+----------+----------+----   |
          |26040-26049    |balance range |        |          |        |
          |               |------------- |        |          |        |
          |               |$250 - $1000  |   353.65|   353.65|    1   |
          |               |------------- +----------+----------+----   |
          |               |All           |   353.65|   353.65|    1   |
          |---------------+--------------+----------+----------+----   |
          |26500-26509    |balance range |        |          |        |
          |               |------------- |        |          |        |
          |               |$250 - $1000  |  1280.56|   640.28|    2   |
          |               |------------- +----------+----------+----   |
          |               |All           |  1280.56|   640.28|    2   |
          ------------------------------------------------------------
```

Example 7: Updating Information in the CUSTOMER Segment

This example uses GHN calls to retrieve CUSTOMER segments and then tests the values of the STATE and COUNTRY fields. If a segment has a valid value for STATE but does not have COUNTRY='UNITED STATES', the COUNTRY value is changed to UNITED STATES and the corrected segment is replaced using a REPL call.

Follow the notes corresponding to the numbered statements in the following code for a detailed explanation of this example:

```
filename tranrept '<your.sas.tranrept>' disp=old;
data _null_;
❶ length ssa1 $ 9;
❷ infile acctsam dli ssa=ssa1 call=func pcbno=4
       status=st;
❸ func = 'GHN ';
❹ ssa1 = 'CUSTOMER';
```

```
❺ input @12   customer_name  $char40.
        @140  state          $char2.
        @142  country        $char20.;
❻ if st ⌐= ' ' and
      st ⌐= 'CC' and
      st ⌐= 'GA' and
      st ⌐= 'GK' then
    link abendit;
❼ if country ⌐= 'UNITED STATES' &
      state < 'Z ' &
      state > 'A ' then
    do;
❽      oldland = country;
❾      country = 'UNITED STATES';
❿      file acctsam dli;
⓫      func = 'REPL';
⓬      ssa1 = ' ';
⓭      put @1  _infile_
           @142 country;
⓮      if st ⌐= ' ' then
         link abendit;
⓯      file tranrept header=newpage notitles;
⓰      put @10 customer_name
           @60 state
           @65 oldland;
⓱    end;
⓲    return;

⓳    newpage: put / @15
           'Customers Whose Country was Changed to
               UNITED STATES'
           // @17 'Name' @58 'State' @65 'old Country';
⓴    return;

   abendit:
     file log;
     put _all_;
     abort;
run;
filename tranrept clear;
```

❶   The length of SSA1, an SSA variable specified in the INFILE
                statement, is set prior to execution of the DL/I INFILE statement,
                as required.

❷   The INFILE statement specifies ACCTSAM as the PSB, and the DLI
                specification tells SAS that this step will access DL/I resources. The
                SSA= option identifies SSA1 as a variable that contains a Segment
                Search Argument. (The length of SSA1 was established by the
                LENGTH statement.) The CALL= option specifies FUNC as the
                variable containing DL/I call functions, and STATUS is used to
                return the status code. The value of PCBNO is used to select the
                appropriate PCB for this program. This value is carried over in
                successive executions of the DATA step.

These defaults are in effect for other DL/I INFILE options: the input and output buffers are 1000 bytes in length, and segment names and PCB mask data are not returned. Program access is sequential.

❸ The FUNC variable is assigned a value of **GHN**, so the next DL/I INPUT statement issues a get-hold-next call.

❹ The SSA1 variable is assigned a value of **CUSTOMER**. The GHN call is qualified to retrieve a CUSTOMER segment.

❺ The DL/I INPUT statement specifies positions and informats for some of the fields in the CUSTOMER segment. When this statement executes, a qualified GHN call is issued. If the call is successful, a CUSTOMER segment is retrieved and placed in the input buffer. Since variables are named in the INPUT statement, the segment data is moved to SAS variables in the program data vector.

❻ When a call is not successful (that is, when the DL/I status code is something other than blank, **CC**, **GA**, or **GK**), the automatic SAS variable _ERROR_ is set to 1. If the status code is set to **GB** (indicating end of database), and if the DATA step is processing sequentially (as this one is), the DATA step is stopped automatically with an end-of-file return code sent to SAS.

❼ If the call is successful, the values of COUNTRY and STATE are checked. If COUNTRY is not **UNITED STATES**, and the STATE value is alphabetic, a DO group (statements 8 through 17) executes.

❽ The value of COUNTRY is assigned to a new variable called OLDLAND.

❾ COUNTRY's value is changed to **UNITED STATES**.

❿ A DL/I FILE statement indicates that an update call is to be issued. Notice that the FILE statement specifies the same PSB named in the DL/I INFILE statement, as required.

⓫ The value of FUNC is changed from GHN to REPL. If the FUNC value is not changed, an update call cannot be issued.

⓬ The value of SSA1 is changed from CUSTOMER to blanks. Since the REPL call uses the segment retrieved by the GHN call, an SSA is not needed.

⓭ The DL/I PUT statement formats the CUSTOMER segment in the output buffer and issues the REPL call. The entire segment must be formatted, even though the value of only one field, COUNTRY, is changed.

⓮ If the REPL call is not successful (that is, the status code from DL/I was not blank), all values from the program data vector are written to the SAS log and the DATA step aborts.

⓯ If the REPL call is successful, the step goes on to execute another FILE statement. This is not a DL/I FILE statement; instead, it specifies the fileref (TRANREPT) of an output file for a printed report on the replaced segments. The HEADER= option points to the NEWPAGE subroutine. Each time a new page of the update report is started, SAS links to NEWPAGE and executes the statement.

⓰ The PUT statement specifies variables and positions to be written to the TRANREPT output file.

❼   The DO group is terminated by the END statement.

❽   Execution returns to the beginning of the DATA step when this
     RETURN statement executes.

❾   This PUT statement executes when a new page starts in the output
     file TRANREPT. The HEADER= option in the FILE TRANREPT
     statement points to the NEWPAGE label, so when a new page
     begins, SAS links to this labeled statement and prints the specified
     heading.

❿   After printing the heading, SAS returns to the PUT statement
     immediately after the FILE TRANREPT statement (item 16) and
     continues execution of the step.

Example 8: Using the Blank INPUT Statement

     This program calculates customer balances by retrieving a CUSTOMER segment
and then all CHCKACCT and SAVEACCT segments for that customer record. The
CUSTOMER segments are retrieved by qualified get-next calls, and the CHCKACCT
and SAVEACCT segments are retrieved by qualified get-next-within-parent calls. A **GE**
or **GB** status when retrieving the CHCKACCT and SAVEACCT segments indicates that
there are no more of that segment type for the current parent segment (CUSTOMER).
     The numbered comments following this program correspond to the numbered
statements in the program:

```
❶ data balances;
❷    length ssa1 $9;
❸    keep soc_sec_number
            chck_bal
            save_bal;
❹    chck_bal = 0;
     save_bal = 0;

❺    infile acctsam dli pcbno=4 call=func ssa=ssa1
      status=st;
❻    func = 'GN  ';
❼    ssa1 = 'CUSTOMER ';

❽    input @;
❾    if st ¬= '  ' and
        st ¬= 'CC' and
        st ¬= 'GA' and
        st ¬= 'GK' then
       link abendit;

❿    input @1 soc_sec_number $char11.;
⓫    st = '  ';
⓬    func = 'GNP ';
⓭    ssa1 = 'CHCKACCT ';

⓮    do while (st = '  ');
⓯       input @;
⓰       if st = '  ' then
            do;
```

```
❼          input @13 check_amount pd5.2;
❽          chck_bal=chck_bal + check_amount;
❾      end;
❿   end;

㉑   if st ¬= 'GE' then
      link abendit;
㉒   st = '  ';
㉓   _error_ = 0;
㉔   input;
㉕   ssa1 = 'SAVEACCT ';

㉖   do while (st = '  ');
        input @;
        if st = '  ' then
           do;
               input @13 savings_amount pd5.2;
               save_bal = save_bal + savings_amount;
           end;
      end;

      if st = 'GE' then
         _error_ = 0;
      else
         link abendit;
      return;

㉗ abendit:
      file log;
      put _all_;
      abort;
   run;

㉘  proc print data=balances;
      title2 'Customer Balances';
   run;
```

❶   The DATA step creates a new SAS data set called BALANCES.

❷   The length of SSA1, an SSA variable specified in the INFILE statement, is set prior to execution of the DL/I INFILE statement, as required.

❸   The KEEP statement tells SAS that the variables SOC_SEC_NUMBER, CHCK_BAL, and SAVE_BAL are the only variables to be included in the BALANCES data set.

❹   The CHCK_BAL and SAVE_BAL variables are assigned an initial value of 0 and are reset to 0 for each new customer.

❺   The INFILE statement specifies ACCTSAM as the PSB, and the DLI specification tells SAS that this step will access DL/I resources. The SSA= option identifies SSA1 as a variable that contains an SSA. (The length of SSA1 was established by the LENGTH statement.) The CALL= option specifies FUNC as the variable containing DL/I

call functions, and the PCBNO= option specifies which database PCB should be used.

These defaults are in effect for the other DL/I INFILE statement options: the input buffer is 1000 bytes in length, and segment names and PCB mask data are not returned. There are no qualified SSAs in the program, so access is sequential.

❻ The FUNC variable is assigned a value of **GN**, so the next DL/I INPUT statement will issue a get-next call.

❼ The SSA1 variable is assigned a value of CUSTOMER, so the GN call will retrieve the CUSTOMER segment.

❽ The only specification in the DL/I INPUT statement is the trailing @ sign. When the statement executes, the GN call is issued and, if the call is successful, a CUSTOMER segment is retrieved and placed in the input buffer. Since no variables are named in the INPUT statement, the segment data is not moved to SAS variables in the program data vector. Instead, the segment is held in the input buffer for the next DL/I INPUT statement that executes (that is, the next DL/I INPUT statement does not issue a call but uses the data already in the buffer).

❾ When a call is not successful (that is, when the DL/I status code is something other than blank, **CC**, **GA**, or **GK**), the automatic SAS variable _ERROR_ is set to 1. If the status code is set to **GB** (indicating end of database) and if the DATA step is processing sequentially (as this one is), the DATA step is stopped automatically with an end-of-file return code sent to SAS.

❿ If the call is successful, this DL/I INPUT statement executes. It moves the SOC_SEC_NUMBER value from the input buffer (where the segment was placed by the previous DL/I INPUT statement) to a SAS variable in the program data vector.

⓫ The value of the ST variable for status codes is reset to blanks.

⓬ The value of the FUNC variable is reset to **GNP**. The next call issued will be a get-next-within-parent call.

⓭ The SSA1 variable is reset to **CHCKACCT**, so the next call will be for CHCKACCT.

⓮ This DO/WHILE statement initiates a DO-loop (statements 15 through 20) that iterates as long as blank status codes are returned.

⓯ Again, the only specification in this DL/I INPUT statement is the trailing @ sign. When the statement executes, the GNP call is issued for a CHCKACCT segment. If the call is successful, a CHCKACCT segment is retrieved and placed in the input buffer. The segment data is not moved to SAS variables in the program data vector. Instead, the segment is held in the input buffer for the next DL/I INPUT statement that executes.

⓰ If a blank status code is returned, the GNP call was successful, and a DO-group (statements 17 and 18) executes.

⓱ This DL/I INPUT statement moves the CHECK_AMOUNT value (in the PD5.2 format) from the input buffer to a SAS variable in the program data vector.

18   The variable CHCK_BAL is assigned a new value by adding the value of CHECK_AMOUNT just obtained from the CHCKACCT segment.

19   The END statement signals the end of the DO-group.

20   This END statement ends the DO-loop.

21   If the GNP call is not successful and returns a non-blank status code other than **GE**, the DATA step stops and the job abends.

22   If the GNP call is not successful and returns a **GE** status code, the remainder of the step executes. (The **GE** status code indicates that all checking accounts for the customer have been processed.) In this statement, the ST= variable is reset to blanks.

23   _ERROR_ is reset to 0 to prevent SAS from printing the contents of the input buffer and program data vector to the SAS log.

24   The blank INPUT statement releases the hold placed on the input buffer by the last INPUT @ statement. This enables you to issue another call with the next DL/I INPUT statement.

25   The SSA1 variable is reset to **SAVEACCT**, so the next call will be qualified for SAVEACCT.

26   This DO/WHILE statement initiates a DO loop that is identical to the one described in items 14 through 20, except that the GNP calls retrieve SAVEACCT segments rather than CHCKACCT segments. The GNP calls also update SAVE_BAL.

27   The ABENDIT code, if linked to, aborts the DATA step.

28   The PROC PRINT step prints the BALANCES data set created by the IMS DATA step.

The following output shows the results of this example.

Output 8.8   Results of Using the Blank INPUT Statement

```
            Customer Balances


                                  soc_sec_
    OBS      chck_bal     save_bal    number

     1       3005.60       784.29   667-73-8275
     2        826.05      8406.00   434-62-1234
     3        220.11       809.45   436-42-6394
     4       2392.93      9552.43   434-62-1224
     5          0.00         0.00   232-62-2432
     6       1404.90       950.96   178-42-6534
     7          0.00         0.00   131-73-2785
     8        353.65       136.40   156-45-5672
     9       1243.25       845.35   657-34-3245
    10       7462.51       945.25   667-82-8275
    11        608.24       929.24   456-45-3462
    12        672.32         0.00   234-74-4612
```

Example 9: Using the Qualified SSA

In this example, path calls with qualified SSAs are used to produce a report showing which accounts in the ACCTDBD database had checking account debits on March 28,

1995. The numbered comments following this program correspond to the numbered statements in the program:

```
filename tranrept 'your.sas.tranrept' disp=old;
data _null_;
❶    retain ssa1 'CHCKACCT*D '
             ssa2 'CHCKDEBT(DEBTDATE =032895) ';

❷    infile acctsam dli ssa=(ssa1,ssa2) status=st
             pcbno=4;
❸    input @1   check_account_number $char12.
             @13  check_amount          pd5.2
             @18  check_date            mmddyy8.
             @26  check_balance         pd5.2
             @41  check_debit_amount    pd5.2
             @46  check_debit_date      mmddyy8.
             @54  check_debit_time      time8.
             @62  check_debit_desc      $char40.;

❹    if st ¬= '  ' and
         st ¬= 'CC' and
         st ¬= 'GA' and
         st ¬= 'GK' then
❺          if st = 'GB' | st = 'GE' then
             do;
                _error_ = 0;
                stop;
             end;
❻          else
             do;
                 file log;
                 put _all_;
❼               abort;
             end;

❽    file tranrept header=newpage notitles;
❾    put @10 check_account_number
         @30 check_debit_amount dollar13.2
         @45 check_debit_time    time8.
         @55 check_debit_desc;
❿    return;
⓫    newpage: put / @15 'Checking Account Debits
                    Occurring on 03/28/95'
                 // @08 'Account Number' @37 'Amount'
                    @49 'Time' @55 'Description' //;
⓬    return;
run;
filename tranrept clear;
```

❶   The RETAIN statement specifies values for the two SSA variables, SSA1 and SSA2.

SSA1 is an SSA for the CHCKACCT segment with the command code for a path call, *D. This command code means that the CHCKACCT segment is returned as well as the target segment,

CHCKDEBT. SSA2 is a qualified SSA specifying that CHCKDEBT segments for which DEBTDATE=032895 be retrieved.

These values are retained for each iteration of the DATA step. The RETAIN statement satisfies the requirement that the length of an SSA variable be specified before the DL/I INFILE statement.

❷ The INFILE statement specifies ACCTSAM as the PSB. The DLI specification tells SAS that the step will access DL/I resources. Two variables containing SSAs are identified by the SSA= option, SSA1 and SSA2. (Their values were set by the earlier RETAIN statement.) The STATUS= option specifies the ST variable for status codes returned by DL/I, and the PCBNO= option specifies the PCB selection.

These defaults are in effect for the other DL/I INFILE options: all calls are get-next calls, the input buffer length is 1000, and the segment names and PCB mask data are not returned.

❸ When the DL/I INPUT statement executes, the GN call is issued. If successful, CHCKACCT and CHCKDEBT segments are placed in the input buffer, and the values are then moved to SAS variables in the program data vector. The DL/I INPUT statement specifies positions and informats for the variables in both the CHCKACCT and CHCKDEBT segments because the path call returns both segments.

❹ If the qualified GN call issued by the DL/I INPUT statement is not successful (that is, it obtains any return code other than blank, **CC**, **GA**, or **GK**), _ERROR_ is set to 1 and the program does further checking.

❺ If the ST variable value is **GB** (a status code meaning that the end-of-file has been reached) or **GE** (segment not found), _ERROR_ is reset to 0 so that the contents of the input buffer and program data vector are not printed to the SAS log, and SAS stops processing the DATA step. In a program issuing path calls with qualified SSAs, DL/I might first return a GE status code when it reaches end-of-file. Then, if another get call is issued, DL/I returns the GB status code. Therefore, in this program, we treat a GE code as a GB code.

In a sequential-access program with unqualified SSAs, this statement is not necessary because the end-of-file condition stops processing automatically. However, when a program uses qualified SSAs, the end-of-file condition is not set on because DL/I might not be at the end of the database. Therefore, you need to check status codes and explicitly stop the step.

❻ For any other non-blank return code, all values from the program data vector are written to the SAS log.

❼ The DATA step execution terminates, and the job abends.

❽ If the GN call is successful, the step goes on to execute another FILE statement. This is not a DL/I FILE statement. Instead, it specifies the fileref (TRANREPT) of an output file for a printed report on the retrieved segments.

The HEADER= option points to the NEWPAGE statement label (statement 11). When a new page begins, SAS links to the labeled statement and prints the specified heading.

❾   The PUT statement specifies variables and positions to be written to the output file.

❿   Execution returns to the beginning of the DATA step when this RETURN statement executes.

⓫   The PUT statement labeled NEWPAGE executes when a new page is started in the output file TRANREPT. This PUT statement writes the title for the report at the top of the new page.

⓬   After printing the heading, SAS returns to the PUT statement immediately after the FILE TRANREPT statement (statement 8) and continues execution of the step.

9
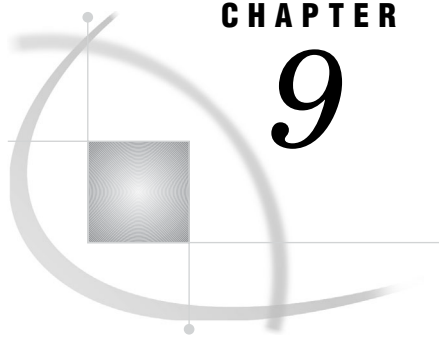
How to Use the IMS DATA Step Interface

Introduction to Using the IMS DATA Step Interface

The SAS/ACCESS interface to IMS can access databases through a DLI or DBB batch region, and an IMS/ESA DB/DC BMP region.* Chapter 8, "Overview of the IMS DATA Step Interface," on page 153 describes DATA step programming statements and DL/I statements that are available with the IMS DATA step interface. This section describes Fast Path DL/I database access and non-database access calls.

---

*   Beginning with Version 6, the SLI region type is not supported; SLI functionality is supported through BMP regions. Databases that are allocated to CICS control regions can be accessed by SAS applications through a BMP region by using the DBCTL facility of IMS/ESA and CICS/ESA.

OS/390 DL/I System Calls

The following table summarizes the functions and region types for non-database access calls that are supported by the IMS DATA step interface.

Table 9.1 Summary of Fast Path and Non-Database Access Calls

Function	Purpose	Valid Region Types	Notes
CHKP	create the synchronization point, recovery	all IMS DATA step interface region types	OS/VS option not supported. In transaction-processing BMPs, next call must be GU using I/O PCB.
CHNG	change destination for messages	IMS/ESA BMP regions	sets the destination for a modifiable TP PCB
CMD	issue IMS/ESA commands from a program	IMS/ESA BMP regions	when CC status returned, must next issue GU to retrieve response
DEQ	release a class of segments enqueued with the Q command code	IMS/ESA BMP regions	specify class (A-J) of segments to dequeue
FLD	access fields in MSDBs	IMS/ESA BMP regions	Fast Path Facility only
GCMD	retrieve additional response segments to a command if more than one	IMS/ESA BMP regions	functions as a GN to the queue after first response segment retrieved with GU
GN	retrieve additional segments of a message with more than one segment	IMS/ESA BMP regions	uses I/O PCB
GU	retrieve the first segment of a message	IMS/ESA BMP regions	uses I/O PCB
ISRT	format and send message segment to the queue	IMS/ESA BMP regions	uses I/O or TP PCB
LOG	insert a record to the DL/I system log	MVS DL/I regions	uses I/O PCB
POS	return position information from DEDBs	IMS/ESA BMP regions	Fast Path Facility only
PURG	terminate the current message being inserted; optionally, insert the first segment of the next message	IMS/ESA BMP regions	uses TP PCB

Function	Purpose	Valid Region Types	Notes
ROLB	back out database updates since last sync point	IMS/ESA BMP regions and batch DL/I regions in IMS/ESA Release 3	in BMP regions, also backs out messages inserted to the queue since the last synchronization point. Next call must be GU using I/O PCB if ROLB requested return of previous message.
ROLL	back out database updates since last sync point, and abend	IMS/ESA BMP regions, batch DL/I regions in IMS/ESA Release 3, and CICS/VS shared DL/I regions	in BMP regions also backs out messages inserted to the queue since the last synchronization point

Fast Path DL/I Database Access

The following two Fast Path database types are supported by the IMS DATA step interface by using a BMP region:

☐ Main storage databases (MSDBs) store and provide access to an installation's most frequently used data, which resides in virtual storage during execution. The data is stored in segments, and each segment can be available to all terminals or to specific terminals.

☐ Data entry databases (DEDBs) provide a high level of availability for, and efficient access to, large volumes of detailed data. They are hierarchic structures that contain a special type of segment that is used for the fast collection of detailed information. The segments are called *sequential dependent segments* because they are stored in time sequence as they are committed to the database.

Standard DL/I database calls can be used with a PCB that references an MSDB or DEDB to access database segments. Two additional calls are available:

☐ The FLD call allows read and update access to a field in an MSDB.

☐ The POS call returns information about the position of the current sequential dependent segment in a DEDB and free space in the DEDB area.

The IMS DATA step interface supports the FLD and POS calls from a BMP region.

FLD Call

The FLD call is used to verify and, optionally, to update the contents of one or more fields in an MSDB segment. Individual field verification or change specifications are specified in *field search arguments* (FSAs). (The format and use of FSAs are described in the IBM publication *IMS/ESA: Application Programming: EXEC DLI Commands for CICS and IMS*.) FSAs are passed to DL/I in the I/O area. Therefore, in the IMS DATA step interface, the PUT statement is used to format the FSAs in the output buffer and to execute the FLD call.

Like any DL/I call, the FLD call returns a status code. In addition, DL/I returns abnormal status information for each FSA in the call. If a non-blank status code is returned from a FLD call, it might be necessary to examine the contents of the FSA return codes. The DL/I INFILE statement option FSARC= specifies a 200-byte character variable to which the first 200 FSA status code bytes can be returned.

The following example issues a FLD call against an MSDB called INVNTORY:

