CHAPTER

8

Introducing the IMS-DL/I DATA Step Interface

Introduction

Special SAS System 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-DL/I DATA step can contain standard SAS statements as well as the SAS statements that are used with the SAS/ACCESS interface to IMS-DL/I.

The first section of this chapter describes the syntax of the SAS statement extensions that are used with the SAS/ACCESS interface to IMS-DL/I. The next section describes basic DATA step programming techniques and considerations for this IMS-DL/I

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 chapter.

This chapter assumes that you understand the SAS DATA step and the statements used in the DATA step. See *SAS Language Reference: Dictionary* for details of 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 book's Chapter 2, "Understanding IMS-DL/I 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-DL/I .

Overview of the DATA Step Statement Extensions

In a DATA step, the SAS/ACCESS interface to IMS-DL/I 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 update calls.

The DL/I INFILE statement tells the SAS System where to find the parameters needed to build DL/I calls. Special DL/I INFILE statement extensions

- 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 chapter.

DL/I Input and Output Buffers

Two separate buffers are allocated by the SAS System 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 follow 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 follow 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 179 for more information on 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-DL/I 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 may 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-DL/I 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 for information on the databases, PSBs, segments, and fields used in this example and other examples in this book. This example uses the DLI option of the INFILE statement, which tells the SAS System 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 are not examined.

If you do not want to use these defaults, the special statement and product options that you can specify for IMS-DL/I are described later in this chapter.

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-DL/I 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 are retrieved from a CUSTOMER segment and placed in the input buffer. The data are 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.

Output 8.1 on page 155 shows the SAS log for this example.

Output 8.1 SAS LOG for ACCTSAM 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.
```

Output 8.2 on page 156 and Output 8.3 on page 157 show the output of this example.

Output 8.2 Customer Phone List

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

Customer Address List

customer_name	addr_ 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

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 are 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-DL/I 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 DL/I 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 153.

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 other 10 overrides are:

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 153.

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 may 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;
```

where

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 the SAS System 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 allows 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 non-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 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'**). Table 8.1 on page 162 shows the calls executed by DL/I INPUT statements and those executed by DL/I PUT statements.

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.

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

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 may not represent the length of the segment data correctly. This is due to the method the SAS System 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 overwrite the X'2E' characters until the segment data end. 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 happen 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 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 are the same as those 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 are formatted as shown in Table 8.2. 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-DL/I DATA Step Interface,” on page 219 for information on the format of the mask data for a non-database PCB.

If PCBF= is not specified, the mask data are 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 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 have been removed.) Consider the example for the level number of data in bytes 9-10 (see above). 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 are described in this table under bytes 33-200.

Bytes	Description
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* are contained in bytes 33-200. Key feedback data consist 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 are 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 "Using SSAs in IMS-DL/I DATA Step Programs" on page 234 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 173 for more information on checking status codes in IMS-DL/I 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-DL/I 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-DL/I 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 the SAS System 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 are 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 167 summarizes the DLI INFILE statement options and other options that affect the DATA step interface to IMS-DL/I, 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. The SAS System 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 the PSB and DLI. 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*	specify 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 may 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	specify 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. △

Output 8.4 on page 169 shows the output of this example.

Output 8.4 Multiple Input Sources in a DATA Step

The SAS System
2 Files Combined

OBS	soc_sec_ 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

OBS	zip_code	home_phone	prt	employer	savings_ account_ number	savings_ amount	savings_ date	savings_ 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 are then moved from the input buffer to SAS variables in the program data vector by the SAS System.

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 are 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 the SAS System 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 are already in the input buffer. This form of the INPUT statement is very useful in IMS-DL/I DATA step programs. Refer to "Using the DL/I INPUT Statement" on page 173 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 173.

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.

Output 8.5 on page 172 shows the output of this example.

Output 8.5 Customer Checking Accounts

The SAS System
Customer Checking Accounts

OBS	soc_sec_ number	customer_name	addr_ 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

OBS	country	zip_code	home_phone	office_phone	check_ account_ number	check_ amount	check_ date	check_ 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 183 later in this chapter for a detailed explanation of a sample IMS-DL/I 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 may or may 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**. The SAS System 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 the SAS System 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_ may 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 may 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 may 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 the SAS System 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 are 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 shows 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 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.