```
ssa1='PRODUCT (PRODUCT = LOCKS    )';
  infile msdbpsb dli call=cfunc dbname=database
```

```
      ssa=ssa1 fsarc=fsa_rc;
   file msdbpsb dli;
   cfunc = 'FLD ';
   database = 'INVNTORY';
   put @1 'QUANTITY  H100*QUANTITY -100*ORDERS +1 ';
```

The call accesses a segment called PRODUCT containing data on locks. The FLD call performs these functions:

□ verifies that the QUANTITY field is greater than 100

□ updates the QUANTITY field by subtracting 100 from its current value

□ updates the ORDERS field by adding 1 to its value.

If the QUANTITY field value is not greater than 100 when the FLD call is executed, the return code for the first FSA contains a D. The following statements check for errors in the call and print an appropriate message on the SAS log for this error:

```
if _error_ then do;
   file log;
   if substr(fsa_rc,1,1) = 'D'
   then put / '*** Quantity of Product Locks Less
      Than 100 ***';
   put _all_;
   _error_=0;
   end;
```

POS Call

The POS call is used with a DEDB to perform one of the following:

□ retrieve the position of a specific sequential dependent segment

□ retrieve the position of the last inserted sequential dependent segment

□ find out how much free space is available within a DEDB area.

In an IMS DATA step program, the POS call is issued with a DL/I INPUT statement and a DB PCB. After a POS call is issued, the input buffer is formatted with the requested data as explained in the IBM publication *IMS/ESA: Application Programming: EXEC DLI Commands for CICS and IMS*.

The SAS statements below execute a POS call for a DEDB called ORDERS:

```
retain ssa1 'PRODUCT (PRODUCT = LOCKS     )';
infile dedbpsb dli call=cfunc dbname=database
   ssa=ssa1;
cfunc = 'POS ';
database = 'ORDERS  ';
input @3 areaname $char8.
      @11 cycl_cnt $pib4.
      @15 vsam_rba $pib4.;
```

The call obtains the position of the last inserted ORDRITEM sequential dependent segment for the locks PRODUCT segment.

Non-Database Access Calls

Some DL/I calls communicate with DL/I for reasons other than database access. This section describes how to use the non-database calls in IMS DATA step programs.

Most non-database calls require either an I/O PCB or a TP PCB. The basic CHKP call, the LOG call, the ROLL call, and the ROLB call, however, are supported in all DL/I region types that can be accessed through OS/390.

Also, some calls can be executed only from an IMS/ESA BMP region. All of these calls are described in the following sections

I/O PCBs

An I/O PCB is a program communication block that is used only in OS/390 DL/I environments. An I/O PCB is similar to a DB PCB, but an I/O PCB communicates non-database access requests to DL/I instead of database requests. The type of DL/I region executed and an option specified when PSBs are generated determine whether an I/O PCB is included in a PSB. The IMS/ESA control region automatically provides an I/O PCB for BMP regions. The I/O PCB is generated in batch DL/I regions if the CMPAT=YES option is specified in the PSBGEN statement when the PSB is generated.

If an I/O PCB is present, it is always the first PCB in the PSB. Therefore, be careful in how you specify the DL/I INFILE statement options PCBNO=, PCB=, and DBNAME= when you need the I/O PCB. The value of PCBNO= must be 1. If the DBNAME= option is specified, that variable's value must be set to blanks. Finally, if a PCB= variable is specified, it must have a value of 1.

In all OS/390 DL/I regions, the I/O PCB is used to issue the CHKP and LOG calls. In an IMS/ESA BMP region, the I/O PCB is also used to read transaction messages from the IMS/ESA message queues, to insert response messages to the terminal that originated the transaction, and to communicate certain system calls that are unique to the IMS/ESA DB/DC system.

TP PCBs

A TP PCB is a program communication block that is used with the IMS DATA step interface only in IMS/ESA BMP regions. It is similar to the I/O PCB, but there are two important differences:

- A TP PCB is used to insert messages *only* to terminal or transaction message queues. A TP PCB cannot be used for a Get call to a message queue.

- Unlike an I/O PCB, a TP PCB can direct a message to a destination (transaction or terminal message queue) other than the terminal that originated the message.

There are two kinds of TP PCBs: *nonmodifiable* and *modifiable*. A *nonmodifiable* TP PCB has a fixed destination that is specified when the PSB is generated. The destination can be either a terminal or transaction message queue. A *modifiable* TP PCB does not have a destination associated with it when the PSB is generated. Instead, the program must set the destination before using the PCB to insert a message to the message queue. The destination can be changed between messages so that more than one destination can be accessed by one TP PCB.

When TP PCBs are present, they follow the I/O PCB (if any) and precede the DB PCBs. Unless the TP PCB is the first PCB in the PSB, you must use the PCB= option in the DL/I INFILE statement to select the appropriate TP PCB. You cannot use the DBNAME= option because no DBD name is associated with a TP PCB.

Feedback Data

Just as information from DB PCBs is available to the SAS program through the STATUS= and PCBF= variables after a DL/I call, so is information from the I/O and TP PCBs.* The format of the data in the PCBF= variable differs, however, according to the PCB type.

If a DL/I call uses the I/O PCB, the PCBF= variable data is formatted as shown in the following table.

Table 9.2 Format of I/O PCB Feedback Data

Bytes	Description
1-8	These bytes of the PCBF= variable contain the name of the logical terminal (LTERM) that issued the message.
9-10	These bytes are reserved for IMS/ESA usage.
11-12	These bytes contain the DL/I status code. The status code can also be obtained by specifying the STATUS= option in the DL/I INFILE statement.
13-16	These bytes contain the date that the message was queued. The date is in packed decimal, right aligned, Julian date format (YYDDD).
17-20	The time that the message was queued is contained in these bytes in packed decimal format (HHMMSS.S).
21-24	The input message number assigned by IMS/ESA is contained in these bytes in IB4. (full-word binary) format.
25-32	These bytes contain the Message Output Descriptor (MOD) name. An MOD name is connected to this PCB if Message Format Services (MFS) is used. If MFS is not used, there is no MOD, and this field is blank.
33-40	These bytes contain the user identification data. The contents vary according to the source of the message

If a DL/I call uses a TP PCB, the data in the PCBF= variable is formatted as shown in the following table.

---

* *IMS/ESA: Application Programming: EXEC DLI Commands for CICS and IMS*, an IBM publication, describes the PCB mask data.

Table 9.3 Format of TP PCB Feedback Data

Bytes	Description
1-8	These bytes of the PCBF= variable contain the name of the destination associated with the PCB.
9-10	These bytes are reserved for IMS/ESA usage.
11-12	These bytes contain the DL/I status code. The status code can also be obtained by specifying the STATUS= option in the DL/I INFILE statement.

Basic CHKP Call

The basic CHKP call can be issued in batch DL/I regions as well as in online DL/I regions. This call establishes a program synchronization point.* (Synchronization points are described in "General Considerations for Sharing Resources" on page 35.)

The following example shows SAS programming statements that issue a CHKP call. The example is run using the SAS system option IMSREGTP=DLI:

```
data _null_;
  retain chkpnum 0;
  infile acctsam dli call=func pcb=pcbindex
    status=st;
  file acctsam dli;
  func = 'CHKP';
  pcbindex = 1;
  chkpnum = chkpnum +1;
  put @1 'SAS'
      @4 chkpnum z5.;
  if st = '  ' then
    return;
  file log;
  put _all_;
  abort;
run;
```

The CHKPNUM variable, first referenced in the RETAIN statement, is used to build a *checkpoint ID*. A checkpoint ID is an 8-byte value that is written to the DL/I log record to identify the program checkpoint. A checkpoint ID is not required but is very useful and should be included routinely in programs that issue CHKP calls. In this example, the checkpoint ID is built in the output buffer. If the same sequence of statements is used for each CHKP call, the checkpoint ID is incremented by 1 for each call.

The PCB= variable, PCBINDEX, has a value of 1. This indicates that the first eligible PCB is used for the CHKP call. A CHKP call requires the I/O PCB that is the first PCB in the PSB (see "I/O PCBs" on page 201).

Note: An I/O PCB is always generated for PSBs in a BMP region. If you are going to issue a CHKP call under DL/I, you must use the CMPAT=YES option in the PSBGEN statement for batch regions DLI and DBB. If an I/O PCB is not present, you get the message that the call is invalid for a DB PCB. △

The CHKP call is successful if _ERROR_=0 and the STATUS= variable (ST) is blank. Otherwise, the STATUS= variable contains a status code that indicates the cause of the

---

* The OS/VS checkpoint option of the CHKP call in an IMS/ESA DL/I region is not supported in the IMS DATA step interface.

failure. In particular, an **XD** status code in an IMS/ESA BMP region indicates that the IMS/ESA control region is being shut down.

CHKP Calls in IMS/ESA BMP Regions

A CHKP call performs an additional function when it is issued in an IMS/ESA BMP transaction-processing program (that is, when the SAS system option IMSBPIN= specifies a valid transaction code and the PCB used is type TP). In addition to establishing a synchronization point, the call returns the first segment of the next message to the call's I/O area. Since a CHKP call is issued by a DL/I PUT statement, the I/O area is the SAS output buffer.

You cannot read from the output buffer in a DATA step, but you can access the message segments that are placed in the output buffer. You format a GU call that uses the I/O PCB. When the DL/I INPUT statement executes, the SAS/ACCESS interface remembers that the output buffer contains data from a previous CHKP call. Instead of issuing the GU call, the SAS/ACCESS interface moves the segment from the output buffer to the input buffer, where it can be read. Therefore, in a BMP transaction-processing program, the first call issued after a CHKP call *must be* a GU that references the I/O PCB.*

Consider the previous example in "Basic CHKP Call" on page 203, which shows SAS statements that issue a CHKP call. If you issue the CHKP call in a BMP transaction-processing program, additional statements are needed. This example issues one CHKP call and optionally moves a message segment to the input buffer.

In this example, change *trancode* in the OPTIONS statement to a valid transaction code at your site. This example is run using the SAS system options IMSREGTP=BMP and IMSBPIN=*trancode*:

```
options imsbpin=trancode;
data _null_;
  retain chkpnum 0;
  infile acctsam dli call=func pcb=pcbindex
     status=st;
  file acctsam dli;
  func = 'CHKP';
  pcbindex = 1;
  chkpnum = chkpnum +1;
  put @1 'SAS'
     @4 chkpnum z5.;
  if st = '  ' then
    do;
      func = 'GU  ';
      input @;
      if st ¬= '  ' then
        if st= 'QC' then
          do;
            _error_ = 0;
            stop;
          end;
        else
          link abendit;
    end;
  else
```

---

* This is not the call sequence that would be used if programming in PL/I, COBOL, or Assembler, but it is consistent with the actions taken by DL/I after a CHKP call.

```
      if st = 'QC' then
        do;
          _error_ = 0;
          stop;
        end;
    else
      link abendit;
    stop;

    abendit:
      file log;
      put _all_;
      abort;
  run;
  options imsbpin=*;
```

If DL/I did not return the first segment of the next message automatically after a CHKP call, the GU call would be necessary to retrieve the next message.

LOG Call

A LOG call inserts user log records in the DL/I log with the I/O PCB (see "I/O PCBs" on page 201). To insert a log record, you must specify the following:

☐ the text of the log record

☐ a valid log code

☐ a value for the ZZ field

☐ the value of the LL field, which is the sum of the lengths of the log record, log code, ZZ field, and LL field.

In an IMS DATA step program, the LOG call is issued with the DL/I PUT statement. The PUT statement must format the log record being inserted. The following statements from a sample program insert a log record with a code of **'A0'x** in the IMS log. The example can be run using the SAS system options IMSREGTP=DLI or IMSREGTP=BMP:

```
data _null_;
  infile acctsam dli call=func pcb=pcbindex
    status=st;
  file acctsam dli;
  func = 'LOG ';
  pcbindex = 1;
  ll = 23;
  zz = '0000'x;
  logcode = 'A0'x;
  logsegm = 'Text of Log Record';
  put @1 ll pib2.
      @3 zz
      @5 logcode
      @6 logsegm;
  if st ⌐= '  ' then
    do;
      file log;
      put _all_;
      abort;
    end;
```

```
        stop;
   run;
```

After the LOG call, you can check the values of the STATUS= variable and _ERROR_
to see whether the call was successful. If _ERROR_=0, the log record was inserted
properly. Otherwise, the STATUS= variable contains an error code that indicates why
the call was not successful.

If the PSB is generated with LANG=PLI, then the PUT statement must be modified
because the LL field has a 4-byte length:

```
put @1 ll pib4.
       @5 zz
       @7 logcode
       @8 logsegm;
```

The value of the LL variable does not change.

ROLL Call

In an online access region, the ROLL call has two purposes:

☐ to back out any DL/I updates to database segments or message queues that have
   been made since the last program synchronization point

☐ to abend the program with a user 0778 completion code.

The ROLL call performs the same functions in a batch DL/I region if the following
conditions are present:

☐ a DASD log data set is used

☐ the IMS DATA step interface option IMSDLBKO= specifies a value of Y.

Otherwise, the ROLL call in a batch DL/I region only causes the program to abend
with a user 0778 completion code. In this latter case, the database back-out utility
must be run with the log data set in order to back out any database updates made since
the last program synchronization point.

The following example shows statements that issue a ROLL call. This example is run
using the SAS system option IMSREGTP=DLI:

```
data _null_;
   infile acctsam dli call=func pcb=pcbindex
     status=st;
   file acctsam dli;
   func = 'ROLL';
   pcbindex = 1;
   put;
   if st ¬= ' ' then
     do;
        file log;
        put _all_;
        abort;
     end;
   stop;
run;
```

ROLB Call

A ROLB call is used in a batch DL/I region to back out any DL/I database updates
that have been made since the last program synchronization point. ROLB differs from

the ROLL call because it does not cause an 0778 abend. The ROLB call requires use of the I/O PCB (see "I/O PCBs" on page 201).

The ROLB call can be issued in batch DL/I regions if the following is true:

□ a DASD log data set is used

□ the IMS DATA step interface option IMSDLBKO= specifies a value of Y.

Otherwise, the ROLB call can be issued only from an IMS/ESA BMP region, as described in "IMS/ESA Message Queue Access" on page 212.

The following sequence of SAS statements issues a ROLB call. This example is run using the SAS system options IMSREGTP=DLI and IMSDLBKO=Y:

```
options imsdlbko=y;
data _null_;
  infile acctsam dli call=func pcb=pcbindex
    status=st;
  file acctsam dli;
  func = 'ROLB';
  pcbindex = 1;
  put;
  if st ¬= '  ' then
    do;
      file log;
      put _all_;
      abort;
    end;
  stop;
run;
```

The ROLB call has been successfully executed if _ERROR_=0 after the call; otherwise, you can check the value of the STATUS= variable to see why the call did not complete successfully.

IMS/ESA BMP System Calls

DEQ Call

The DEQ call is used in a BMP region to dequeue a class of database segments that have been enqueued with the Q command code of a Get call. The DEQ call is issued with the PUT statement and requires the use of the I/O PCB. The PUT statement specifies the class of segments to be dequeued. The following sequence of SAS statements dequeue the segments that have been enqueued to Class A with a QA command code in a Get call. This example is run using the SAS system option IMSREGTP=BMP:

```
data _null_;
  infile tranpsb dli call=func pcb=pcbindex
    status=st;
  file tranpsb dli;
  func = 'DEQ ';
  pcbindex = 1;
  put @1 'A';
  if st ¬= '  ' then
    do;
      file log;
      put _all_;
```

```
      abort;
    end;
  stop;
run;
```

The call has been successfully executed if _ERROR_=0 after the call. Otherwise, the STATUS= variable contains a status code that indicates the reason for the failure.

ROLB Call

The ROLB call is used in a BMP region to back out any DL/I updates to database segments or message queues that have been made since the last program synchronization point. The ROLB call is issued with a PUT statement and requires the use of the I/O PCB.

Examples 1 to 3 are run using the SAS system options IMSREGTP=BMP and IMSBPIN=*trancode*. Example 1 shows a sequence of SAS statements that issue a ROLB call.

```
options imsbpin=trancode;
data _null_;
  infile acctsam dli call=func pcb=pcbindex
     status=st;
  file acctsam dli;
  func = 'ROLB';
  pcbindex = 1;
  put;
  if st ¬= '  ' then
    do;
      file log;
      put _all_;
      abort;
    end;
  stop;
run;
```

The call has been successfully executed if _ERROR_=0 after the call. Otherwise, the ST variable contains a status code that indicates the reason for the failure.

If the ROLB call is issued in a BMP transaction processing program *and* the DL/I PUT statement issuing the call formats non-blank data in columns 1 through 6, the call also returns the first segment of the previous message. Any non-blank data can be written in columns 1 through 6 of the output buffer.

When these conditions are fulfilled, the IMS DATA step interface saves the returned message segment. The next call *must be* a GU that uses the I/O PCB. The DATA step interface intercepts the GU call when the INPUT statement executes, so the call is not actually issued. Instead, the returned segment is moved to the input buffer where it can be read.

Example 2 shows a sequence of SAS statements that issue a ROLB call and then a GU call with the I/O PCB:

```
/* put a message in the queue      */
data _null_;
  infile tranpsb dli call=func pcb=pcbindex
     status=st;
  file tranpsb dli;
  func = 'ISRT';
  pcbindex = 2;
  ll = 33;
```

```
      zz = '0000'x;
      msgsegm = 'trancode Message for Example # 2.';
      put @1 ll pib2.
           @3 zz
           @5 msgsegm;
      if st ¬= '  ' then
        do;
          file log;
          put _all_;
          abort;
        end;
      stop;
   run;
   data _null_;
     infile acctsam dli call=func pcb=pcbindex
          status=st;
     pcbindex = 1;
     file acctsam dli;
     func = 'ROLB';
     put @1 'SAVEIO';
     if st ¬= '  ' then
       if st = 'QC' then
         _error_ = 0;
       else
          link abendit;
       func = 'GU  ';
       input @;
       if st = '  ' then
          _error_ = 0;
       else
          link abendit;
       stop;

       abendit:
         file log;
         put _all_;
         abort;
   run;
```

Example 3 shows a sequence of SAS statements that issue a ROLB call and with no GU call to the message queue:

```
   data _null_;
     infile acctsam dli call=func pcb=pcbindex
          status=st;
     file acctsam dli;
     func = 'ROLB';
     pcbindex = 1;
     put @1 'SAVEIO';
     if st ¬= '  ' and
        st ¬= 'QC' then
       link abendit;
     return;

     abendit:
```

```
      file log;
      put _all_;
      abort;
  run;
  options imsbpin=*;
```

The message segment has been successfully moved if _ERROR_=0 after the INPUT statement executes.

If the PUT statement above is changed to **PUT;**, the message segment would *not* be returned by the ROLB call.

CMD Call

A SAS program that executes in a BMP region can insert commands to IMS/ESA with the CMD call if the following conditions are met:

□ the IMS/ESA security allows the PSB and transaction to do so

□ BMPREAD= does not specify Y.

The CMD call is issued by a PUT statement and uses the I/O PCB.

For example, the following sequence of SAS statements issues the **'/START   DB ACCTDBD. '** command. This example is run using the SAS system options IMSREGTP=BMP and IMSBPIN=*trancode* :

```
  options imsbpin=trancode;
  data _null_;
    infile tranpsb dli call=func pcb=pcbindex
        status=st;
    file tranpsb dli;
    func = 'CMD ';
    pcbindex = 1;
    ll = 23;
    zz = '0000'x;
    put @1 ll pib2.
        @3 zz
        @5 '/START DB ACCTDBD. ';
    if st ¬= '  ' then
      do;
        file log;
        put _all_;
        abort;
      end;
  run;
  options imsbpin=*;
```

If _ERROR_=0 after the call, the command was issued properly. If a blank STATUS= code is returned, the command might have completed or it might be in progress, depending on the IMS/ESA command issued.

If a CC status code is returned, the command returned a response message to the output buffer and the IMS DATA step interface saved the response. To retrieve the response, the next call must be a GU that uses the I/O PCB, as is done after CHKP and ROLB calls in the IMS DATA step interface. If subsequent response segments are queued, a CC status code is returned as a result of the GU call. The program can issue GCMD calls (see "GCMD Call" on page 211) to retrieve the subsequent response segments.

See the IBM publication *IMS/ESA: Application Programming: EXEC DLI Commands for CICS and IMS* for more information about the CMD call.

If the PSB is generated with LANG=PLI, the format specified for the LL field must be changed to PIB4.:

```
put @1 ll pib4.
      @5 zz
      @7 '/START DB D1MK0001.';
```

However, the value of the LL variable does not change.

GCMD Call

A SAS program that issues CMD calls can retrieve additional response segments with the GCMD call. The GCMD call acts like a GN to the queue and is issued with a DL/I INPUT statement. The first segment must have been retrieved with a GU call by using the I/O PCB.

The following sequence of statements issues a GCMD call. This example is run using the SAS system options IMSREGTP=BMP and IMSBPIN=*trancode* :