Output 8.6 on page 175 shows the output of this example:

```
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;
```

Output 8.6 Checking and Savings Accounts

The SAS System
Checking Accounts

OBS	account_ 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

OBS	account_ 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 are already in the buffer and does not issue a DL/I call. See "Example 8: Using the Blank INPUT Statement" on page 189 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-DL/I 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 form of the DL/I FILE statement is:

```
FILE PSBname DLI;
```

where

PSBname
specifies the same PSB referenced in the DATA step's DL/I INFILE statement. Refer to "The DL/I INFILE Statement" on page 160 for more information. A PSB name must be specified.

DLI
tells the SAS System 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 179.

Example 3: An Update Call

This DATA step reads MYDATA.CUSTOMER, an existing SAS data set containing information on 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 that

- 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 may or may 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 173 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 DATA step interface, except that the Version 7 DATA step interface supports 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
      if st = 'GE' then
         _error_ = 0;
      else
         link abendit;
   return;
```

* The effect of a trailing @ in a DL/I PUT statement is slightly different from 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 are moved to the input buffer. The next DL/I INPUT statement can then move data to the program data vector.

```
   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 182 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. In example 5, 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-DL/I DATA Step Examples

Complete IMS-DL/I DATA step examples are presented in this section. Each example illustrates one or more of the concepts described earlier in this chapter.

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-DL/I 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-DL/I 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 the SAS System 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), the SAS System 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.

Output 8.7 on page 185 shows the output of this example.

Output 8.7 Checking Account Balance Distribution by ZIP Code

Checking Account Balance Distribution By ZIP code

		check_amount		
		$ total	$ average	# of accounts
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 the SAS System 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 are 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 the SAS System.

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, the SAS System links to NEWPAGE and executes the statement.

⓰ The PUT statement specifies variables and positions to be written to the TRANREPT output file.

⓱ The DO group is terminated by the END statement.

⓲ Execution returns to the beginning of the DATA step when this RETURN statement executes.

⓳ 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, the SAS System links to this labeled statement and prints the specified heading.

⓴ After printing the heading, the SAS System 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 = '  ');
```

```
15     input @;
16     if st = '  ' then
         do;
17           input @13 check_amount pd5.2;
18           chck_bal=chck_bal + check_amount;
19         end;
20     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;
```

1. The DATA step creates a new SAS data set called BALANCES.
2. 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.
3. The KEEP statement tells the SAS System that the variables SOC_SEC_NUMBER, CHCK_BAL, and SAVE_BAL are the only variables to be included in the BALANCES data set.
4. The CHCK_BAL and SAVE_BAL variables are assigned an initial value of 0 and are reset to 0 for each new customer.
5. The INFILE statement specifies ACCTSAM as the PSB, and the DLI specification tells the SAS System 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 are 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 the SAS System.

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 are 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.

⓲ The variable CHCK_BAL is assigned a new value by adding the value of CHECK_AMOUNT just obtained from the CHCKACCT segment.

⓳ The END statement signals the end of the DO-group.

⓴ 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 the SAS System 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 allows 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-DL/I DATA step.

Output 8.8 on page 192 shows the output of this example.

Output 8.8 Customer Balances

Customer Balances

OBS	chck_bal	save_bal	soc_sec_ 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 the SAS System 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 the SAS System stops processing the DATA step. In a program issuing path calls with qualified SSAs, DL/I may 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 may 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, the SAS System 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, the SAS System returns to the PUT statement immediately after the FILE TRANREPT statement (statement 8) and continues execution of the step.

The correct bibliographic citation for this manual is as follows: SAS Institute Inc., *SAS/ACCESS® Interface to IMS-DL/I Software: Reference, Version 8*, Cary, NC: SAS Institute Inc., 1999. 316 pp.

SAS/ACCESS® Interface to IMS-DL/I Software: Reference, Version 8

ISBN 1–58025–548–5

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513.

1st printing, October 1999