```
data _null_;
  infile tranpsb dli call=func pcb=pcbindex
     status=st;
  func = 'GU  ';
  pcbindex = 1;
  input @;
  if st = 'CC' then
    do;
      func = 'GCMD';
      input @;
      if st = '  ' or
         st = 'QD' then
        do;
          _error_ = 0;
           stop;
        end;
      else
        link abendit;
    end;
  else
    if st = 'QC' then
      do;
        _error_ = 0;
         stop;
      end;
    else
      link abendit;
  return;

  abendit:
    file log;
    put _all_;
    abort;
run;
options imsbpin=*;
```

If _ERROR_=0 after the call, the next response segment is in the input buffer. If a QD status code is returned, there are no more response segments for this response.

IMS/ESA Message Queue Access

If you use the IMS DATA step interface to access IMS data and use that data in programs with a BMP region, you can access the IMS/ESA control region message queues as well as DL/I databases. A BMP program accesses message queues in two ways:

□ A program that is *transaction driven* reads a transaction message from the message queues using the I/O PCB.

□ A program can insert messages to terminal message queues or transaction message queues. When responding to the terminal that originated a transaction, the I/O PCB is used. When inserting a message to a terminal queue that did not originate the message or to a transaction queue, a TP PCB is used.

See the IBM publication *IMS/ESA: Application Programming: EXEC DLI Commands for CICS and IMS* for more information about IMS/ESA data communications programming. This section describes the use of the IMS DATA Step interface to issue DL/I message queue access calls.

Get Calls That Use the I/O PCB

To retrieve message segments for transaction processing, an IMS DATA step interface program

□ must have the IMS DATA step interface option IMSBPIN= set to a valid transaction code

□ issues Get calls with the I/O PCB using DL/I INPUT statements.

To retrieve the first segment of any message, use a GU call. To retrieve subsequent segments of the same transaction message, issue a GN call. You can use the same sequence of SAS statements that issued a GU call for the first segment of a message, but the value of FUNC must be changed to GN. (For more information about GU and GN calls, see "OS/390 DL/I System Calls" on page 198.)

In this example, change *trancode* in the OPTIONS statement to a valid transaction code at your site. This example is run using the SAS system options IMSREGTP=BMP and IMSBPIN=*trancode* :

```
options imsbpin=trancode;
data _null_;
  infile acctsam dli call=func pcb=pcbindex
    status=st;
  func = 'GU  ';
  pcbindex = 1;
  input @;
  if st = '  ' then
    do;
      func = 'GN  ';
      do while (st = '  ');
        input @;
        if st ¬= '  ' then
          if st = 'QD' then
            do;
              _error_ = 0;
              stop;
            end;
          else
            link abendit;
```

```
      end;
    end;
  else
    if st = 'QC' then
      do;
        _error_ = 0;
        stop;
      end;
    else
      link abendit;
  stop;

  abendit:
    file log;
    put _all_;
    abort;
run;
options imsbpin=*;
```

A transaction message segment has been successfully retrieved if _ERROR_=0 or if the STATUS= variable is blank after the call. If _ERROR_ does not equal 0, check the value of the STATUS= variable. When _ERROR_=1 and ST='QC' or ST='QD', there are no more messages in the queue. To find out if there are more messages in the queue, issue another GU call.

The format of a retrieved message segment in the SAS input buffer differs depending on the language that generated the PSB. If an Assembler PSB is used, the message segment is formatted as shown in the following table.

Table 9.4   Assembler PSB Input Buffer Message Segment Format

Bytes	Description
1-2	These bytes of the SAS buffer contain a value that is the length of the segment data plus 4 (2 for the LL field and 2 for the ZZ field) in the PIB2. format.
3-4	These bytes contain the ZZ fields and are reserved for IMS usage.
5-*n*	The segment data begin at byte 5. If this is the first segment of the message, the transaction code (up to 8 bytes in length) is in the first bytes of the message data.

If a PL/I PSB is used, the message segment is formatted as shown in the following table.

Table 9.5 PL/I PSB Input Buffer Message Segment Format

Bytes	Description
1-4	These bytes of the SAS buffer contain a value that is the length of the segment data plus 4 (2 for the LL field and 2 for the ZZ field) in the PIB4. format. (The length here will be 2 bytes less than the total message segment.)
5-6	These bytes contain the ZZ fields and are reserved for IMS usage.
7-*n*	The segment data begins at byte 7. If this is the first segment of the message, the transaction code (up to 8 bytes in length) is in the first bytes of the message data.

ISRT Calls to Message Queues

A SAS program executing in a BMP region can insert messages to the IMS/ESA control region message queues with an ISRT call and the I/O or TP PCBs. For message segments to be inserted the following must be true:

☐ either IMSBPIN= or IMSBPOUT= must specify a valid IMS/ESA destination

☐ BMPREAD= must not equal Y

☐ the message segment text must be specified

☐ a value must be assigned to the ZZ field

☐ the value of the LL field must be specified. The LL field contains the length of the message segment, which is the sum of the lengths of the text, the ZZ field, and the LL field.

The following SAS statements insert a message segment. This example uses the second PCB in the PSB, which is assumed to be a TP PCB. In this example, change *trancode* in the OPTIONS statement to a valid transaction code at your site. This example is run using the SAS system options IMSREGTP=BMP and IMSBPIN=*trancode* :

```
options imsbpin=trancode;
data _null_;
  infile tranpsb dli call=func pcb=pcbindex
     status=st;
  file tranpsb dli;
  func = 'ISRT';
  pcbindex = 2;
  ll = 35;
  zz = '0000'x;
  msgsegm = 'trancode Text of Message Segment';
  put @1 ll pib2.
      @3 zz
      @5 msgsegm;
  if st ⌐= '  ' then
    do;
      file log;
      put _all_;
      abort;
    end;
```

```
      stop;
   run;

   data _null_;
      infile acctsam dli call=func pcb=pcbindex
         status=st;
      func='GU  ';
      pcbindex= 1;
      input @;
      if st ¬= '  ' then
         if st = 'QC' then
            do;
               _error_ = 0;
                stop;
            end;
         else
            do;
               file log;
               put _all_;
               abort;
            end;
      stop;
   run;
   options imsbpin=*;
```

If _ERROR_=0 after the ISRT call, the segment was inserted properly. Otherwise, the STATUS= variable contains a status code that indicates why the call was not successful.

If the PSB is generated with LANG=PLI, the PUT statement must be modified because the length of the LL field is 4 bytes. For example:

```
put @1 ll pib4.
       @5 zz
       @7 msgsegm;
```

The value of the LL variable does not change.

Notes on Inserting Message Segments

□ If the destination of the message is a transaction queue, the text of the first segment of the message must contain the transaction code. This code must match the destination in the TP PCB.

□ If Message Format Services (MFS) is used, a Message Output Descriptor (MOD) is associated with the PCB used for the call. If you want to change the MOD that is associated with the PCB, specify an SSA value of "#MODNAME=*modname*" when the first message segment is inserted.* In the previous example, you could add this statement before the first DL/I PUT statement for the message:

```
SSA1='#MODNAME=DFSMO4';
```

This causes the message to be formatted with the MOD DFSMO4. The **SSA1='** **';** statement should follow the first DL/I PUT so that the MOD is not re-specified on ISRT calls for subsequent message segments.

---

* Although a message queue call does not use an SSA, it is provided as a way to specify the MOD.

PURG Calls for Message Segments

You might want your SAS DATA step program to insert multiple messages with one TP PCB. The requirements for this might vary depending on whether the messages go to the same destination or to different destinations.

When you insert more than one message to the same destination, you can use a PURG call to terminate the current message *and* to insert the first segment of the next message. You issue the PURG call with a PUT statement that formats the first segment of the message to be inserted.

For example, consider the following SAS statements:

```
data _null_;
  infile tranpsb dli call=func pcb=pcbindex
     status=st;
  file tranpsb dli;
  func = 'PURG';
  pcbindex = 2;
  ll = 27;
  zz = '0000'x;
  msgsegm = 'Text of Message';
  put @1 ll pib2.
     @3 zz
     @5 msgsegm;
  if st ¬= '  ' then
    do;
      file log;
      put _all_;
      abort;
    end;
  stop;
run;
```

The PCBINDEX variable is set to 2, so that a TP PCB is used. The values of the LL and ZZ fields are set by assignment statements, and then the message segment text is specified. Notice that the PUT statement, which issues the PURG call, formats the output buffer just as if this were an ISRT call. This example is run using the SAS system option IMSREGTP=BMP.

If you want to change the MOD, use an SSA variable, as described in "ISRT Calls to Message Queues" on page 214.

When you insert messages to different destinations with one TP PCB, you cannot use the PURG call to insert the first segment of the next message. Instead, you should do one of the following:

□ issue a PURG call with the TP PCB to end the current message. The PUT statement that issues the PURG call *must not* format a message segment. The PUT statement should simply be **PUT;**

□ issue a CHNG call to change the TP PCB destination.

□ issue an ISRT call to insert the message segment.

"CHNG Call to TP PCBs" on page 216 shows an example of this sequence of calls. Remember that you must use a modifiable TP PCB in order to change destination between calls.

CHNG Call to TP PCBs

A CHNG call is issued to set or change the destination for a modifiable PCB. Issue CHNG calls to alter the destination before the ISRT calls when you need to do the following:

□ set a destination for a modifiable TP PCB

□ insert message segments in more than one message queue by using one modifiable PCB.

For example, the following SAS statements issue a CHNG call to set the destination of the third PCB in the PSB to *destname*, where *destname* must be a valid IMS/ESA transaction code or logical terminal name. This example is run using the SAS system option IMSREGTP=BMP:

```
data _null_;
  infile tranpsb dli call=func pcb=pcbindex
     status=st;
  file tranpsb dli;
  func = 'CHNG';
  pcbindex = 3;
  put @1 'destname';
  if st ¬= ' ' then
    do;
      file log;
      put _all_;
      abort;
    end;
  stop;
run;
```

The destination has been changed successfully if _ERROR_=0 after the call. Otherwise, the STATUS= variable contains a status code that indicates the reason for the failure.

If a modifiable TP PCB is used to send messages to more than one destination, the PURG call must be used to complete the current message prior to issuing a CHNG call to alter the destination for a new message. The following example shows the PURG, CHNG, and ISRT call sequence. It is run using the SAS system option IMSREGTP=BMP:

```
data _null_;
  infile tranpsb dli call=func pcb=pcbindex
     status=st;
  file tranpsb dli;
  func = 'PURG';
  pcbindex = 3;
  put;
  if st = ' ' then
    do;
      func = 'CHNG';
      put @1 '<destname>';
      if st = ' ' then
        do;
          func = 'ISRT';
          ll = 27;
          zz = '0000'x;
          msgsegm = 'Text of Message Segment';
          put @1 ll pib2.
              @3 zz
              @5 msgsegm;
          if st = ' ' then
            stop;
```

```
        else
          link abendit;
      end;
    else
      link abendit;
  end;
else
  link abendit;
return;

abendit:
  file log;
  put _all_;
  abort;
run;
```

The PCBINDEX variable points to the third PCB, which is a modifiable TP PCB. The PURG call is issued by a DL/I PUT statement. Because this PURG call only terminates the current message and does not insert a message segment, the DL/I PUT statement has no specifications. If _ERROR_=0, the PURG call is successful and the program goes on to issue a CHNG call. The destination specified for the TP PCB is changed.

If the CHNG call is successful, a message segment is built and an ISRT call is issued. The DL/I PUT statement issuing the ISRT call formats the output buffer.
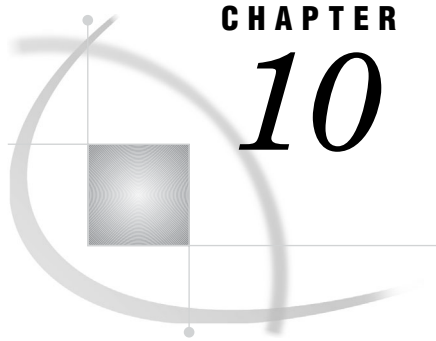
Advanced Topics for the IMS DATA Step Interface

Introduction to Advanced Topics for the IMS DATA Step Interface

This section discusses the use of the IMS DATA step interface in some of the more advanced areas of DL/I programming, specifically, restarting update programs and constructing and using SSAs in DATA step programs. Because this information is intended for experienced DL/I programmers, there is little explanation of DL/I concepts and facilities in this section. The purpose of this information is to explain how SAS programs can be used to perform advanced DL/I functions, not to explain these functions.

Restarting an Update Program

There is always a risk of abnormal termination in any program. If an update program ends before processing is completed, you can complete processing by restarting the program, but you do not want to repeat updates that have already been made. The synchronization point feature of DL/I helps to prevent duplicate updating in a restarted program.

If an online access region program or control region abends, the DL/I control region restores databases up to the last synchronization point. In a batch subsystem, a batch back-out utility must be executed to back out updates made since the last synchronization point. After backing out updates, any updates made by the program prior to the last synchronization point are intact and any made after the last synchronization point are not. When an update program is restarted after an abend, processing must resume at the synchronization point or duplicate updating might occur.

When building synchronization points into an online access region program, keep these things in mind:

□ If the program updates a large number of database records between synchronization points, the DL/I control region enqueue tables can overflow and cause the online DL/I system to abend.

□ The DL/I control region dynamic log can also overflow, which can cause the online access region or the whole online system to abend, depending on the online system used.

□ On the other hand, if synchronization points are too frequent, they can tie up the master console and prevent other IMS messages from being sent.

Your database administration staff can help you determine how frequently synchronization points should be executed.

Examples

Programs that update DL/I databases should be designed to avoid the problem of duplicating updates if a restart is required. A DATA step program can be written to be inherently restartable, as in "Example 1: Updating a Database" on page 220 or additional recovery logic can be built in for restarts, as in "Example 3: Correctly Updating a Database with Recovery Logic" on page 230.

The CHKP call, which is used in these examples, is discussed in Chapter 9, "How to Use the IMS DATA Step Interface," on page 197.

Example 1: Updating a Database

This sample program updates the ACCTDBD database with data from wire transactions in the WIRETRN database. (See Chapter 3, "Defining SAS/ACCESS Descriptor Files," on page 41 for complete database information about the WIRETRN database.) The program takes checkpoints and thereby releases database resources at regular intervals. Because the program is set up with checkpoints, it is appropriate for shared update access.

As you study this example, you will notice that the WIRETRAN segments are deleted from the WIRETRN database as soon as the ACCTDBD segments are successfully updated. There are no synchronization points between the ACCTDBD segment updates and the WIRETRAN deletions. Therefore, if an abend occurs and changes are backed out to the last synchronization point, you know that any WIRETRAN segments remaining in the database have not been processed. There is no danger of duplicating updates, and the program is inherently restartable. No special recovery logic is required for restarts.

The numbered comments following this program correspond to the numbered statements in the program:

```
data _null_;
   length ssa1 $ 43
          ssa2 $ 32
          ssa3 $ 9;
   retain blanks '                    '
          wirenum 0
          chkpnum 0;

❶  infile acctsam dli ssa=(ssa1,ssa2,ssa3) call=func
          pcb=pcbindex status=st segment=seg;
```

```
   /*  get hold next WIRETRAN segment
       from WIRETRN database         */

   func = 'GHN ';
   ssa1 = ' ';
   ssa2 = ' ';
   ssa3 = ' ';
❷  pcbindex = 5;
❸  input @1   wiressn  $char11.
          @12  wireacct $char12.
          @24  accttype $char1.
          @25  wiredate mmddyy8.
          @33  wiretime time8.
          @41  wireammt pd5.2
          @46  wiredesc $char40.;

   if st ¬= '  ' then
      if st = 'GB' then
         do;
            _error_ = 0;
            go to reptotal;
         end;
      else
         link abendit;

❹  if wirenum/5 = chkpnum then
      link chkp;

❺  amount = abs(wireammt);


   /*  insert debit or credit segment into
       ACCTDBD database         */

❻  if accttype = 'C' then
      do;
         ssa2 = 'CHCKACCT
          (ACNUMBER= '|| wireacct ||')';
         if wireammt > 0 then
            ssa3 = 'CHCKCRDT';
         else
            ssa3 = 'CHCKDEBT';
      end;
   else
❼     if accttype = 'S' then
         do;
            ssa2 = 'SAVEACCT
             (ACNUMBER= ' || wireacct || ')';
            if wireammt > 0 then
               ssa3 = 'SAVECRDT';
            else
               ssa3 = 'SAVEDEBT';
         end;
```

```
❽      else
       do;
          file log;
          put / '***** Invalid ' accttype= 'for '
            wiressn= wireacct= '*****';
          return;
       end;

❾   ssa1 = 'CUSTOMER
       (SSNUMBER= ' || wiressn || ')';
    func = 'ISRT';
    pcbindex = 4;
    file acctsam dli;

❿ put @1  amount   pd5.2
        @6  wiredate mmddyy6.
        @14 wiretime time8.
        @22 wiredesc $char40.
        @62 blanks   $char19.;

⓫  if st ¬= '  ' then
      if st = 'GE' then
         do;
            _error_ = 0;
            file log;
            if seg = 'CUSTOMER' then
               if accttype = 'C' then
                  put / '***** No CHCKACCT segment with '
                        wiressn= wireacct= '*****';
               else
                  put / '***** No SAVEACCT segment with '
                        wiressn= wireacct= '*****';
            else
               put / '***** No CUSTOMER segment with '
                     wiressn= '*****';
            return;
         end;
      else
         link abendit;


   /*  get hold checking or savings segment from
       ACCTDBD database  */

⓬  ssa3 = ' ';
    func = 'GHU';

    input @1    acnumber $char12.
          @13   balance  pd5.2
          @18   stmtdate mmddyy6.
          @26   stmt_bal pd5.2;

⓭  if st ¬= '  ' then
       link abendit;
```

```
/*  replace checking or savings segment into
    ACCTDBD database    */

balance = balance + wireammt;
ssa1 = ' ';
ssa2 = ' ';
func = 'REPL';

put @1   acnumber $char12.
    @13  balance  pd5.2
    @18  stmtdate mmddyy6.
    @26  stmt_bal pd5.2;

if st ¬= '  ' then
   link abendit;


/*  delete WIRETRAN segment from WIRETRN
    database                */
```

❹
```
  func = 'DLET';
  ssa1 = ' ';
  pcbindex = 5;
  put @1   wiressn  $char11.
      @12  wireacct $char12.
      @24  accttype $char1.
      @25  wiredate mmddyy8.
      @33  wiretime time8.
      @41  wireammt pd5.2
      @46  wiredesc $char40.;
```

❺
```
  if st ¬= '  ' then
     link abendit;
```

❻
```
  wirenum +1;
  return;
```

❼
```
  reptotal:
     file log;
     put // 'Number of Wire Transactions Posted ='
         wirenum 5.
          / '        Number of CHKP Calls Issued ='
         chkpnum 5.;
     stop;
```

❽
```
  chkp:
     chkpnum +1;
     func = 'CHKP';
     pcbindex = 1;
     file acctsam dli;
     put @1 'SAS'
         @4 chkpnum z5.;
```

```
           if st ¬= ' ' then
              link abendit;
           func = 'GHU ';
           ssa1 = 'WIRETRAN
            (SSNACCT = ' || wiressn || wireacct || ')';
           pcbindex = 5;
           input;
           if st ¬= ' ' then
              link abendit;
           return;

⑲    abendit:
        file log;
        put _all_;
        abort;
        run;
```

❶   The program uses the ACCTSAM PSB. It contains PCBs for the ACCTDBD database and a PCB for the WIRETRN database, both of which are needed in this program.

❷   PCBINDEX is set to point to the WIRETRN PCB.

❸   The INPUT statement issues the GHN call to retrieve a WIRETRAN segment. If the call is not successful, and there is a **GB** status code (end-of-database), _ERROR_ is reset to 0 and the program branches to the REPTOTAL subroutine, which prints a summary report. For any other non-blank status code, the program skips to the ABENDIT subroutine, which forces an abend.

❹   If the GHN call is successful, the program continues with a test to see if a CHKP call should be issued. Two accumulator variables, WIRENUM and CHKPNUM, are evaluated. WIRENUM is a value that is incremented each time an ACCTDBD database record is successfully updated. CHKPNUM is a value incremented each time a CHKP call is issued.

           A CHKP call is issued any time the WIRENUM value divided by five equals CHKPNUM. That is, after five successful updates the program links to the subroutine labeled CHKP to issue the CHKP call. After the CHKP call, the program repositions itself in the database and continues processing the DATA step (see item 18).

❺   The program goes on to set up for the REPL call that updates the balance information in the CHCKACCT and SAVEACCT segments of the ACCTDBD database. The absolute value of WIREAMMT is saved.

❻   The value of the ACCTTYPE field is checked. If the ACCTTYPE is **C** (checking), a qualified SSA for the CHCKACCT segment is built by concatenating literal values with the value of the WIREACCT variable from WIRETRAN. The value of WIREAMMT is checked to build another, unqualified SSA that specifies the segment type to insert. If WIREAMMT is greater than 0, the SSA specifies the CHCKCRDT segment. If WIREAMMT is less than or equal to 0, the SSA specifies CHCKDEBT.

❼
These statements are identical to the preceding group of statements, except that they build SSAs that define a savings account segment path rather than a checking account segment path.

❽
If the value of ACCTTYPE is not **C** or **S**, the account type is not valid for the DATA step and an explanatory message is written to the log. Processing returns to the beginning of the DATA step again.

❾
A qualified SSA for the CUSTOMER segment is built by concatenating literals with the value of WIRESSN from WIRETRAN. An ISRT call using the ACCTDBD PCB is set up.

❿
The ISRT call is issued. Depending on the ACCTTYPE and the value of WIREAMMT, the inserted segment is a CHCKCRDT, CHCKDEBT, SAVECRDT, or SAVEDEBT segment, as specified by the SSAs. Since all four transaction segment types have the same format, only one PUT statement is needed.

⓫
This series of statements checks the status code after the ISRT call and writes explanatory messages to the SAS log if the status code is **GE** (segment not found). If the status code is a non-blank code other than **GE**, the program skips to the ABENDIT subroutine. Note that a FILE statement is issued, changing the output destination from the DL/I database to the SAS log.

⓬
If the ISRT call is successful, the account balance must be updated to reflect the amount of the processed transaction. First, a GHU call is set up. The variable SSA3 is set to blank, but SSA1 (for the CUSTOMER segment) and SSA2 (for the CHCKACCT or SAVEACCT segment) are still in effect. The INPUT statement issues the GHU call, which retrieves the parent CHCKACCT or SAVEACCT segment for the segment just added by the ISRT call.

⓭
If the GHU call fails, the program skips to the ABENDIT subroutine. Otherwise, the program updates the BALANCE value by adding the value of WIREAMMT from the wire transaction and issues a REPL call to replace the CHCKACCT or SAVEACCT segment retrieved by the GHU call. If the REPL call fails, the program branches to the ABENDIT subroutine.

⓮
If the REPL call is successful, a DLET call is issued for the WIRETRN database. The WIRETRAN segment just used to update the ACCTDBD database (retrieved with a GHN or GHU call earlier) is deleted. Because wire transaction segments are deleted as they are processed, this program can be restarted. That is, if the program stops for some reason (such as a system failure), it can be started again without any danger of duplicate transactions being added to the ACCTDBD database.

⓯
If the DLET call is not successful, the program links to the ABENDIT subroutine.

⓰
If the DLET call is successful, the WIRENUM accumulator variable is incremented, and processing returns to the beginning of the DATA step.

⓱
This subroutine is executed when a get call to the WIRETRN database returns a **GB** (end-of-database) status code (see item 2).

⓲
This subroutine issues the CHKP call after every fifth update (see item 4). If the CHKP call is not successful, the program links to the

ABENDIT subroutine. If the CHKP call is successful, the database position has been lost. Therefore, a GHU call is set up to re-retrieve the WIRETRAN segment that is retrieved by the previous GHN call. Because the values from the segment are still in the program data vector, the INPUT statement issuing the GHU call does not need to specify variable names.

If the GHU call fails for any reason, the program links to the ABENDIT subroutine. If the call succeeds, the program resumes processing at the assignment statement that follows the LINK CHKP statement.

⓳ These statements are executed when a bad status code is returned by one of the calls in the program. The contents of the program data vector are printed on the SAS log, and the program abends.

Example 2: Incorrectly Updating a Database without Recovery Logic

Unless a program is structured so that it can be restarted without duplicating updates, special recovery logic should be included. The previous example shows a data program designed so that it can be restarted if necessary. The following example is not designed to be restarted and does not include special recovery logic. We include it as an example of the kind of program that should not be used for updating in a shared environment because it could result in erroneous data.

This program updates the ACCTDBD database with wire transactions that are stored in a sequential file rather than in the WIRETRN database. The program is similar to "Example 1: Updating a Database" on page 220, but it is not designed to be restarted. Example program 3 illustrates the modifications to this program to add recovery logic.

The numbered comments following this sample program correspond to the numbered statements in the example:

```
filename tranin '<your.sas.tranin>' disp=shr;
data _null_;
   length ssa1 $31
          ssa2 $32
          ssa3 $9;
   retain blanks '                        '
          wirenum 0
          chkpnum 0;


   /*  get data from TRANIN flatfile      */

❶ infile tranin eof=reptotal;
   input  @1   cust_ssn $char11.
          @12  acct_num $char12.
          @24  accttype $char1.
          @25  wiredate mmddyy8.
          @33  wiretime time8.
          @41  wireammt pd5.2
          @46  wiredesc $char40.;
   if _error_  then
      link abendit;

❷   if wirenum/5  = chkpnum then
      link chkp;
```

```
❸    amount = abs(wireammt);
❹    if accttype = 'C' then
       do;
           ssa2 = 'CHCKACCT
             (ACNUMBER =' || acct_num || ')';
           if wireammt < 0 then
              ssa3 = 'CHCKCRDT';
           else
              ssa3 = 'CHCKDEBT';
        end;
     else
        if accttype = 'S' then
           do;
               ssa2 = 'SAVEACCT
                 (ACNUMBER =' || acct_num || ')';
               if wireammt < 0 then
                  ssa3 = 'SAVECRDT';
               else
                  ssa3 = 'SAVEDEBT';
           end;
        else
           do;
               file log;
               put / '***** Invalid ' accttype= 'for '
                  cust_ssn=  acct_num= '*****';
               return;
           end;

     /*  insert debit or credit segment into
         ACCTDBD database         */

❺    infile acctsam dli ssa=(ssa1,ssa2,ssa3) call=func
         pcb=pcbindex status=st segment=seg;
     ssa1 = 'CUSTOMER(SSNUMBER =' || CUST_SSN || ')';
     func = 'ISRT';
     pcbindex = 4;
     file acctsam dli;
     put  @1   amount   pd5.2
          @6   wiredate mmddyy6.
          @14  wiretime time8.
          @22  wiredesc $char40.
          @62  blanks   $char19.;

❻    if st ¬= '  ' then
        if st = 'GE' then
           do;
               _error_ = 0;
               file log;
               if seg = 'CUSTOMER' then
                  if accttype = 'C' then
                     put / '***** No CHCKACCT segment with '
                           cust_ssn= acct_num= ' *****';
                  else
```

```
                     put / '***** No SAVEACCT segment with '
                           cust_ssn= acct_num= ' *****';
                 else
                     put / '***** No CUSTOMER segment with
                           ' cust_ssn= '*****';
                 return;
             end;
         else
             link abendit;


     /*  get hold checking or savings segment from
         ACCTDBD database  */

     ssa3 = '  ';
❼   func = 'GHU';
     input @1  acnumber $char12.
           @13 balance  pd5.2
           @18 stmtdate mmddyy6.
           @26 stmt_bal pd5.2;
     if st ¬= '  ' then
         link abendit;
     balance = balance + wireammt;


     /*  replace checking or savings segment into
         ACCTDBD database    */

     ssa1 = ' ';
     ssa2 = ' ';
     func = 'REPL';

❽  put @1  acnumber $char12.
        @13 balance  pd5.2
        @18 stmtdate mmddyy6.
        @26 stmt_bal pd5.2;
     if st ¬= '  ' then
         link abendit;

❾   if wireammt > 0 then
         debtnum +1;
     else
         crdtnum +1;

❿   wirenum +1;
     return;

     reptotal:
         file log;
         put // 'Number of debit  transactions posted ='
               debtnum 8.
             / 'Number of credit transactions posted ='
               crdtnum 8.;
         stop;
```

```
⓫     chkp:
        chkpnum +1;
        func = 'CHKP';
        pcbindex = 1;
        file acctsam dli;
        put @1 'SAS'
            @4 chkpnum z5.;
        if st ¬= '  ' then
            link abendit;
        return;

        abendit:
            file log;
            put _all_;
        abort;
    run;
    filename tranin clear;
```

❶   The standard INFILE statement specifies the external sequential
            file containing the data to update ACCTDBD. The fileref is TRANIN.
            When the end-of-file condition is set, the program branches to the
            REPTOTAL subroutine to print a summary report. The standard
            INPUT statement reads a record from TRANIN. If any error occurs,
            the program links to the ABENDIT subroutine.

❷   As in the previous example, this program issues CHKP calls after
            every fifth update. If the value of WIRENUM divided by five is
            equal to the value of CHKPNUM, the program links to a section
            that issues the CHKP call.

❸   The DATA step sets up for the REPL call that will update balance
            information in the CHCKACCT and SAVEACCT segments of the
            ACCTDBD database. The absolute value of WIREAMMT is saved.

❹   Depending on the value of ACCTTYPE, SSAs are built for the
            CHCKACCT and either the CHCKDEBT or CHCKCRDT segments,
            or for the SAVEACCT and either the SAVEDEBT or SAVECRDT
            segments.

❺   The DL/I INFILE statement specifies the ACCTSAM PSB. An ISRT
            call for the ACCTDBD database is formatted and issued. Depending
            on the account type and transaction type, a new CHCKCRDT,
            CHCKDEBT, SAVECRDT, or SAVEDEBT segment is inserted.

❻   This section checks status codes and prints explanatory messages on
            the SAS log if the status code is **GE** (segment not found). For other
            non-blank status codes, the program links to the ABENDIT
            subroutine.

❼   If the ISRT call is successful, a GHU call is issued to retrieve the
            parent of the added segment. The status code is checked after the
            call and, if it is not successful, the program links to the ABENDIT
            routine.

❽   If the GHU call is successful, the account balance is updated by a
            REPL call. The status code is checked after the call and, if it is not
            successful, the program links to the ABENDIT routine.

❾   Accumulator variables count the number of debits and credits posted by the program. These values are used to print a summary report.

❿   The WIRENUM variable is incremented. It is used to determine whether or not a CHKP call is needed (see item 2).

⓫   This section is like the one in "Example 1: Updating a Database" on page 220, but no GHU call is issued to re-establish database position because there is no database position to maintain. (This is because the wire transactions are not coming from an IMS database on which the program can reposition.)

Example 3: Correctly Updating a Database with Recovery Logic

This example is a modified version of "Example 2: Incorrectly Updating a Database without Recovery Logic" on page 226. The modifications consist of the recovery logic added to allow the program to be restarted. The same sequential file is used to update the ACCTDBD database.

The numbered comments following this program describe the statements added to allow a restart:

```
filename tranin '<your.sas.tranin>' disp=shr;
❶ filename restart '<your.sas.restart>' disp=shr;
data _null_;
   length ssa1 $31
          ssa2 $32
          ssa3 $9
          chkpnum 5;
   retain wireskip
          wirenum 0
          chkpnum 0
          first 1
          debtnum
          crdtnum
          errnum 0
          blanks '                   ';

   infile restart eof=process;
   input @1  chkpid   5.
         @6  chkptime datetime13.
         @19 chkdebt  8.
         @27 chkcrdt  8.
         @35 chkerr   8.;

   wireskip = chkdebt + chkcrdt + chkerr;

   file log;
   put 'Restarting from checkpoint ' chkpid
       'taken at ' chkptime datetime13.
       ' to bypass ' wireskip 'trans already processed';

   do while(wireread < wireskip);
      infile tranin;
      input  @1  cust_ssn $char11.
             @12 acct_num $char12.
             @24 accttype $char1.
```

```
                    @25 wiredate mmddyy8.
                    @33 wiretime time8.
                    @41 wireammt pd5.2
                    @46 wiredesc $char40.;
            wireread + 1;
        end;

    debtnum = chkdebt;
    crdtnum = chkcrdt;
    wirenum = debtnum + crdtnum;
    errnum = chkerr;

❷   process:
      infile tranin eof=reptotal;
      input  @1   cust_ssn $char11.
             @12  acct_num $char12.
             @24  accttype $char1.
             @25  wiredate mmddyy8.
             @33  wiretime time8.
             @41  wireammt pd5.2
             @46  wiredesc $char40.;
      if _error_ then
         link abendit;

      if wirenum/5 = chkpnum or first =1 then
         do;
             link chkp;
             first =0;
         end;

      amount = abs(wireammt);

      if accttype = 'C' then
         do;
             ssa2 = 'CHCKACCT
              (ACNUMBER= ' || acct_num || ')';
             if wireammt < 0 then
                ssa3 = 'CHCKCRDT';
             else
                 ssa3 = 'CHCKDEBT';
         end;
      else
         if accttype = 'S' then
            do;
             ssa2 = 'SAVEACCT
               (ACNUMBER= ' || acct_num || ')';
               if wireammt < 0 then
                   ssa3 = 'SAVECRDT';
               else
                   ssa3 = 'SAVEDEBT';
            end;
❸        else
           do;
               file log;
```

```
                   put / '***** Invalid ' accttype= 'for '
                       cust_ssn= acct_num= '*****';
                   go to outerr;
                 end;

          infile acctsam dli ssa=(ssa1,ssa2,ssa3) call=func
               pcb=pcbindex status=st segment=seg;

          ssa1 = 'CUSTOMER(SSNUMBER= ' || cust_ssn || ')';
          func = 'ISRT';
          pcbindex = 4;
          file acctsam dli;
          put  @1  amount   pd5.2
               @6  wiredate mmddyy6.
               @14 wiretime time8.
               @22 wiredesc $char40.
               @62 blanks   $char19.;

          if st ¬= '   ' then
             if st = 'GE' then
               do;
                  _error_ = 0;
                  file log;
                  if seg = 'CUSTOMER' then
                     if accttype = 'C' then
                       put / '***** No CHCKACCT segment with '
                             cust_ssn= acct_num= '*****';
                     else
                       put / '***** No SAVEACCT segment with '
                             cust_ssn= acct_num= '*****';
                  else
                    put / '***** No CUSTOMER segment with '
                          cust_ssn= '*****';
                  go to outerr;
               end;
             else
               link abendit;

          ssa3 = ' ';
          func = 'GHU ';
          input  @1  acnumber $char12.
                 @13 balance  pd5.2
                 @18 stmtdate mmddyy6.
                 @26 stmt_bal pd5.2;
          if st ¬= '   ' then
             link abendit;

          balance = balance + wireammt;
          ssa1 = ' ';
          ssa2 = ' ';
          func = 'REPL';
          put  @1  acnumber $char12.
               @13 balance  pd5.2
               @18 stmtdate mmddyy6.
```

```
          @26 stmt_bal pd5.2;
      if st ¬= '  ' then
         link abendit;

      if wireammt > 0 then
         debtnum = debtnum +1;
      else
         crdtnum = crdtnum +1;
      wirenum = wirenum +1;
      return;

      reptotal:
         file log;
         put // 'Number of debit  transactions posted ='
             debtnum 8.
              / 'Number of credit transactions posted ='
             crdtnum 8.;
         stop;

❹   chkp:
        chkpnum +1;
        chkptime = datetime();
        file log;
        put @1  'Next checkpoint will be'
            @25 chkpnum
            @30 chkptime datetime13.
            @43 debtnum
            @51 crdtnum
            @59 errnum;
        func = 'CHKP';
        pcbindex = 1;
        file acctsam dli;
        put @1 'SAS'
            @4 chkpnum z5.;
        if st ¬= '   ' then
           link abendit;
        return;

    outerr:
        errnum = errnum +1;
        return;

    abendit:
        file log;
        put _all_;
        abort;
run;
filename tranin clear;
filename restart clear;
```

❶   This group of statements initiates the restart, if a restart is
               necessary. The standard INFILE statement points to a file with
               fileref RESTART. The RESTART file has one record, a "control card"
               with data that will determine where processing should resume in the

sequential input file. The data in the RESTART file is taken from the last checkpoint message written on the SAS log by the program that ended before completing processing. The message includes the number and time of the last checkpoint, and the values of the accumulator variables counting the number of debit transactions posted (CHCKDEBT), credit transactions posted (CHCKCRDT), and the number of bad records in the TRANIN file (CHKERR).

The RESTART DD statement can be dummied out to execute the program normally (not as a restart). If RESTART is dummied out in the control language, end-of-file occurs immediately, and the program skips to the PROCESS subroutine (see item 6), as indicated by the EOF= option.

The WIRESKIP variable is the sum of CHCKDEBT, CHCKCRDT, and CHKERR; that is, WIRESKIP represents the number of records in TRANIN that were processed by the program before the last checkpoint.

A message is written to the SAS log that shows the checkpoint from which processing resumes.

To position itself at the correct TRANIN record, the program reads the number of records indicated by the WIRESKIP variable. In other words, the program re-reads all records that were read in the first execution of the program, up to the last checkpoint.

The values of DEBTNUM, CRDTNUM, WIRENUM, and ERRNUM are reset so that the final report shows the correct number of transactions. Otherwise, the report would show only the number of transactions processed in the restarted execution.

❷ These statements are the same as the statements in "Example 2: Incorrectly Updating a Database without Recovery Logic" on page 226 except that they are labeled "PROCESS." If the program is not being restarted, end-of-file for the INFILE RESTART occurs immediately, and the program branches to this subroutine.

❸ If the value of ACCTTYPE is anything but **c** or **s**, the TRANIN record is a bad record. The program prints a message on the SAS log and branches to the OUTERR subroutine, which increments the ERRNUM accumulator variable.

❹ The CHKP call is issued by this group of statements. This group is like that in "Example 2: Incorrectly Updating a Database without Recovery Logic" on page 226 except that a message about the checkpoint is also printed on the SAS log. This message provides the necessary information for a restart.

Note that the message is written to the SAS log before the CHKP call is actually issued, so it is possible that a system failure could occur between the time the message is written and the time the call is issued. Therefore, if a restart is necessary, you should verify that the last checkpoint referenced in the SAS log is the same as the last checkpoint in the DL/I log. This can be done by comparing checkpoint IDs.

SSAs in IMS DATA Step Programs

When a DATA step program uses qualified calls, you designate variables containing the SSAs with the SSA= option in the DL/I INFILE statement. The values of SSA variables do not have to be constants. They can be built by the program using SAS assignment statements, functions, and operators. You can construct SSAs conditionally and change SSA variable values between calls.

The Concatenation Operator

One of the techniques for building an SSA is to incorporate the value of another variable in the SSA variable's value. This can be accomplished with the concatenation operator (||), as in this example:

```
ssa1='CUSTOMER(SSNUMBER ='||ssn||')';
```

This statement assigns a value to SSA1 that consists of the literal CUSTOMER(SSNUMBER =, the current value of the variable SSN, and the right parenthesis. If the current value of SSN is 303-46-4887, the SSA is

```
CUSTOMER(SSNUMBER =303-46-4887)
```

Note: The concatenation operator acts on character values. If you use a numeric variable or value with the concatenation operator, the numeric value is converted automatically to character using the BEST12. format. If the value is less than 12 bytes, it is padded with blanks and, if longer than 12 bytes, it could lose precision when converted. If you want to insert a numeric value via concatenation, you should explicitly convert the value to character with the PUT function (described in the next section). △

The PUT Function

SSA variables in a DATA step program must be character variables. However, you might sometimes need to qualify an SSA with a numeric value. To insert a numeric value in an SSA character variable, you can use the SAS PUT function.* For more information about the PUT statement, see *SAS Language Reference: Dictionary*.

The PUT function's form is as follows:

PUT(*argument1, format*)

Argument1 is a variable name or a constant, and *format* is a valid SAS format of the same type (numeric or character) as *argument1*. The PUT function writes a character string that consists of the value of *argument1* output in the specified format. The result of the PUT function is always a character value, regardless of the type of the function's arguments. For example, in the following statement the result of the PUT function is a character string assigned to the variable NEWDATE, a character variable.

```
newdate=put(datevalu,date7.);
```

The result is a character value even though DATEVALU and the DATE7. format are numeric. If DATEVALU=38096, the value of NEWDATE is:

```
newdate='20APR64'
```

---

* The PUT function can also be used to format a character value with any valid character format.

Using the PUT function, you can translate numeric values for use in SSAs. For example, to select WIRETRAN segments with WIREAMMT values less than $500.00, you could construct an SSA like this:

```
maxamt=500;
ssa1='WIRETRAN(WIREAMMT <'||put(maxamt,pd5.2)||')';
```

First, you assign the numeric value to be used as the search criterion to a numeric variable. In this case, the value 500 is assigned to the numeric variable MAXAMT. Then you construct the qualified SSA using concatenation and the PUT function. The PUT function's result is a character string consisting of the value of MAXAMT in PD5.2 format.

Consider a more complicated example using the ACCTDBD database. In this case, you want to select all checking accounts for which the last statement was issued a month ago today or more than 31 days ago.

The following SAS statements illustrate one approach to constructing an SSA to select the appropriate accounts. The numbered comments after this example correspond to the numbered statements:

```
data _null_;
❶   tday = today();
❷   d = day(tday);
    m = month(tday);
    y = year(tday);

❸   if d = 31 then
       if m = 5 or
          m = 7 or
          m = 10 or
          m = 12 then
          d = 30;
❹   if m = 3 then
       if d < 28 then
          d = 28;
    if m = 1 then
       do;
          m = 12;
          y = y - 1;
       end;
    else
       m = m - 1;

❺   datpmon = mdy(m,d,y);
❻   datem31 = tday - 31;

❼   ssa1 = 'CHCKACCT
       (STMTDATE= ' || put(datpmon,mmddyy6.) ||
       '| STMTDATE> ' || put(datem31,mmddyy6.) || ')';
    stop;
run;
```

❶   Use the SAS function TODAY to produce the current date as a SAS date value and assign it to the variable TDAY.

❷   Use the SAS functions DAY, MONTH, and YEAR to extract the corresponding parts of the current date and assign them to appropriate variables.

❸   Modify D values to adjust when previous month has fewer than 31 days.

❹   Modify the month variable (M) to contain the prior month value.

❺   Assign the SAS date value for last month, the same day as today, to the variable DATPMON.

❻   Subtract 31 from the SAS date representing today's date and assign the value to the variable DATEM31.

❼   To build the SSA, concatenate these elements:

         □ a literal that is composed of the segment name (CHCKACCT), a left parenthesis, search field name (STMTDATE), and the relational operator =.

         □ a character string consisting of the value of DATPMON output in the MMDDYY6. format. The character string is the result of the PUT function.

         □ a literal consisting of the Boolean operator | (or), the search field name (STMTDATE), and the relational operator >.

         □ a character string consisting of the value of DATEM31 output in the MMDDYY6. format. The character string is the result of the PUT function.

         □ a literal consisting of a right parenthesis.

If these statements are executed on 28 March 1995, the value of SSA1 is

```
CHCKACCT(STMTDATE =02/28/95|STMTDATE >02/28/95)
```

Setting SSAs Conditionally

Using SAS IF-THEN/ELSE statements, SSA variables can be assigned values conditionally. Consider "Example 2: Incorrectly Updating a Database without Recovery Logic" on page 226 in which the ACCTDBD database is updated with transaction information stored in a standard sequential file with fileref TRANIN. Each TRANIN record contains data for one deposit or withdrawal transaction for a checking or savings account. The program uses the TRANIN records to construct new CHCKDEBT, CHCKCRDT, SAVEDEBT, or SAVECRDT segments and then inserts the new segment in the ACCTDBD database. Notice that the concatenation operator (||) is used to incorporate the value of the ACCT_NUM variable in the SSA.

The program first reads a record from the TRANIN file and then determines whether the data is for a checking or a savings account by evaluating the value of the variable ACCTTYPE. If ACCTTYPE='C', the program constructs a qualified SSA for a CHCKACCT segment. Next, the program determines whether the record represents a debit or credit transaction and builds an unqualified SSA for a CHCKDEBT or CHCKCRDT segment, as appropriate.

If ACCTTYPE='S', a qualified SSA for a SAVEACCT segment is built, and then an unqualified SSA for a SAVEDEBT or SAVECRDT segment is set up.

Changing SSA Variable Values between Calls

A DATA step program can issue multiple calls within a DATA step execution, and the value of an SSA variable can be changed between each call. An example of this is the following code, which is used in "Example 4: Issuing REPL Calls" on page 181 in Chapter 9, "How to Use the IMS DATA Step Interface," on page 197:

```
data _null_;
   set ver6.newaddr;
   length ssa1 $31;
   infile acctsam dli ssa=ssa1 call=func status=st
     pcbno=4;
   ssa1 = 'CUSTOMER(SSN =' || ssn || ')';
   func = 'GHU ';
   input;
   if st = '  ' then
      do;
         func = 'REPL';
         ssa1 = ' ';
         file acctsam dli;
         put _infile_ @;
         put @52 newaddr1  $char30.
             @82 newaddr2  $char30.
             @112 newcity  $char28.
             @140 newstate $char2.
             @162 newzip   $char10.;
         if st ¬= '  ' then
            link abendit;
      end;
   else
      if st = 'GE' then
         do;
            _error_ = 0;
            stop;
         end;
      else
         link abendit;
   return;

   abendit:
      file log;
      put _all_;
      abort;
run;
```

These statements are part of a program that updates CUSTOMER segments in the ACCTDBD database with information from the SAS data set VER6.NEWADDR. CUSTOMER segments are retrieved using GHU calls with a qualified SSA, SSA1. Once a segment is retrieved, the data from the SAS data set is overlaid on the old values of the segment and a REPL call is issued. Since a REPL call acts on a segment retrieved previously, no SSA is needed. Therefore, the value of the SSA1 variable is changed to blanks before the REPL call is issued.
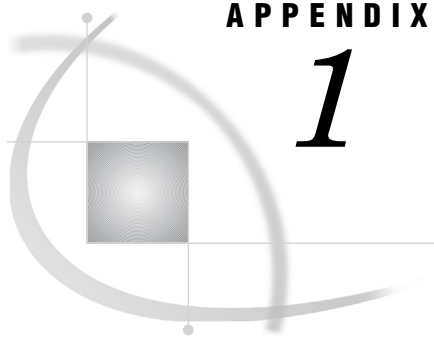
A P P E N D I X

1

SAS System Options for IMS Databases

Introduction to SAS System Options for IMS Databases

The SAS/ACCESS interface to IMS uses a group of SAS system options to specify the type of DL/I region through which DL/I calls are executed and to provide the DL/I region execution parameters. For example, either the IMSREGTP= option or its alias, DLIRGNTP=, specifies the type of DL/I region to be invoked. Appropriate defaults are assigned for the system options when the SAS/ACCESS interface is installed at a site.

The next section provides more information about how to use SAS options for IMS. In later sections, the options are divided into two sections according to their operating system. Each reference section includes a quick reference table for the options, followed by a longer description of each option.

Specifying System Options

The system options described in this appendix fall into two categories:

- *Invocation* options are processed when SAS is initialized. They can be specified in the following ways:

  - in the default OPTIONS table

  - in a system or user configuration file

  - in the OPTIONS parameter of the host command that you use to invoke SAS at your site.

- *Session* options can be specified when SAS is invoked, in the configuration file, or in an OPTIONS statement.

"SAS System Options for IMS" on page 241 indicates whether each option is an invocation option or a session option.

Restrictable Options

The DBA or SAS Software Representative at your installation might choose to restrict an *invocation* option to a particular value for security or data integrity reasons. *Session* options cannot be restricted; you can override them any time during a SAS session by using an OPTIONS statement. If you try to override a restricted invocation option, you get an error message.

You can use two methods to determine which invocation options are restricted at your installation:

☐ Ask the SAS Software Representative or DBA who installed the SAS/ACCESS interface to IMS at your site.

☐ Invoke SAS with the VERBOSE option. In the list of options that appears, the restricted invocation options (if any) follow the VERBOSE option.

Note: From a TSO session under OS/390, the list of options might be displayed only briefly on your terminal before the SAS session comes up. In this case, you will need to exit SAS in order to see the list of options. △

Displaying the Current Values of the Options

To check your installation's current settings for the SAS system options for IMS, check the settings for the options (except for DLIREAD and BMPREAD) by executing PROC OPTIONS with the IMS option:

```
proc options ims;
run;
```

To see the values of DLIREAD and BMPREAD, use PROC OPTIONS without the IMS option. The OPTIONS procedure is documented in the *Base SAS Procedures Guide*.

Overriding Option Defaults

Most option defaults are probably correct for your applications, and you might never need to override the default settings. In fact, many of the options might specify information that is unfamiliar to a DL/I applications programmer. However, if you decide that one or more of the defaults is not appropriate for your IMS application, you can override the default value(s) as follows:

☐ You can override the default value of an *unrestricted invocation* option in any of the following ways:

   ☐ when you invoke SAS

   ☐ in the system configuration file (if one is used)

   ☐ in the user configuration file (if one is specified)

   ☐ in the default options (DFLTOPTS) table

   ☐ in the OPTIONS parameter at invocation of the CLIST (TSO)

   ☐ in the OPTIONS parameter in the cataloged procedure (batch only).

Note: You cannot override the default value of any invocation option, whether restricted or unrestricted, *during a SAS session*. For example, your installation might specify that the invocation option DLIREAD= is unrestricted, and set the value of that option at **Y** so that programs using a batch region can issue only get calls by default. Because DLIREAD= is not restricted, you can specify the following at invocation time to override the default value of **Y**.

```
DLIREAD=N;
```

However, you cannot override the value of DLIREAD= during a SAS session. △

☐ You can override the default value of a *session* option by specifying the option in any of the following ways:

    ☐ in an OPTIONS statement

    ☐ in the system configuration file (if one is used)

    ☐ in the user configuration file (if one is specified)

    ☐ in the Default Options (DFLTOPTS) table

    ☐ in the OPTIONS parameter at invocation of the CLIST (TSO)

    ☐ in the OPTIONS parameter in the cataloged procedure (batch only)

For more information about overriding SAS system options, see the SAS companion for your operating system.

Most Frequently Altered Options

If you need to override any options, it will probably be the following:

IMSREGTP=
specifies the type of DL/I region that is used to execute DL/I calls. It is altered whenever you want to execute calls through a DL/I region that is not the installation default.

If you use a batch DLI or DBB region, you are not likely to alter any other system option. If you use an online access region (BMP), you might need to change one or more of these options:

IMSBPIN= IMSBPOUT=
identify message queues for access in advanced DL/I programming when running a BMP region. This option is valid only for the IMS DATA step interface.

IMSID=
identifies the IMS subsystem that contains the databases that you want to access. You might need to use this option with a BMP, DLI, or DBB region in order to specify a test or production system.

SAS System Options for IMS

Since Version 6, the first three letters of all SAS system options for IMS are IMS. If the option applies only to certain region types, the next two letters indicate the region type: DL for DLI or DBB and BP for BMP. Some options such as IMSSPIE apply to more than one region type. These options do not include one of the region-type codes.

Most of the SAS system options for IMS parallel the functions of DL/I parameters. For example, the option IMSBPAGN= specifies a value for the AGN parameter, which is used in BMP regions. If you need more information about IMS parameters, refer to your IBM documentation.

Note: For the DATA step interface, the SLI region type–and hence, the SLICWTO, SLIREAD, and CICSID options–are no longer supported. Sites that use CICS as opposed to IMS/DC can gain access to CICS dedicated databases by using the IMS-ESA or CICS-ESA DBCTL feature. This feature enables an application like SAS software to access the databases through a BMP region. △

Detailed Reference for Options

The following sections describe the SAS system options for IMS.

BMPREAD= SAS System Option

Specifies whether a SAS IMS program accessing databases is restricted to get calls within a BMP region.

Syntax: BMPREAD=N|Y

Default: BMPREAD=N specifies that programs are not restricted to get calls, and update calls can be issued within a BMP region.

Valid as part of: SAS invocation

Interface(s):   DATA step

Restrictable: Y

Details: BMPREAD=Y causes SAS to return a status code of SE and to set _ERROR_=1 when a DL/I update call is issued.

DLIREAD= SAS System Option

Specifies whether a SAS IMS program accessing databases is restricted to get calls within a DLI region.

Syntax: DLIREAD=N|Y

Default: DLIREAD=N specifies that programs are not restricted to get calls, and update calls can be issued within a DLI region.

Valid as part of: SAS invocation

Interface(s):   DATA step

Restrictable: Y

Details: DLIREAD=Y causes SAS to return a status code of SE and to set _ERROR_ =1 when a DL/I update call is issued.

IMSBPAGN= SAS System Option

Specifies a value for the AGN (application group name) parameter.

Syntax: IMSBPAGN=*value* or BMPAGN=*value*

Default: IMSBPAGN=* specifies that the AGN parameter is null in the attach parameter list.

Valid as part of: SAS invocation or in the OPTIONS statement

Interface(s): Engine, DATA step

Restrictable: N

Details: The AGN parameter can be used to limit BMP region execution to particular PSBs.

IMSBPCPU= SAS System Option

Specifies a value for the CPUTIME parameter.

Syntax: IMSBPCPU=*value* or BMPCPUTM=*value*

Default: IMSBPCPU=0 specifies that no task timing is to be done for the BMP region (also the default IMS action). A non-zero value specifies a maximum number of minutes allowed for execution of the BMP region.

Valid as part of: SAS invocation or in the OPTIONS statement

Interface(s): Engine, DATA step

Restrictable: N

IMSBPDCA= SAS System Option

Specifies a value for the DIRCA parameter.

Syntax: IMSBDCA=*value* or BMPDIRCA=*value*

Default: IMSBPDCA=0 enables IMS/ESA to calculate the maximum size of the dependent region inter-region communication area required by any non-dynamic PSB in the control region.

Valid as part of: SAS invocation or in the OPTIONS statement

Interface(s): Engine, DATA Step

Restrictable: N

Details: A non-zero value specifies the size of the DIRCA in 1K blocks. Valid values range from 0 to 99.

IMSBPIN= SAS System Option

Assigns a value for the IN parameter, which specifies the TRANCODE of the message queue that is accessed.

Syntax: IMSBPIN=*value* or BMPIN=*value*

Default: IMSBPIN=* specifies that the IN parameter is null in the attach parameter list and that no transaction queue is to be read.

Valid as part of: SAS invocation or in the OPTIONS statement

Interface(s):   DATA step

Restrictable: N

Details: Use this option only when you intend to read from transaction queues with the I/O PCB.

IMSBPNBA= SAS System Option

Specifies a value for the NBA parameter.

Syntax: IMSBPNBA*value* or BMPNBA=*value*

Default: IMSBPNBA=0

Valid as part of: SAS invocation or in the OPTIONS statement

Interface(s):   DATA step

Restrictable: N

Details: The NBA parameter specifies the number of Fast Path database buffers made available in the common service area. Valid values range from 0 to 999. Unless Fast Path databases is used, this value should be 0.

IMSBPOBA= SAS System Option

Specifies a value for the OBA parameter.

Syntax: IMSBPOBA=*value* or BMPOBA=*value*

Default: IMSBPOBA=0

Valid as part of: SAS invocation or in the OPTIONS statement

Interface(s):   DATA step

Restrictable: N

Details: The OBA parameter specifies the number of additional page-fixed Fast Path database buffers made available if the normal allotment is used. Valid values range from 0 to 999. Unless Fast Path databases is used, this value should be 0.

IMSBPOPT= SAS System Option

Assigns a character value to the OPT parameter.

Syntax: IMSBPOPT=C | N | W or BMPOPT=C | N | W

Default: IMSBPOPT=C specifies that the BMP region is canceled automatically.

Valid as part of: SAS invocation or in the OPTIONS statement

Interface(s):   Engine, DATA step

Restrictable: N

Details: The OPT parameter specifies the action taken if the control region is not active. IMSBPOPT=N specifies that the console operator is asked for a decision. IMSBPOPT=W specifies that the region waits for the control region to be started.

IMSBPOUT= SAS System Option

Specifies a value for the OUT parameter.

Syntax: IMSBPOUT=*value* or BMPOUT=*value*

Default: IMSBPOUT=* specifies that the OUT parameter is null in the attach parameter list.

Valid as part of: SAS invocation or in the OPTIONS statement

Interface(s): DATA step

Restrictable: N

Details: The OUT parameter specifies the TRANCODE or LTERM that is the destination of a message insert. Specify this option only if you intend to write to the IMS/ESA message queue with the I/O PCB, and the IN parameter is not specified.

IMSBPPAR= SAS System Option

Specifies the value of the PARDLI parameter.

Syntax: IMSBPPAR=0 | 1 or BMPARDLI=0 | 1

Default: IMSBPPAR=0 specifies that DL/I processing is performed within the BMP region (also the IMS default action).

Valid as part of: SAS invocation or in the OPTIONS statement

Interface(s): Engine, DATA step

Restrictable: N

Details: IMSBPPAR=1 specifies that all IMS processing for the BMP region is performed in the IMS/ESA control region.

IMSBPSTI= SAS System Option

Specifies whether the BMP timer is set.

Syntax: IMSBPSTI=0 | 1 or BMPSTIMR=0 | 1

Default: IMSBPSTI=0 specifies that the BMP timer is not set (also the IMS default action).

Valid as part of: SAS invocation or in the OPTIONS statement

Interface(s): Engine, DATA step

Restrictable: N

Details: IMSBPSTI=1 specifies that the BMP timer is set.

IMSBPUPD= SAS System Option

Specifies whether a SAS IMS program that accesses databases can issue update calls in a BMP region.

Syntax: IMSBPUPD=Y | N

Default: IMSBPUPD=Y allows update processing of databases within a BMP region.

Valid as part of: SAS invocation

Interface(s): Engine

Restrictable: Y

Details: IMSBPUPD=N causes SAS to return an error message that indicates that you are not authorized to update the database if an update call is issued.

IMSDEBUG= SAS System Option

Specifies whether the DL/I call function code, segment search arguments, and status code returned from DL/I calls issued by the IMS engine should be displayed in the SAS log.

Syntax: IMSDEBUG=N | Y | 1 to 9999

Default: IMSDEBUG=N causes no DL/I calls to be displayed.

Valid as part of: SAS invocation or in the OPTIONS statement.

Interface(s): Engine, DATA step

Restrictable: N

Details: A value of Y causes the first 50 DL/I calls to be displayed. A value of *nnnn* (where *nnnn* is a number between 1 and 9999) causes the first *nnnn* DL/I calls to be displayed.

IMSDLBKO= SAS System Option

Determines the value of the BKO parameter when SAS invokes an IMS/ESA DLI or DBB region.

Syntax: IMSDLBKO=* | Y | N or DLIBKO=* | Y | N

Default: IMSDLBKO=* specifies that the BKO parameter is null in the IMS region parameter list, so the default IMS action is taken.

Valid as part of: SAS invocation or in the OPTIONS statement.

Interface(s): Engine, DATA step

Restrictable: N

Details: The BKO parameter setting determines whether or not updates in a disk log are backed out automatically if the program abends. When IMSDLBKO=Y and the SAS session abends, all database updates since the last CHKP call are backed out automatically unless the system crashed. (If IMSDLBKO=Y, a DASD log data set must be used.)

IMSDLBUF= SAS System Option

Specifies a value for the BUF parameter.

Syntax: IMSDLBUF=*value* or DLIBUF=*value*

Default: IMSDLBUF=16

Valid as part of: SAS invocation or in the OPTIONS statement.

Interface(s): Engine, DATA step

Restrictable: N

Details: The BUF parameter specifies the number of 1K blocks that are available in the ISAM/OSAM buffer pool. Valid values range from 0 to 999. When the DFSVSAMP DD control statements are used, they override the specification.

IMSDLDBR= SAS System Option

Determines the value used as the DBRC (database recovery control facility) parameter when SAS invokes a DLI or DBB region.

Syntax: IMSDLDBR=* | Y | N or DLIDBRC=* | Y | N

Default: IMSDLDBR=* specifies that the DBRC parameter is null in the IMS/ESA parameter list, so the default IMS action is taken.

Valid as part of: SAS invocation or in the OPTIONS statement.

Interface(s): Engine, DATA step

Restrictable: N

Details: When IMSDLDBR=Y, DBRC is used during execution of IMS/ESA (the default IMS action if IMS/ESA is generated with DBRC). IMSDLDBR=N specifies that DBRC is not to be used in the execution of IMS/ESA.

IMSDLEXC= SAS System Option

Specifies a value for the EXCPVR parameter.

Syntax: IMSDLEXC=0 | 1 or DLIEXCPV=0 | 1

Default: IMSDLEXC=0 specifies that the ISAM/OSAM database buffer pool is not to be long-term page-fixed.

Valid as part of: SAS invocation or in the OPTIONS statement.

Interface(s): Engine, DATA step

Restrictable: N

Details: IMSDLEXC=1 specifies that the ISAM/OSAM database buffer pool is to be long-term page-fixed.

IMSDLFMT= SAS System Option

Specifies a value for the FMTO parameter.

Syntax: IMSDLFMT=P | T | N or DLIFMT=P | T | N

Default: IMSDLFMT=P ignores processing of the FDDL table.

Valid as part of: SAS invocation or in the OPTIONS statement

Interface(s): Engine, DATA step

Restrictable: N

Details: IMSDLFMT=T specifies that a formatted dump will contain IMS/ESA data areas and that the formatted dump delete list (FDDL) is processed (the default IMS action). IMSDLFMT=N suppresses production of a formatted dump.

IMSDLIRL= SAS System Option

Determines the value of the IRLM parameter when SAS invokes a DLI or DBB region.

Syntax: IMSDLIRL=* | Y | N or DLIRLM=* | Y | N

Default: IMSDLIRL=* specifies that the IRLM parameter is null in the IMS/ESA parameter list so that the default IMS action is taken.

Valid as part of: SAS invocation or in the OPTIONS statement.

Interface(s): Engine, DATA step

Restrictable: N

Details: IMSDLIRL=Y specifies that IRLM is to be used in this execution of IMS/ESA (the default IMS action if IMS/ESA was generated with IRLM). IMSDLIRL=N specifies that IRLM is not to be used in this execution of IMS/ESA.

IMSDLIRN= SAS System Option

Specifies an IRLM subsystem name.

Syntax: IMSDLIRN=* | *name* or DLIRLMNM=* | *name*

Default: IMSDLIRN=* specifies that the parameter is null in the attach parameter list, and no IRLM subsystem is used.

Valid as part of: SAS invocation or in the OPTIONS statement.

Interface(s):   Engine, DATA step

Restrictable: N

Details: The name is used at initialization to specify the IRLM subsystem name when IMSDLIRL=Y.

IMSDLLOG= SAS System Option

Specifies a value for the LOGA parameter.

V5 Alias: DLILOGA=

Syntax: IMSDLLOG=0 | 1 or DLILOGA=0 | 1

Default: IMSDLLOG=0 specifies that BSAM be used to access the IEFRDER log data set.

Valid as part of: SAS invocation or in the OPTIONS statement.

Interface(s):   Engine, DATA step

Restrictable: N

Details: If IMSDLLOG=1 is specified, OSAM is used to access the IEFRDER log data set.

IMSDLMON= SAS System Option

Specifies a value for the MON parameter.

Syntax: IMSDLMON=N | Y or DLIMON=N | Y

Default: IMSDLMON=N specifies that DB Monitor output is not produced (also the default IMS action).

Valid as part of: SAS invocation or in the OPTIONS statement.

Interface(s):   Engine, DATA step

Restrictable: N

Details: IMSDLMON=Y produces DB Monitor records on the IMSMON file (if allocated), or on the IEFRDER log if the IMSMON file is not allocated.

IMSDLSRC= SAS System Option

Specifies a value for the SRCH parameter.

Syntax: IMSDLSRC=0 | 1 or DLISRCH=0 | 1

Default: IMSDLSRC=0 specifies a standard module search for directed load.

Valid as part of: SAS invocation or in the OPTIONS statement.

Interface(s):   Engine, DATA step

Restrictable: N

Details: IMSDLSRC=1 specifies that the job pack area (JPA) and link pack area (LPA) are searched before a PDS in a directed load.

IMSDLSWP= SAS System Option

Determines the value of the SWAP parameter when SAS invokes an IMS/ESA DLI or DBB region.

V5 Alias: DLISWP=

Syntax: IMSDLSWP=* | Y | N or DLISWP=* | Y | N

Default: IMSDLSWP=* specifies that the SWAP parameter is null in the IMS/ESA parameter list so that the default IMS action is taken.

Valid as part of: SAS invocation or in the OPTIONS statement.

Interface(s):   Engine, DATA step

Restrictable: N

Details: A value of IMSDLSWP=Y indicates that the address space is swappable; IMSDLSWP=N indicates that the address space is not swappable.

IMSDLUPD= SAS System Option

Specifies whether or not a SAS IMS program that accesses databases through the engine can issue update calls within a DLI or DBB region.

Syntax: IMSDLUPD=Y | N

Default: IMSDLUPD=Y allows update processing of databases within a DLI or DBB region.

Valid as part of: SAS invocation

Interface(s):   Engine

Restrictable: Y

Details: IMSDLUPD=N causes SAS to return an error message indicating that you are not authorized to update the database if an update call is issued.

IMSID= SAS System Option

Specifies a value for the IMSID parameter (the subsystem identifier) when SAS attaches a BMP, DLI, or DBB region.

Syntax: IMSID=* | *value*

Default: IMSID=* specifies that the parameter is null in the attach parameter list, and therefore the identifier specified at IMS/ESA definition is used.

Valid as part of: SAS invocation

Interface(s):   Engine, DATA step

Restrictable: Y

Details: The value is used in initialization to establish communication with the control region that has the same IMSID value.

IMSIOB= SAS System Option

Specifies a value for the number of OSAM I/O requests that can be active concurrently.

Syntax: IMSIOB=* | 999

Default: IMSIOB=* specifies that the value is null in the attach parameter list.

Valid as part of: SAS invocation or in the OPTIONS statement.

Interface(s):   Engine, DATA step

Restrictable: N

Details: For IMS/VS Release 2.2.0, enter a value of 999 to disable OSAM processing. This option is provided by an IBM APAR and PTF to eliminate CSA shortages due to the allocation of OSAM blocks.

IMSREGTP= SAS System Option

Determines the type of IMS region invoked.

V5 Alias: DLIRGNTP=

Syntax: IMSREGTP=DLI | DBB | BMP or DLIRGNTP=DLI | DBB | BMP

Default: IMSREGTP=DLI

Valid as part of: SAS invocation

Interface(s):   Engine, DATA step

Restrictable: Y

Details: You can specify IMSREGTP=DLI or IMSREGTP=DBB to invoke a batch region using PSB or ACB libraries.

IMSSPIE= SAS System Option

Specifies a value for the SPIE parameter when SAS invokes a DLI, DBB, or BMP region.

Syntax: IMSSPIE=0 | 1 or DLISPIE=0 | 1

Default: IMSSPIE=0 enables a user SPIE (if any) to remain in effect while processing DL/I calls.

Valid as part of: SAS invocation or in the OPTIONS statement.

Interface(s):   Engine, DATA step

Restrictable: N

Details: IMSSPIE=1 negates the user SPIE while processing the DL/I calls but reinstates it prior to returning to the application program.

IMSTEST= SAS System Option

Specifies a value for the TEST parameter when SAS invokes a DLI, DBB, or BMP region.

Syntax: IMSTEST=0 | 1 or DLITEST=0 | 1

Default: IMSTEST=0 specifies that the addresses in the user call lists are not checked for validity.

Valid as part of: SAS invocation or in the OPTIONS statement.

Interface(s):   Engine, DATA step

Restrictable: N

Details: When IMSTEST=1, the addresses in the user call lists are checked for validity.

IMSWHST= SAS System Option

Specifies whether or not the IMS engine should retrieve records if qualified segment search arguments are not generated to be passed to IMS.

Syntax: IMSWHST=N | Y

Default: IMSWHST=N specifies that records should be retrieved for processing regardless of whether or not qualified segment search arguments are passed to IMS.

Valid as part of: SAS invocation

Interface(s):   Engine

Restrictable: Y

Details: IMSWHST=Y specifies that records should be retrieved for processing only if qualified segment search arguments are passed to IMS.

Quick Reference for Options

The following table summarizes SAS system options for IMS.

Table A1.1 SAS System Options for IMS

V6 and later Option	Default	Invocation	Session	Restrict	Engine	DATA Step
	N	Y		Y		Y
	N	Y		Y		Y
IMSBPAGN	*	Y	Y		Y	Y
IMSBPCPU	0	Y	Y		Y	Y
IMSBPDCA	0	Y	Y		Y	Y
IMSBPIN	*	Y	Y			Y
IMSBPNBA	0	Y	Y			Y
IMSBPOBA	0	Y	Y			Y
IMSBPOPT	C	Y	Y		Y	Y
IMSBPOUT	*	Y	Y			Y
IMSBPPAR	0	Y	Y		Y	Y
IMSBPSTI	0	Y	Y		Y	Y
IMSBPUPD	Y	Y		Y	Y	
IMSDEBUG	N	Y	Y		Y	Y
IMSDLBKO	*	Y	Y		Y	Y
IMSDLBUF	16	Y	Y		Y	Y
IMSDLDBR	*	Y	Y		Y	Y
IMSDLEXC	0	Y	Y		Y	Y
IMSDLFMT	P	Y	Y		Y	Y
IMSDLIRL	*	Y	Y		Y	Y
IMSDLIRN	*	Y	Y		Y	Y
IMSDLLOG	0	Y	Y		Y	Y
IMSDLMON	N	Y	Y		Y	Y
IMSDLSRC	0	Y	Y		Y	Y
IMSDLSWP	*	Y	Y		Y	Y
IMSDLUPD	Y	Y		Y	Y	
IMSID	*	Y		Y	Y	Y
IMSIOB	*	Y	Y		Y	Y
IMSREGTP	DLI	Y		Y	Y	Y
IMSSPIE	0	Y	Y		Y	Y
IMSTEST	0	Y	Y		Y	Y
IMSWHST	N	Y		Y	Y	
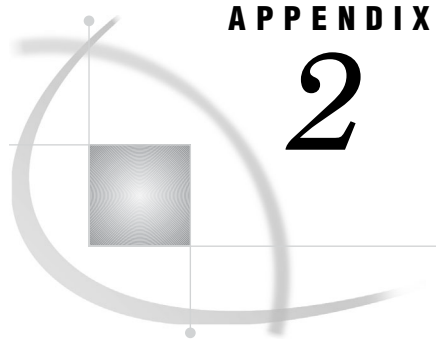
Example Data

Introduction to IMS Example Data

This appendix lists the data in the sample IMS databases ACCTDBD and EMPLINF2, and the DB2 table BANKCHRG that are used in the examples in this document. It also includes the data in the descriptor files and SAS data files that are used in the examples in Chapter 4, " IMS Data in SAS Programs," on page 49 and Chapter 5, "Browsing and Updating IMS Data," on page 71. See Chapter 3, "Defining SAS/ACCESS Descriptor Files," on page 41 for complete information about the WIRETRN database.

Sample JCL for allocating the IMS databases, creating DBDs, creating PSBs, and creating needed flat files is provided in the SAS Sample Library files. If you want to run these examples, see "About the Example Data in the Document" on page 8 or

contact your SAS Software Representative for information about how to access the files in the SAS Sample Library provided with this release.

Access Descriptors for IMS

ACCTDBD Database Access Descriptor

This section describes the MYLIB.ACCOUNT access descriptor for the ACCTDBD database that is used in the examples. This section provides the statements used to create the ACCOUNT access descriptor in batch, interactive line, or noninteractive mode. The ACCTDBD database is described in detail in Chapter 2, "IMS Essentials," on page 11.

```
JCL statements;
proc access dbms=ims;
  create mylib.account.access;
    dbd=acctdbd dbtype=hdam;
    record='customer_record' sg=customer sl=225;
      item=soc_sec_number lv=2  dbf=$11.
                          key=u
                          se=ssnumber;
      item=customer_name  lv=2  dbf=$40.
                          se=custname;
      item='address info' lv=2;
      item=addr_line_1    lv=3  dbf=$30.
                          se=custadd1;
      item=addr_line_2    lv=3  dbf=$30.
                          se=custadd2;
      item=city           lv=3  dbf=$28.
                          se=custcity;
      item=state          lv=3  dbf=$2.
                          se=custstat;
      item=country        lv=3  dbf=$20.
                          se=custland;
      item=zip_code       lv=3  dbf=$10.
                          se=custzip;
      item=home_phone     lv=2  dbf=$12.
                          se=custhphn;
      item=office_phone   lv=2  dbf=$12.
                          se=custophn;

    record='checking_account_record' sg=chckacct
          sl=40;
      item=check_account_number lv=2  dbf=12.
                                key=u
                                se=acnumber;
      item=check_amount         lv=2  dbf=pd5.2
                                se=stmtamt
                                dbc=l;
```

```
    item=check_date              lv=2  dbf=6.0
                                 fmt=date7.
                                 se=stmtdate
                                 dbc=mmddyy6.;
    item=filler1                 lv=2  dbf=$2.;
    item=check_balance           lv=2  dbf=pd5.2
                                 se=stmtbal
                                 dbc=l;

  record='checking_debit_record' sg=chckdebt sl=80;
    item=check_debit_amount  lv=2  dbf=pd5.2
                                 key=y
                                 se=debtamt
                                 dbc=l;
    item=check_debit_date        lv=2  dbf=6.0
                                 fmt=date7.
                                 se=debtdatd
                                 dbc=mmddyy6.;
    item=filler2                 lv=2  dbf=$2.;
    item=check_debit_time        lv=2  dbf=$8.
                                 se=debttime;
    item=check_debit_desc        lv=2  dbf=$59.
                                 se=debtdesc;

  record='checking_credit_record' sg=chckcrdt sl=80;
    item=check_credit_amount  lv=2  dbf=pd5.2
                                 key=y
                                 se=crdtamt
                                 dbc=l;
    item=check_credit_date       lv=2  dbf=6.0
                                 fmt=date7.
                                 se=crdtdate
                                 dbc=mmddyy6.;
    item=filler3                 lv=2  dbf=$2.;
    item=check_credit_time       lv=2  dbf=$8.
                                 se=crdttime;
    item=check_credit_desc       lv=2  dbf=$59.
                                 se=crdtdesc;

  record='savings_account_record' sg=saveacct sl=40;
    item=savings_account_number lv=2 dbf=12.
                                 key=y
                                 se=acnumber;
    item=savings_amount          lv=2  dbf=pd5.2
                                 se=stmtamt
                                 dbc=l;
    item=savings_date            lv=2  dbf=6.0
                                 fmt=date7.
                                 se=stmtdate
                                 dbc=mmddyy6.;
    item=filler4                 lv=2  dbf=$2.;
    item=savings_balance         lv=2  dbf=pd5.2
                                 se=stmtbal
                                 dbc=l;
```

```
            record='savings_debit_record' sg=savedebt sl=80;
              item=savings_debit_amount lv=2  dbf=pd5.2
                                        key=y
                                        se=debtamt
                                        dbc=l;
              item=savings_debit_date   lv=2  dbf=6.0
                                        fmt=date7.
                                        se=debtdate
                                        dbc=mmddyy6.;
              item=filler5              lv=2  dbf=$2.;
              item=savings_debit_time   lv=2  dbf=$8.
                                        se=debttime;
              item=savings_debit_desc   lv=2  dbf=$59.
                                        se=debtdesc;

            record='savings_credit_record' sg=savecrdt sl=80;
              item=savings_credit_amount lv=2 dbf=pd5.2
                                         key=y
                                         se=crdtamt
                                         dbc=l;
              item=savings_credit_date   lv=2  dbf=6.0
                                         fmt=date7.
                                         se=crdtdate
                                         dbc=mmddyy6.;
              item=filler6               lv=2  dbf=$2.;
              item=savings_credit_time   lv=2  dbf=$8.
                                         se=crdttime;
              item=savings_credit_desc   lv=2  dbf=$59.
                                         se=crdtdesc;
         list all;
       run;
```

EMPLINF2 Database Access Descriptor

This section describes the MYLIB.EMPLOYEE access descriptor for the EMPLINF2
database that is used in the examples and provides the statements that are used to
create the EMPLOYEE access descriptor in batch, interactive line, or noninteractive
mode.

```
proc access dbms=ims;
  create mylib.employee.access;
    database=emplinf2 dbtype=hidam;
    record='employee record' segment=employee
                          seglng=150;
      item=employee_id      lv=2  dbf=pd3.0
                            key=u
                            se=empid;
      item=last_name        lv=2  dbf=$10.
                            se=lastname;
      item=first_name       lv=2  dbf=$20.
                            se=frstname;
      item=hire_date        lv=2  dbf=6.0
```

```
                                fmt=date7.
                                se=hiredate
                                dbc=mmddyy6.;
        item=birthday          lv=2  dbf=7.0
                                fmt=date7.
                                se=birthday
                                dbc=mmddyy6.;
        item=ssn               lv=2  dbf=$11.
                                se=ssn;
        item=gender            lv=2  dbf=$6.
                                se=gender;
        item=status            lv=2  dbf=$9.
                                se=status;
        item=phone_extension   lv=2  dbf=$9.
                                se=phone;
        item=vacation          lv=2  dbf=ib4.
                                se=vacation
                                dbc=l;
        item=department        lv=2  dbf=zd6.0
                                se=deptment;
        item=zip_code          lv=2  dbf=$5.
                                se=zipcode;
        item=city_and_state    lv=2  dbf=$15.
                                se=citystat;
        item=street            lv=2  dbf=$20.
                                se=street;
        item=security          lv=2  dbf=rb4.
                                fmt=10.0
                                se=security
                                dbc=l;
        item=sick_leave        lv=2  dbf=6.2
                                se=sicklv
                                dbc=l;
     list all;
```

WIRETRAN Database Access Descriptor

This section describes the MYLIB.WIRETRAN access descriptor for the WIRETRAN database that is used in examples and provides the statements that are used to create the WIRETRAN access descriptor in batch, interactive line, or noninteractive mode. The WIRETRAN database is described in detail in Chapter 3, "Defining SAS/ACCESS Descriptor Files," on page 41.

```
   proc access dbms=ims;
     create mylib.wiretrn.access;
       database=wiretrn dbtype=hdam;
       record='wire transaction' segment=wiretran
                                seglng=100;
          item='ssn - account'   lv=2  dbf=$23.
                                se=ssnacc
                                key=y;
          item='account type'    lv=2  dbf=$1.
                                se=accttype;
          item='wire date'       lv=2  dbf=$8.
```

```
                                     se=wiredate;
          item='wire time'          lv=2  dbf=$8.
                                     se=wiretime;
          item='wire amount'        lv=2  dbf=pd5.2
                                     se=wireammt
                                     dbc=l;
          item='wire description' lv=2  dbf=$40.
                                     se=wiredesc;
      an=y;
     list all;

   run;
```

View Descriptors Based on the Access Descriptors for IMS

ACCTDBD Database View Descriptors

This section shows the SAS statements that are used to create the view descriptors for the ACCTDBD database that is used in the examples in this document. The ACCTDBD database is described in Chapter 2, "IMS Essentials," on page 11. The view descriptors are presented here in alphabetical order for easy reference.

You can create all the view descriptors used in the document by using PROC ACCESS statements. These view descriptors are based on the MYLIB.ACCOUNT access descriptor shown earlier in this appendix.

```
proc access dbms=ims ad=mylib.account;
  create vlib.account.view psb=accupsb;
    select  soc_sec_number
            customer_name
            city
            state
            zip_code;
  list view;

  create vlib.cdbtdate.view psb=accupsb;
    select  check_account_number
            check_date;
  list view;

  create vlib.chckacct.view psb=accupsb;
    select  soc_sec_number
            customer_name
            check_account_number
            check_date
            check_balance;
  list view;

  create vlib.chkcrd.view psb=accupsb pcb=2;
    select  customer_record
```

```
                 checking_account_record
                 checking_credit_record;
       reset 17 28;
   list view;

   create vlib.chkdeb.view psb=accupsb pcb=3;
     select   customer_record
                 checking_account_record
                 checking_debit_record;
       reset 17 22;
   list view;

   create vlib.chktrans.view psb=accupsb;
     select   customer_name
                 check_account_number
                 check_date
                 check_balance;
   list view;

   create vlib.credits.view psb=accupsb;
     select   soc_sec_number
                 check_account_number
                 check_credit_amount
                 check_credit_date
                 check_credit_time
                 check_credit_desc;
   list view;

   create vlib.custacct.view psb=accupsb;
     select   soc_sec_number
                 customer_name
                 check_account number;
   list view;

   create vlib.custinfo.view psb=accupsb;
     select   2 3 5 6 7 8 9 10 11 12;
   list view;

   create vlib.custphon.view psb=accupsb;
     select   soc_sec_number
                 customer_name
                 home_phone
                 office_phone;
   list view;

   create vlib.savebal.view psb=accupsb;
     select   soc_sec_number
                 customer_name
                 city
                 32 36;
   list view;

   create vlib.ssname.view psb=accupsb;
     select   soc_sec_number
```

```
            customer_name;
    list view;

    create vlib.trans.view psb=accupsb;
      select  soc_sec_number
              check_account_number
              check_debit_amount;
    list view;

  run;
```

EMPLINF2 Database View Descriptors

This section shows the SAS statements that are used to create the view descriptors for the EMPLINF2 database used in the examples in this document. The view descriptors are presented here in alphabetical order for easy reference. You can create all the view descriptors used in the document by using PROC ACCESS statements. These view descriptors are based on the MYLIB.EMPLOYEE access descriptor shown earlier in this appendix.

```
proc access dbms=ims accdesc=mylib.employee;
  create vlib.emplload.view psbname=empilpsb;
    select  employee_record;
  list view;

  create vlib.emplview.view psbname=empiupsb;
    select  employee_record;
  list view;

  create vlib.empbday.view psbname=empiupsb;
    select  employee_id
            last_name
            first_name
            birthday
            phone_extension;
  list view;

  run;
```

WIRETRAN Database View Descriptor

This section shows the SAS statements that are used to create the VLIB.WIREDATA view descriptor for the WIRETRAN database that is used in the examples in this document. The WIRETRAN database is described in detail in Chapter 3, "Defining SAS/ACCESS Descriptor Files," on page 41. This view descriptor is based on the MYLIB.WIRETRAN access descriptor shown earlier in this appendix.

```
proc access dbms=ims ad=mylib.wiretran;
  create vlib.wiredata.view psbname=acctsam
     pcbindex=5;
    select 'wire transaction';
  list view;
run;
```

Creating SAS Data Sets for IMS

This section shows the SAS statements that create each data file used in this document's examples.

MYDATA.BIRTHDAY Data Set

The SAS data set MYDATA.BIRTHDAY is updated with data from the EMPLINF2 database.

```
data mydata.birthday;
  input @01 employee_id   6.
        @08 last_name      $10.
        @19 birthday       date7.;
  format employee_id   6.
         last_name      $10.
         birthday       date7.;
  datalines;
  1247 Garcia      04APR54
  1078 Gibson      23APR36
  1005 Knapp       06OCT38
  1024 Mueller     17JUN53
;

proc print data=mydata.birthday;
  title2 'SAS Data Set MYDATA.BIRTHDAY';
run;
```

MYDATA.CHECKS Data Set

The SAS data set MYDATA.CHECKS is used to update the ACCTDBD database.

```
data mydata.checks;
  length customer_name  $40.;
  input  customer_name  & $
         soc_sec_number  $11.
         check_account_number
         check_balance
         check_date  date7.;
  format check_account_number  12.
         check_balance  12.2
         check_date  date7.;
  datalines;
COWPER, KEITH  241-98-4542 183352795865
862.31 25MAR95
OLSZEWSKI, STUART  309-22-4573 382654397566
486.00 02APR95
NAPOLITANO, BARBARA  250-36-8831 284522378774
104.20 10APR95
MCCALL, ROBERT  367-34-1543 644721295973
571.92 05APR95
;
```

```
proc print data=mydata.checks;
  title2 'SAS Data Set MYDATA.CHECKS';
run;
```

MYDATA.CHGDATA Data Set

The SAS data set MYDATA.CHGDATA is used to update the ACCTDBD database.

```
data mydata.chgdata;
  input account  12.
        charge;
  format account 14.
         charge  dollar7.;
  datalines;
345620135872 10
345620134522 7
345620123456 12
382957492811 3
345620134663 8
345620131455 6
345620104732 9
;

proc print data=mydata.chgdata;
  title2 'SAS Data Set MYDATA.CHGDATA';
```

MYDATA.CHKCRED Data Set

The SAS data set MYDATA.CHKCRED is used to add the checking credit path to the
ACCTDBD database.

```
data mydata.chkcred;
      /****  CUSTOMER data    ***/
  input @1   soc_sec_number    $11.
        @13  customer_name     $40.  /
        @1   addr_line_1       $30.
        @32  addr_line_2       $30.  /
        @1   city              $28.
        @30  state             $2.
        @33  country           $20.
        @54  zip_code          $10.  /
        @1   home_phone        $12.
        @14  office_phone      $12.
      /****  CHCKACCT data     ***/
        @27  check_account_number 12.0
        @40  check_amount         12.2
        @53  check_date           date7. /
        @1   filler1           $2.
        @4   check_balance        12.2
      /****  CHCKCRDT data     ***/
        @17  check_credit_amount  12.2
        @30  check_credit_date    date7.
        @38  filler3           $2.
        @41  check_credit_time    $8.    /
```

```
        @1   check_credit_desc     $59.;
    format check_credit_date     date7.;
  datalines;
667-73-8275 WALLS, HOOPER J.


            345620145345
        1563.23 31MAR95    15:42:43
MAIN ST BRANCH DEPOSIT
667-73-8275 WALLS, HOOPER J.


            345620154633
        1563.23 31MAR95    15:42:43
BAD ACCT_NUM
434-62-1234 SUMMERS, MARY T.


            345620104732
        400.00 02APR95    10:23:46
ACH DEPOSIT
436-42-6394 BOOKER, APRIL M.


            345620135872
         50.00 02APR95    12:16:34
ACH DEPOSIT
434-62-1224 SMITH, JAMES MARTIN


            345620134564
        1342.42 22MAR95    23:23:52
ACH DEPOSIT
434-62-1224 SMITH, JAMES MARTIN


            345620134663
        120.00 28MAR95    10:26:45
ACH DEPOSIT
178-42-6534 PATTILLO, RODRIGUES


            745920057114
       1300.00 12JUN95    14:34:12
ACH DEPOSIT
156-45-5672 O'CONNOR, JOSEPH


            345620123456
        100.00 01APR95    12:24:34
ATM DEPOSIT
657-34-3245 BARNHARDT, PAMELA S.
```

```
          345620131455
     230.00 04APR95     14:24:11
ACH DEPOSIT
667-82-8275 COHEN, ABRAHAM


          382957492811
    100.00 16APR95     09:21:14
ACH DEPOSIT
456-45-3462 LITTLE, NANCY M.


          345620134522
   50.00 05APR95     12:14:52
ACH DEPOSIT
234-74-4612 WIKOWSKI, JONATHAN S.


          345620113263
         672.32 31MAR95
ATM DEPOSIT
;
```

MYDATA.CHKDEBD Data Set

The SAS data set MYDATA.CHKDEBD is used to add the checking debit path to the ACCTDBD database.

```
data mydata.chkdebd;
      /****  CUSTOMER data     ***/
  input @1    soc_sec_number     $11.
        @13   customer_name      $40.  /
        @1    addr_line_1        $30.
        @32   addr_line_2        $30.  /
        @1    city               $28.
        @30   state              $2.
        @33   country            $20.
        @54   zip_code           $10.  /
        @1    home_phone         $12.
        @14   office_phone       $12.
        /****  CHCKACCT data     ***/
        @27   check_account_number  12.0
        @40   check_amount        12.2
        @53   check_date          date7. /
        @1    filler1             $2.
        @4    check_balance       12.2
        /****  CHCKDEBT data     ***/
        @17   check_debit_amount  12.2
        @30   check_debit_date    date7.
        @38   filler2             $2.
        @41   check_debit_time    $8.     /
        @1    check_debit_desc    $59.;
      format check_date     date7.;
      format check_debit_date   date7.;
```

```
   datalines;
667-73-8275 WALLS, HOOPER J.


             345620145345
   1266.34      820.00 23MAR95    23:54:53
CHECK 2958
667-73-8275 WALLS, HOOPER J.


             345620145345
   1266.34       52.00 23MAR95    23:54:53
CHECK 2948
667-73-8275 WALLS, HOOPER J.


             345620145345
   1266.34      193.00 28MAR95    22:51:43
CHECK 2951
667-73-8275 WALLS, HOOPER J.


             345620154633  1303.41 28MAR95
   1298.04          .    .


434-62-1234 SUMMERS, MARY T.
                  4322 LEON ST.
GORDONSVILLE      VA USA        26001-0670
803-657-1687    345620104732  826.05 27MAR95
    825.45          .    .


436-42-6394 BOOKER, APRIL M.
                  9712 WALLINGFORD PL.
GORDONSVILLE      VA USA        26001-0670
803-657-1346    345620135872   220.11 26MAR95
    234.89          .   30MAR94    22:34:45
CHECK 103
434-62-1224 SMITH, JAMES MARTIN
                  133 TOWNSEND ST.
GORDONSVILLE      VA USA        26001-0670
803-657-3437    345620134564  2392.93 16MAR95
   2645.34      432.87 18MAR95    22:13:48
CHECK 1826
434-62-1224 SMITH, JAMES MARTIN


             345620134564
   2645.34       19.23 18MAR95    22:13:48
CHECK 1821
434-62-1224 SMITH, JAMES MARTIN


             345620134564
   2645.34      723.23 22MAR95    21:48:12
CHECK 1828
```

```
434-62-1224 SMITH, JAMES MARTIN


                 345620134564
   2645.34      82.32 22MAR95    21:48:12
CHECK 1829
434-62-1224 SMITH, JAMES MARTIN


                 345620134564
   2645.34      73.62 26MAR95    21:22:24
CHECK 1830
434-62-1224 SMITH, JAMES MARTIN


                 345620134564
   2645.34      31.23 26MAR95    21:22:24
CHECK 1831
434-62-1224 SMITH, JAMES MARTIN


                 345620134564
    2645.34     162.87 29MAR94    22:51:12
CHECK 1835
434-62-1224 SMITH, JAMES MARTIN


                 345620134564
   2645.34       7.12 29MAR95    22:51:12
CHECK 1836
434-62-1224 SMITH, JAMES MARTIN


                 345620134564
    2645.34      62.34 31MAR95    23:02:12
CHECK 1833
434-62-1224 SMITH, JAMES MARTIN


                 345620134663     0.00 24MAR95
      143.78       25.00 28MAR95    15:53:29
ATM MAIN ST.
178-42-6534 PATTILLO, RODRIGUES
                           9712 COOK RD.
ORANGE        VA USA           26042-1650
803-657-1346 803-657-1345 745920057114 1404.90
10JUN95     1502.78   25.89 10JUN95   11:45:25
CHECK 412
156-45-5672 O'CONNOR, JOSEPH
                           235 MAIN ST.
ORANGE        VA USA           26042-1650
803-657-5656 803-623-4257 345620123456  353.65
27MAR95     463.23     13.29 28MAR95   22:23:53
CHECK 934
```

```
156-45-5672 O'CONNOR, JOESPH



803-657-5656 803-623-4257 345620123456
          463.23    32.87 31MAR95    23:35:53
CHECK 931
156-45-5672 O'CONNOR, JOSEPH



             345620123456
    463.23      50.00 02APR95    10:23:41
ATM GREEN ST
156-45-5672 O'CONNOR, JOESPH



             345620123456
    463.23      13.42 31MAR95    23:35:53
CHECK 935
657-34-3245 BARNHARDT, PAMELA S.
                       RT 2 BOX 324
CHARLOTTESVILLE    VA USA      25804-0997
803-345-4346 803-355-2543 345620131455
1243.25  29MAR95       1243.25        .   .

667-82-8275 COHEN, ABRAHAM
                       2345 DUKE ST.
CHARLOTTESVILLE    VA USA      25804-0997
803-657-7435 803-645-4234 382957492811 7462.51
03APR95       7302.06        .   .

456-45-3462 LITTLE, NANCY M.
                       4543 ELGIN AVE.
RICHMOND         VA USA       26502-3317
803-657-3566    345620134522    608.24 25MAR95
     831.65      42.73 29MAR95    23:12:34
CHECK 296
456-45-3462 LITTLE, NANCY M.



             345620134522
    831.65     172.45 29MAR95    23:12:34
CHECK 301
456-45-3462 LITTLE, NANCY M.



             345620134522
    831.65      38.23 30MAR95    22:51:34
CHECK 297
456-45-3462 LITTLE, NANCY M.



             345620134522
    831.65      10.00 02APR95    21:51:34
CHECK 298
```

```
234-74-4612 WIKOWSKI, JONATHAN S.
                         4356 CAMPUS DRIVE
RICHMOND        VA USA          26502-5317
803-467-4587 803-654-7238 345620113263 672.32
28MAR95        13.28        .   .


;
```

MYDATA.EMPLDATA Data Set

The SAS data set MYDATA.EMPLDATA is used to load the EMPLINF2 database.

```
data mydata.empldata;
  input @01 employee_id     6.
        @08 last_name       $10.
        @19 first_name      $20.
        @40 hire_date       yymmdd6.
        @47 birthday        yymmdd6.
        @54 ssn             $11. /
        @01 gender          $6.
        @08 status          $9.
        @18 phone_extension $9.
        @28 vacation        8.2
        @37 department      8.
        @46 zip_code        $5.
        @52 city_and_state  $15. /
        @01 street          $20.
        @21 security        5.
        @27 sick_leave      8.2;
  format hire_date   yymmdd6.
         birthday    yymmdd6.;
  datalines;
  1001 Waterhouse Clifton P.781231 480101 254-43-6089
Male  Full Time X5109  8.00   200 78752 Austin,TX
505 Cat Mountain Tr.  310      8.00
  1002 Bowman     Hugh E.    801230 310714 329-88-6729
Male  Full Time X5901  40.00 1000 78741 Austin,TX
47 Cypress Point Cir  310     80.00
  1003 Salazar    Yolanda    821230 401212 166-88-7516
Female Full Time X5169  80.00 200 78641 Leander,TX
6811 Picket Fence Dr  310     56.00
  1004 Knight     Althea     841229 500409 942-62-3354
Female Full Time X5218 300 78664 Round Rock,TX
8222 Whitewing Way    110     16.00
  1005 Knapp      Patrice R. 811230 371004 353-43-1272
Female Full Time X5012 8.00   100 78748 Austin,TX
19 Pack Saddle Pass   110     44.00
  1006 Garrett    Olan M.    781231 350123 776-94-3545
Male  Full Time X5208 80.00   300 78731 Austin,TX
67 Running Doe Ln.    110     60.00
  1007 Brown      Virgina P.801230 460524 675-29-9081
Female Full Time X5258 48.00  300 78610 Buda,TX
2713 Nutty Brown Mil  110     32.00
  1008 Hernandez Jesse L.    821230 330326 123-12-0987
```

```
Male  Full Time X5448 56.00  500 78664 Round Rock,TX
4319 Red Stone Lane     .     8.00
  1009 Jones      Michael Y. 850330 310521 543-87-1934
Male  Full Time X5713 80.00  800 78748 Austin,TX
23 Moonlight Bend La    .    80.00
  1010 Smith      Janet F.   811230 470807 105-32-9011
Female Full Time X5621 16.00  700 78737 Austin,TX
523 Rim Rock Road       .     8.00
  1011 Van Hotten Gwendolyn 790201 420913 766-30-9237
Female Full Time X5311  .     400 78641 Leander,TX
623 Fauntleroy Trail    .    32.00
  1012 Quintero   Pedro      810214 480221 339-94-2674
Male  Full Time X5348 32.00  400 78741 Austin,TX
77 Button Quail Cove    .    40.00
  1015 Scholl     Madison A. 830304 450319 765-43-0581
Male  Full Time X5419 40.00  500 78741 Austin,TX
3910 Covered Wagon      .    80.00
  1017 Waggonner Merrilee D.850330 360427 586-54-8967
Female Full Time X5914 56.00 1000 78722 Austin,TX
941 Bridgewater Dr.     .    40.00
  1020 Rudd       Fred       601230     . 145-67-6532
Male  Part Time         .    100
                        .     .
  1024 Mueller    Patsy      790403 520617 857-51-1838
Female Full Time X5822 40.00 900 78620 Dripping Spring
6935 Cherry Creek Rd  110    40.00
  1031 Chan       Tai        810502 460704 843-09-7123
Male  Full Time X5331 40.00  400 78755 Austin,TX
1412 Arapahoe Trail     .    80.00
  1049 Fernandez  Sophia     830516 440911 764-91-0193
Female Full Time X5847 96.00  900 78744 Austin,TX
4700 Old Stage Trail    .    40.00
  1050 Ameer      David      850530 511010 456-34-6543
Male  Full Time X5495 56.00  500 78735 Austin,TX
231 Little Hill Cir.    .     .
  1062 Littlejohn Fannie     850429 540517 978-63-3930
Female Full Time X5653 8.00  700 78660 Pflugerville,TX
813 Lime Rock Dr.     110    48.00
  1067 Cahill     Jacob      790105 401225 102-78-8765
Male  Full Time X5042 60.00  100 78748 Austin,TX
121 Hidden Hollow       .    36.00
  1071 Canady     Frank A.   810331 411119 345-91-4321
Male  Full Time X5406  8.00  500 78756 Austin,TX
741 Canyonwood Lane     .     8.00
  1074 Millsap    Joel B.    830831 360612 675-23-8027
Male  Full Time X5224 24.00  300 78755 Austin,TX
1201 Broken Bow Pass  110    48.00
  1077 Gibson     Teddy B.   850929 460423 567-89-2345
Male            X5703 80.00  800 78753 Austin,TX
4441 Hansford           .    80.00
  1078 Gibson     George J.  820930 460423 567-89-2346
Male  Full Time X5703 80.00  800 78753 Austin,TX
2311 Hansford           .    80.00
  1083 Savage     William D. 791001 530120 211-95-9608
```

```
Male  Full Time X5505 80.00  600 78737 Austin,TX
97 Cimarron Circle      .    48.00
  1086 Schmidt    Penny      811017 270219 901-45-4567
Female Full Time X5822 80.00  900 78735 Austin,TX
6419 Wild Rose Road     .    80.00
  1092 Polanski   Ivan L.    831130 471011 497-36-7845
Male  Full Time X5621 56.00 700 78620 Dripping Spring
2501 Timberline Tr.     .     .
  1101 Nathaniel  Darryl     860101 440321 584-86-6945
Male  Full Time X5544 40.00  600 78735 Austin,TX
1892 Red River Road    210    8.00
  1105 Faulkner   Carrie Ann 830102 510817 987-76-7469
Female Full Time X5417 48.00  500 78756 Austin,TX
5649 Foothill Park     110   16.00
  1112 Jones      Rita M.    790202 481224 890-98-6789
Female Full Time X5271 24.00  300 78735 Austin,TX
907 Hickory Stick      .     8.00
  1119 Goodson    Alan F.    820116 500621 234-67-8901
Male  Full Time X5512 48.00  600 78626 Georgetown,TX
11410 Smokey Hill Rd    .    16.00
  1120 Reid       David G.   830214 450815 442-04-0121
Male  Full Time X5369 80.00  400 78752 Austin,TX
1322 Lazy Lane         224   80.00
  1123 Freeman    Leopold    861030 350209 828-26-7282
Male  Part Time X5604   .    700 78757 Austin,TX
13 Timber Hills Tr.    106     .
  1133 Williamson Janice L. 831103 520519 131-41-9129
Female Full Time X5802 40.00  900 78610 Buda,TX
2706 Frontier Valley   .     8.00
  1139 Seaton     Gary      800403 561003 286-04-6279
Male  Full Time X5545 80.00  600 78757 Austin,TX
2111 Wind Ridge Road   .     80.00
  1145 Juarez     Armando   820501 470528 876-19-0378
Male  Full Time X5987 48.00 1000 78626 Georgetown,TX
1017 Woodstone Sq.     .     16.00
  1156 Reed       Kenneth D. 840830 550105 875-15-1388
Male  Full Time X5307 64.00  400 78641 Leander,TX
1349 Begonia Terrace   .     40.00
  1161 Richardson Travis Z. 860913 371130 654-54-8127
Male  Full Time X5325 88.00  400 78752 Austin,TX
2009 Mountain Lake     110   96.00
  1213 Johnson    Bradford   840131 540415 321-32-9446
Male  Full Time X5446 40.00  500 78724 Austin,TX
678 Buffalo Gap Road   .     40.00
  1217 Rodriguez Romualdo R. 810131 290209 493-77-4863
Male  Full Time X5874 32.00  900 78746 Austin,TX
804 Lazy Brook Lane    .     48.00
  1219 Kaatz      Freddie    830131 570621 181-49-4592
Male  Full Time X5387 80.00  400 78753 Austin,TX
4713 Cedar Tree Lane   .     80.00
  1234 Shropshire Leland G. 850415 490904 555-21-4173
Male  Full Time X5616 32.00  700 78752 Austin,TX
606 Bull Creek Trail   .     40.00
  1238 Throckmort Stewart Q.850516 310804 109-07-5098
```

```
Male  Full Time X5391 40.00  400 78756 Austin,TX
479 Roundup Circle      .    40.00
   1247 Garcia     Francisco 840730 550505 678-23-0123
Male  Full Time X5348 80.00  400 78756 Austin,TX
479 Whispering Wind     .    72.00
   1261 Collins    Lillian   810824 510501 302-59-2781
Female Full Time X5616 80.00  700 78664 Round Rock,TX
9117 Beaver Creek Rd    .    48.00
   1265 Slye      Leonard R. 840331 601218 434-21-1300
Male  Half Time X5123   .    200 78742 Austin,TX
4106 Main St.           .      .
   1266 Redfox    Richard B. 850902 440404 210-65-2786
Male  Full Time X5386 48.00  400 78660 Pflugerville,TX
9807 Three Oaks Tr.     .    48.00
   1272 Smith     Garland P. 850413 540405 397-80-8491
Male  Full Time X5415  8.00  500 78602 Bastrop,TX
7594 Red Cliff Rd.      .    48.00
   1313 Smith     Jerry Lee  850130 420913 823-10-0951
Male  Full Time X5169   .    200 78745 Austin,TX
8203 Friar Tuck Ln.     .    16.00
   1327 Brooks    Ruben R.   820430 520225 789-56-2109
Male  Full Time X5347 80.00  400 78744 Austin,TX
2509 Loganberry Dr.     .    80.00
   1900 Smith     John       .     .      .
                             .    100
                             .     .
 ;
```

MYDATA.INITSEG Data Set

The SAS data set MYDATA.INITSEG is used to initially load the ACCTDBD
database.

```
data mydata.initseg;
     /**** CUSTOMER data   ***/
  input @1   soc_sec_number    $11.
        @13  customer_name     $40.  /
        @1   addr_line_1       $30.
        @32  addr_line_2       $30.  /
        @1   city              $28.
        @30  state             $2.
        @33  country           $20.
        @54  zip_code          $10.  /
        @1   home_phone        $12.
        @14  office_phone      $12.
     /**** CHCKACCT data   ***/
        @27  check_account_number  12.0
        @40  check_amount          12.2
        @53  check_date            date7. /
        @1   filler1           $2.
        @4   check_balance         12.2
     /**** CHCKDEBT data   ***/
        @17  check_debit_amount    12.2
        @30  check_debit_date      date7.
```

```
        @38  filler2                $2.
        @41  check_debit_time       $8.    /
        @1   check_debit_desc       $59.;
     format check_date  date7.;
     format check_debit_date  date7.;
  datalines;
667-73-8275 WALLS, HOOPER J.
                    4525 CLARENDON RD
RAPIDAN         VA USA              22215-5600
803-657-3098 803-645-4418 345620145345 1702.19 15MAR95
      1266.34          .   19MAR94    21:22:53
CHECK 2947
;
```

MYDATA.PHONENUM Data Set

The SAS data set MYDATA.PHONENUM is used to update the ACCTDBD database.

```
data mydata.phonenum;
  soc_sec_number = '667-73-8275';
  home_phone = '703-657-3098';
  office_phone = '703-645-4418';
  output;
  soc_sec_number = '434-62-1234';
  home_phone = '703-645-441 ';
  office_phone = '            ';
  output;
  soc_sec_number = '178-42-6534';
  home_phone = '703-657-1346';
  office_phone = '703-657-1345';
  output;
  soc_sec_number = '156-45-5672';
  home_phone = '703-657-5656';
  office_phone = '703-623-4257';
  output;
  soc_sec_number = '657-34-3245';
  home_phone = '703-345-4346';
  office_phone = '703-355-5438';
  output;
  soc_sec_number = '456-45-3462';
  home_phone = '703-657-3566';
  office_phone = '703-645-1212';
  output;
  soc_sec_number = '416-41-3162';
  home_phone = '703-657-3166';
  office_phone = '703-615-1212';
  output;
run;

proc print data=mydata.phonenum;
  title2 'SAS Data Set MYDATA.PHONENUM';
run;
```

MYDATA.SAVCRED Data Set

The SAS data set MYDATA.SAVCRED is used to add the savings credit path to the ACCTDBD database.

```
data mydata.savcred;
        /****  CUSTOMER data    ***/
   input @1   soc_sec_number      $11.
         @13  customer_name       $40.  /
         @1   addr_line_1         $30.
         @32  addr_line_2         $30.  /
         @1   city                $28.
         @30  state               $2.
         @33  country             $20.
         @54  zip_code            $10.  /
         @1   home_phone          $12.
         @14  office_phone        $12.
         /****  SAVEACCT data    ***/
         @27  savings_account_number 12.0
         @40  savings_amount         12.2
         @53  savings_date           date7. /
         @1   filler4                $2.
         @4   savings_balance        12.2
        /****  SAVECRDT data    ***/
         @17  savings_credit_amount  12.2
         @30  savings_credit_date    date7.
         @38  filler6                $2.
         @41  savings_credit_time    $8.   /
         @1   savings_credit_desc    $59.;
     format savings_credit_date  date7.;
   datalines;
667-73-8275 WALLS, HOOPER J.


            459923888253      784.29  28MAR95
   672.63        8.45 30MAR95    09:34:18
INTEREST
434-62-1234 SUMMERS, MARY T.
                             4322 LEON ST.
GORDONSVILLE       VA USA         26001-0670
            345689404732      8406.0  27MAR95
 8364.24       41.82 30MAR95    23:46:03
INTEREST
436-42-6394 BOOKER, APRIL M.
                             9712 WALLINGFORD PL.
GORDONSVILLE       VA USA         26001-0670
            144256844728      809.45 21MAR95
 1032.23       50.00 26MAR95    12:26:15
INTEREST
434-62-1224 SMITH, JAMES MARTIN
                             133 TOWNSEND ST.
GORDONSVILLE       VA USA         26001-0670
            345689473762      130.64 15MAR95
   261.64        1.31 30MAR95    23:45:53
```

```
INTEREST
434-62-1224 SMITH, JAMES MARTIN
                                  133 TOWNSEND ST.
GORDONSVILLE       VA USA          26001-0670
          345689498217        9421.79 16MAR95
 9374.92         46.07 30MAR95    23:45:32
INTEREST
178-42-6534 PATTILLO, RODRIGUES
                                  9712 COOK RD.
ORANGE             VA USA          26042-1650
          345689462413        950.96 15MAR95
  946.23          4.73 30MAR95    23:44:25
INTEREST
156-45-5672 O'CONNOR, JOESPH
                                  235 MAIN ST.
ORANGE             VA USA          26042-1650
          345689435776        136.40 27MAR95
  284.97          1.43 30MAR95    23:48:56
INTEREST
657-34-3245 BARNHARDT, PAMELA S.
                                  RT 2 BOX 324
CHARLOTTESVILLE    VA USA          25804-0997
          859993641223        845.35 18MAR95
 2553.45         71.44 26MAR95    08:41:28
INTEREST
667-82-8275 COHEN, ABRAHAM
                                  2345 DUKE ST.
CHARLOTTESVILLE    VA USA          25804-0997
          884672297126        945.25 26MAR95
  793.25         52.33 28MAR95    11:45:26
INTEREST
456-45-3462 LITTLE, NANCY M.


          345689463822        929.24 25MAR95
  924.62          4.62 30MAR95    23:46:01
INTEREST
234-74-4612 WIKOWSKI, JONATHAN S.
                                4356 CAMPUS DRIVE
RICHMOND           VA USA          26502-3317
                      .                  .   .
                 .            .   .

;
```

MYDATA.SAVDEBD Data Set

The SAS data set MYDATA.SAVDEBD is used to add the savings debit path to the ACCTDBD database.

```
data mydata.savdebd;
      /****  CUSTOMER data    ***/
  input @1   soc_sec_number      $11.
        @13  customer_name       $40.  /
```

```
            @1    addr_line_1          $30.
            @32   addr_line_2          $30.  /
            @1    city                 $28.
            @30   state                $2.
            @33   country              $20.
            @54   zip_code             $10.  /
            @1    home_phone           $12.
            @14   office_phone         $12.
            /****  SAVEACCT data    ***/
            @27   savings_account_number 12.0
            @40   savings_amount       12.2
            @53   savings_date         date7. /
            @1    filler4              $2.
            @4    savings_balance      12.2
            /****  SAVEDEBT data    ***/
            @17   savings_debit_amount  12.2
            @30   savings_debit_date    date7.
            @38   filler5              $2.
            @41   savings_debit_time   $8.    /
            @1    savings_debit_desc   $59.;
      format savings_date  date7.;
      format savings_debit_date  date7.;
   datalines;
667-73-8275 WALLS, HOOPER J.


          459923888253      784.29 28MAR95
        672.63          .   .


434-62-1234 SUMMERS, MARY T.
                              4322 LEON ST.
GORDONSVILLE       VA USA       26001-0670
          345689404732      8406.00 27MAR95
        8364.24         .   .


436-42-6394 BOOKER, APRIL M.
                         9712 WALLINGFORD PL.
GORDONSVILLE       VA USA       26001-0670
          144256844728      809.45 21MAR95
        1032.23         .   .


434-62-1224 SMITH, JAMES MARTIN
                              133 TOWNSEND ST.
GORDONSVILLE       VA USA       26001-0670
          345689473762      130.64 15MAR95
        261.64     132.31 03APR94    14:42:43
MAIN ST BRANCH WITHDRAWAL
434-62-1224 SMITH, JAMES MARTIN
                              133 TOWNSEND ST.
GORDONSVILLE       VA USA       26001-0670
          345689498217      9421.79 16MAR95
        9374.92         .   .


178-42-6534 PATTILLO, RODRIGUES
```

```
                                  9712 COOK RD.
     ORANGE             VA USA        26042-1650
          345689462413       950.96 15MAR95
          946.23          .    .


156-45-5672 O'CONNOR, JOESPH
                                  235 MAIN ST.
     ORANGE             VA USA        26042-1650
          345689435776       136.40 27MAR95
          284.97     150.00 31MAR94    12:23:42
ATM GREEN ST
657-34-3245 BARNHARDT, PAMELA S.
                                  RT 2 BOX 324
CHARLOTTESVILLE     VA USA        25804-0997
          859993641223       845.35 18MAR95
          2553.45          .    .


667-82-8275 COHEN, ABRAHAM
                                  2345 DUKE ST.
CHARLOTTESVILLE     VA USA        25804-0997
          884672297126       945.25 26MAR95
          793.25          .    .


456-45-3462 LITTLE, NANCY M.



          345689463822       929.24 25MAR95
          924.62          .    .


234-74-4612 WIKOWSKI, JONATHAN S.
                                  4356 CAMPUS DRIVE
RICHMOND             VA USA        26502-3317
                        .                    .    .
                   .         .    .


     ;
```

MYDATA.CUSTOMER Data Set

The SAS data set MYDATA.CUSTOMER is used to update the ACCTDBD database.

```
data mydata.customer;
     /****  CUSTOMER data    ***/
  input @1   soc_sec_number    $11.
        @13  customer_name     $40. /
        @1   addr_line_1       $30.
        @32  addr_line_2       $30. /
        @1   city              $28.
        @30  state             $2.
        @33  country           $20.
        @54  zip_code          $10. /
        @1   home_phone        $12.
        @14  office_phone      $12.;
  datalines;
```

```
131-73-2785 HUTTLINGER, HORTENSE H.
                           2785 HILLARY PL
RAPIDAN        VA USA          22215-5600
803-657-4097 803-645-4419
232-62-2432 MANNERLY, MAYNARD M.
                           6525 MORGAN ST
RAPIDAN        VA USA          22215-5600
803-657-9066 803-645-4420
;
```

MYDATA.NEWADDR Data Set

The SAS data set MYDATA.NEWADDR is in Version 6 format and is used to update the ACCTDBD database.

```
data mydata.newaddr;
  /****  CUSTOMER data    ***/
  input @1   ssn          $11.
       /* social security number */
       @13  newaddr1    $30.
       /* first line of address */
       @44  newaddr2    $30. /
       /* second line of address */
       @1   newcity     $28.
       /* customer city */
       @30  newstate    $2.
       /* customer state */
       @33  newzip      $10.;
       /* customer zip code */
  datalines;
178-42-6534            1111 PAUL PLACE
RAPIDAN      VA 22215-5600
156-45-5672            2222 OSCAR DR.
ORANGE       VA 26042-1650
;
```

VER6.SSNUMS Data Set

The SAS data set VER6.SSNUMS is in Version 6 format and is used to update the ACCTDBD database.

```
data ver6.ssnums;
  input @1   ssnumb $11.
       @13 name   $40.;
  datalines;
267-83-2241 GORDIEVSKY, OLEG
276-44-6885 MIFUNE, YUKIO
352-44-2151 SHIEKELESLAM, SHALA
436-46-1931 NISHIMATSU-LYNCH, CAROL
;

proc print data=mydata.ssnums;
  title2 'SAS Data Set VER6.SSNUMS';
run;
```

SAS Statements for Loading DB2 Table BANKCHRG

The following statements are the executable SAS DATA step, PROC APPEND, and PROC ACCESS statements that are used to load DB2 table BANKCHRG.

Creating SAS Data Set MYDATA.BANK

The SAS data set MYDATA.BANK is used to load the DB2 table BANKCHRG.

Note: If you do not have DB2 at your site, change MYDATA.BANK to MYDATA.BANKCHRG and execute only the following program: △

```
data mydata.bank;
  input @1  ssn       $11.
        @13 accountn 12.
        @26 chckchrg 5.2
        @32 atmfee   5.2
        @38 loanchrg 6.2;
  format accountn 14.
         chckchrg 5.2
         atmfee   5.2
         loanchrg 6.2;
  datalines;
667-73-8275 345620145345 3.75  5.00  2.00
434-62-1234 345620104732 15.00 25.00 552.23
436-42-6394 345620135872 1.50  7.50  332.15
434-62-1224 345620134564 9.50  0.00  0.00
178-42-6534 .            0.50  15.00 223.77
156-45-5672 345620123456 0.00  0.00  0.00
657-34-3245 345620132455 10.25 10.00 100.00
667-82-8275 .            7.50  7.50  175.75
456-45-3462 345620134522 23.00 30.00 673.23
234-74-4612 345620113262 4.50  7.00  0.00
;

proc print data=mydata.bank;
  title2 'SAS Data Set MYDATA.BANK';
run;
```

Loading DB2 Table BANKCHRG from MYDATA.BANK

The following program loads DB2 table BANKCHRG from the SAS data set MYDATA.BANK. You must have DB2 installed at your site to run this program.

```
proc dbload dbms=db2 data=mydata.bank;
  accdesc=mylibdb2.bankchrg;
  table=<owner>.bankchrg;
  load;
run;
```

DB2 View Descriptor for BANKCHRG

The following program creates a DB2 view descriptor for the DB2 table BANKCHRG. You must have DB2 installed at your site to run this program.

```
proc access dbms=db2 ad=mylibdb2.bankchrg;
  create vlibdb2.bankchrg.view;
  select all;
  list view;
run;

proc print data=vlibdb2.bankchrg;
  title2 'DB2 Table BANKS.BANKCHRG';
run;
```
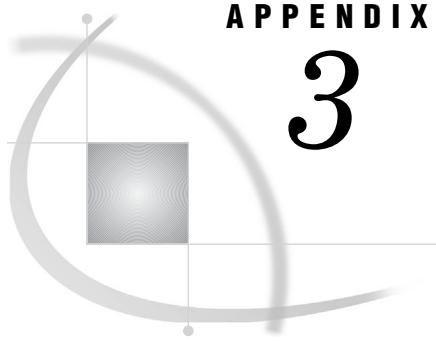
APPENDIX

3

Recommended Reading

Recommended Reading

Here is the recommended reading list for this title:

- □ *The Little SAS Book: A Primer*
- □ *SAS Language Reference: Concepts*
- □ *SAS Language Reference: Dictionary*
- □ *SAS Procedures Guide*
- □ SAS Companion that is specific to your operating system

For a complete list of SAS publications, see the current *SAS Publishing Catalog*. To order the most current publications or to receive a free copy of the catalog, contact a SAS representative at

SAS Publishing Sales
SAS Campus Drive
Cary, NC 27513
Telephone: (800) 727-3228*
Fax: (919) 677-8166
E-mail: sasbook@sas.com
Web address: **support.sas.com/pubs**
* For other SAS Institute business, call (919) 677-8000.

Customers outside the United States should contact their local SAS office.

Glossary

ACB
See Application Control Block (ACB).

ACBLIB
the data set that contains the DL/I Application Control Blocks. See also Application Control Block (ACB).

access descriptor
a SAS/ACCESS file that describes data that is managed by an external software vendor's DBMS product to SAS. After creating an access descriptor, you can use it as the basis for creating one or more view descriptors. See also view descriptor.

Application Control Block (ACB)
a DL/I control block that contains the combined information from the Database Descriptions (DBDs) and Program Specification Blocks (PSBs).

attach parameter list
a set of parameters that are passed to DL/I when the IMS engine or the IMS DATA step interface is executed in a DL/I environment. The parameters vary for each region type. Most parameters can be modified with SAS system options that are specific to the SAS/ACCESS interface to IMS.

batch mode
a method of executing SAS programs in which a file that contains SAS statements plus any necessary operating environment commands is submitted to the computer's batch queue. After you submit the program, control returns to your terminal or workstation, where you can perform other tasks. Batch mode is sometimes referred to as running in the background. The program output can be written to files or printed on an output device.

batch region
a DL/I processing environment for running batch mode jobs to access DL/I databases. Database data sets must be allocated to this region. A batch region is supervised by the DL/I batch control program.

BMP region
Batch Message Processing region. The BMP region is a DL/I processing environment in IMS/ESA DB/DC subsystems and in CICS for running batch programs that access active online DL/I databases and message queues, as well as non-DL/I data sets. Database data sets are allocated to an online control region, not to the BMP region.

browsing data
the process of viewing the contents of a file. You can view the data either one row at a time or in a tabular format. You cannot update data that you are browsing.

call (DL/I)
a request made by the IMS engine to DL/I to access one or more segments of a database or message queue, or to perform some system function.

checkpoint
the result of a CHKP call. A checkpoint establishes a synchronization point in the execution of the program. A synchronization point is used by DL/I backout services to determine which updates to back out (cancel) in the event of an abend or system crash. The program must be restarted at the synchronization point.

checkpoint ID
an eight-byte value that is written to the DL/I log record to identify the program checkpoint.

child
a segment in a DL/I database that is the direct dependent of another segment, which is called the child's parent. The data in a dependent segment rely on the parent segment and on all higher segments for complete identification and qualification.

command code
a special indicator that is used in a Segment Search Argument (SSA) to modify the type of call that is being issued. The most commonly used command code is the D code, which is used to issue a path call.

control block
a storage area that is created by the system and contains information for controlling system operations. The control block is often loaded from a control block library. For example, Database Descriptions (DBDs) can be stored in DBDLIBs.

DATA step
a group of statements in a SAS program that begins with a DATA statement and which ends with either a RUN statement, another DATA statement, a PROC statement, the end of the job, or the semicolon that immediately follows lines of data. The DATA step enables you to read raw data or other SAS data sets and to use programming logic to create a SAS data set, to write a report, or to write to an external file.

DATA step view
a type of SAS data set that consists of a stored DATA step program. Like other SAS data views, a DATA step view contains a definition of data that is stored elsewhere; the view does not contain the physical data. The view's input data can come from one or more sources, including external files and other SAS data sets. Because a DATA step view only reads (opens for input) other files, you cannot update this view's underlying data.

data type
an attribute of every column in a database. The data type tells the operating system how much physical storage to set aside for the column and specifies what type of data the item will contain.

data value
an element in a collection of data values that are organized and presented to SAS in a rectangular structure of columns and rows. A data value represents the intersection of a row and a column.

database

an organized collection of related data. In IMS, a database is a collection of interrelated data elements that are organized according to a particular data model that can be processed by multiple applications.

Database Administrator (DBA)

the person who is responsible for developing and maintaining database management systems at a computer site.

Database Description (DBD)

a DL/I control block that defines the hierarchical data structure and the physical characteristics of a database to DL/I.

database management system (DBMS)

a software application that enables you to create and manipulate data that is stored in the form of databases. See also relational database management system.

Database Recovery Control (DBRC)

an IMS facility that controls the restoration of databases after a system failure. DBRC also supports data sharing among IMS/ESA subsystems.

DBB region

a DL/I batch processing environment for running programs that can access DL/I databases as well as non-DL/I data sets. In a DBB region, DL/I accesses the ACBLIB in order to obtain control block information.

DBD

See Database Description (DBD).

DBDGEN

the utility procedure that generates a Database Description (DBD).

DBDLIB

a library that contains the Database Descriptions (DBDs).

DBRC

See Database Recovery Control (DBRC).

dependent segment

a segment that has a parent segment. The data in a dependent segment relies on the parent segment and on all higher segments for complete identification and qualification. See also segment.

DL/I

Data Language/I. DL/I is the IBM database language for IMS/ESA, CICS/OS/ESA, CICS/DOS/VS, and DL/I DOS/VS systems.

DLI region

a DL/I batch processing environment for running programs that can access DL/I databases as well as non-DL/I data sets. No access to message queues is possible. In a DLI region, DL/I accesses the DBDLIB and the PSBLIB in order to obtain control block information.

editing data

the process of viewing the contents of a file with the intent and ability to change those contents. You can view the data either one observation at a time or in a tabular format.

engine

a component of SAS that reads from or writes to a file. Each engine enables SAS to access files that are in a particular format. See also interface view engine.

feedback data
the data that is returned from IMS to the IMS engine (usually in the PCB mask) after a DL/I call has been issued.

field
the smallest unit of data storage in an IMS database.

file
a collection of related records that are treated as a unit. SAS files are processed and controlled by SAS and are stored in SAS data libraries.

format
an instruction SAS uses to display or write each value of a variable. Some formats are supplied by SAS software. You can create your own formats by using the FORMAT procedure in Base SAS software.

Get call
a DL/I call that retrieves one or more segments so that the contents of the segments can be processed or mapped by the IMS engine.

hierarchical database
a database that is organized as a tree structure of segments. A DL/I database has a hierarchical data structure.

hierarchical sequence
the standard processing sequence for segments of a database record. The sequence is basically top-to-bottom, front-to-back, and left-to-right. When segments in a DL/I database are retrieved sequentially, DL/I presents them in hierarchical sequence.

hierarchical structure
an arrangement of data in which records occur at distinct levels with different types of information at each level. Records are related to other records as ancestors, descendants, siblings, and so on.

IMS/ESA
Information Management System/Enterprise System Architecture. IMS/ESA is an IBM database management system that uses the DL/I database language.

IMS/ESA Resource Lock Manager (IRLM)
a facility for ensuring database integrity among multiple DL/I subsystems.

index
a component of a SAS data set that enables SAS to access observations in the SAS data set quickly and efficiently. The purpose of SAS indexes is to optimize WHERE-clause processing and to facilitate BY-group processing.

informat
an instruction that SAS uses to read raw data values to create variable values. Some informats are supplied by SAS software. Other informats can be written by the user with the FORMAT procedure in Base SAS software.

interactive line mode
a method of running SAS programs in which you enter one line of a SAS program at a time at the SAS session prompt. SAS processes each line immediately after you press the ENTER or RETURN key. Procedure output and informative messages are returned directly to your display device.

interface view engine
a SAS engine that retrieves data directly from files that have been formatted by another vendor's software vendors and presents the data to SAS in the form of a SAS data set. Interface view engines are transparent to user's and are not specified in LIBNAME statements.

I/O area
a data structure in the IMS engine holding retrieved segments for processing or output.

I/O PCB
an abbreviation for Input/Output Program Communication Block. The I/O PCB communicates information about non-database access requests.

IRLM
See IMS/ESA Resource Lock Manager (IRLM).

key field
a field that identifies and provides access to an occurrence of a segment. A key field is also called a sequence field. See also segment.

libref
a name that is temporarily associated with a SAS data library. For example, in the name SASUSER.ACCOUNTS, the name SASUSER is the libref. You assign a libref with a LIBNAME statement or with an operating system command.

logical database
a collection of database segments from one or more physical databases. A logical database enables the IMS engine to view a database structure that is different from the physical structure.

member
a SAS file in a SAS data library.

member name
a name that is assigned to a file in a SAS library. A member name can reference a SAS data set, a SAS catalog, a SAS access descriptor, or a stored program.

member type
a SAS name that identifies the type of information that is stored in a SAS file. Member types include ACCESS, DATA, CATALOG, PROGRAM, and VIEW.

missing value
a term that describes the contents of a variable that contains no data for a particular row or observation. By default, SAS prints or displays a missing numeric value as a single period, and it prints or displays a missing character value as a blank space.

noninteractive mode
a method of running SAS programs in which you prepare a file of SAS statements and submit the program to the operating system. The program runs immediately and occupies your current session.

observation
a row in a SAS data set. Each observation contains one data value for each variable in the data file. An observation is analogous to a record in an IMS database. Unlike rows in a DBMS table or file, observations in a SAS data file have an inherent order.

online access region
a DL/I processing environment for running batch programs that can access active online DL/I databases. The only type of online access region that the SAS/ACCESS interface to IMS supports is the BMP region.

online control region
a DL/I region that controls databases and terminals and which schedules activities, using these resources for online processing.

parent
a segment in a DL/I database that has one or more dependent segments, which are called its children. The data in a dependent segment rely on the parent segment and on all higher segments for complete identification and qualification.

path
a single route through a database, following the hierarchical sequence of segments from a higher-level segment to a lower-level segment.

path call
a DL/I call to a database that returns multiple segments from the hierarchical path.

PCB
See Program Communication Block (PCB).

PCB mask
a data structure to which DL/I returns information about the DL/I calls that an application issues.

physical database
a collection of database segments in a specified hierarchical structure. These segments are organized according to a particular DL/I access method.

PROC SQL view
a SAS data set (of type VIEW) that is created by the SQL procedure. A PROC SQL view contains no data. Instead, it stores information that enables it to read data values from other files, which can include SAS data files, SAS/ACCESS views, DATA step views, or other PROC SQL views. A PROC SQL view's output can be either a subset or a superset of one or more files.

PROC step
a group of SAS statements that call and execute a procedure. A PROC step usually takes a SAS data set as input.

Program Communication Block (PCB)
a DL/I control block that defines a message queue or the part of a database that can be accessed by the IMS engine. A PCB is part of a Program Specification Block (PSB).

Program Specification Block (PSB)
a DL/I control block that defines the DL/I resources used by the IMS engine. Each database used by the IMS engine is defined by a separate Program Communication Block (PCB) within the PSB.

program view
the part of a database that the IMS engine can access. The Program Communication Block (PCB) establishes the program view.

PSB
See Program Specification Block (PSB).

PSBGEN
the process that generates a Program Specification Block (PSB).

PSBLIB
a data set that contains the Program Specification Blocks (PSBs).

qualified call
a DL/I call that specifies at least one segment search argument (SSA).

qualified SSA
a segment search argument that contains one or more qualification statements. The qualification statements specify search criteria for locating occurrences of particular segments.

random access
the access mode used by the IMS engine or by the IMS DATA step interface when a WHERE statement is specified from which the engine can generate qualified Segment Search Arguments. In the SAS/ACCESS interface to IMS-DL/I, the distinction between sequential and random access differs from that of some other programming languages.

read integrity
the sharing of database access so that two programs cannot access a record simultaneously if one of the programs intends to update the record.

region type
the kind of DL/I processing environment. The IMS engine uses two categories of region types: batch regions (DLI or DBB) and online access regions (BMP).

relational database management system
a database management system that organizes and accesses data according to relationships between data items. The main characteristic of a relational database management system is the two-dimensional table.

restart
the process of resuming an interrupted program without repeating completed transactions.

restricted option
a SAS option that has been installed at your site so that its default setting cannot be overridden by the applications programmers.

return code
a code passed to the operating system that indicates whether a command or job step has executed successfully.

root segment
the highest-level segment in a database.

SAS data file
a SAS data set that contains data values as well as descriptor information that is associated with the data.

SAS data library
a collection of one or more SAS files that are stored and referenced as a unit. See also libref and member.

SAS data set
a collection of information that SAS software stores as a unit. Conceptually, a SAS data set is arranged in a rectangular, two-dimensional format. Each item in a SAS data set is called a data value. Data values in a row constitute an observation, and those in a column constitute a variable. See also SAS data file and SAS data view.

SAS data view
a type of SAS data set that retrieves data values from other files. A SAS data view contains only descriptor information such as the data types and lengths of the variables (columns), plus other information that is required for retrieving data values from other SAS data sets or from files that are stored in other software vendors' file formats.

SAS/ACCESS view
See SAS data view and view descriptor.

search field
a field that is defined to DL/I in the Database Description (DBD) and which can be used to search for particular segments. A search field does not provide unique identification of the segment.

segment
in a DL/I database, a grouping of related items of data in a database structure. The segment is the unit of data that can be accessed by the IMS engine or by the IMS DATA step interface.

segment level
the relative distance of a particular segment from the root segment along a hierarchical path. The segment level is usually represented numerically, with the root segment at level 1 and its immediate dependents at level 2.

segment occurrence
in a DL/I database, a specific instance in a set of segments that have the same segment type.

segment search argument (SSA)
the formatted search criteria that are passed to DL/I in order to identify a particular segment or group of segments to be processed. Multiple SSAs can be specified on one DL/I call.

segment type
in a DL/I database, a category of related data elements. There can be multiple segment occurrences for a particular segment type.

sensitive segment
a segment in a DL/I database that the IMS engine or the IMS DATA step interface can access. A segment is defined as *sensitive* for a particular program in the Program Specification Block (PSB).

sequence field
See key field.

sequential access
a method of file access in which the records are read or written one after the other from the beginning of the file to the end. Sequential access is the default access mode that the IMS engine uses to retrieve all segments down one path of a database. In the SAS/ACCESS interface to IMS, the distinction between sequential access and random access differs somewhat from that of other programming languages.

siblings
segments that have the same parent segment.

SSA
See segment search argument (SSA).

status code
a two-byte indicator field that DL/I returns to an application program to indicate the relative success of an attempted call.

Structured Query Language (SQL)
a standardized, high-level query language that is used in relational database management systems to create and manipulate database management system objects. SAS implements SQL through the SQL procedure.

subsystem
a complete DL/I configuration, including the DL/I region controller and service modules, the DL/I databases, and the IMS engine.

synchronization point
   a commitment of completed updates and a time at which all DL/I resources that have
   been held since the previous synchronization point are released. Synchronization
   points are established explicitly by CHKP calls, which the IMS engine issues (by
   default) at the beginning and end of update processing. Synchronization points can
   be used to resume the processing of an interrupted job.

twins
   segments that represent multiple occurrences of the same segment type under a
   single parent.

undefined field
   a field that is not defined to DL/I in a Database Description (DBD). An undefined
   field is neither a sequence field nor a search field. You cannot use an undefined field
   in a segment search argument (SSA).

unqualified call
   a DL/I call that contains no segment search argument (SSA).

unqualified SSA
   a Segment Search Argument that specifies a segment type and, optionally, a
   command code.

Update call
   a DL/I call that signals the intent to alter (modify, delete, or add) information in the
   database.

update integrity
   the result of sharing database access so that two programs cannot access a record
   simultaneously if both programs intend to update the record.

variable
   a column in a SAS data set or in a SAS data view. The data values for each variable
   describe a single characteristic for all observations.

view descriptor
   a SAS/ACCESS file that defines part or all of the database management system
   (DBMS) data on PC file data that is described by an access descriptor. The access
   descriptor describes the data in a single product table or view, when views are
   allowed in the product. See also access descriptor.

windowing procedure
   a procedure that uses windows and menus to accomplish a SAS task. For example,
   the ACCESS, FSVIEW, and DBLOAD procedures are windowing procedures.

Index

Your Turn

If you have comments or suggestions about *SAS/ACCESS 9.1 Interface to IMS: Reference*, please send them to us on a photocopy of this page, or send us electronic mail.

For comments about this book, please return the photocopy to

SAS Publishing
SAS Campus Drive
Cary, NC 27513
E-mail: **yourturn@sas.com**

For suggestions about the software, please return the photocopy to

SAS Institute Inc.
Technical Support Division
SAS Campus Drive
Cary, NC 27513
E-mail: **suggest@sas.com**