CHAPTER 10

Advanced Topics for the IMS-DL/I DATA Step Interface

Introduction

This chapter discusses the use of the IMS-DL/I DATA step interface in some of the more advanced areas of DL/I programming, specifically, restarting update programs and constructing and using SSAs in DATA step programs. Because this information is intended for experienced DL/I programmers, there is little explanation of DL/I concepts and facilities in this chapter. The purpose of this information is to explain how SAS programs can be used to perform advanced DL/I functions, not to explain these functions.

Restarting an Update Program

There is always a risk of abnormal termination in any program. If an update program ends before processing is completed, you can complete processing by restarting the program, but you do not want to repeat updates that have already been made. DL/I's synchronization point feature helps to prevent duplicate updating in a restarted program.

If an online access region program or control region abends, the DL/I control region restores databases up to the last synchronization point. In a batch subsystem, a batch back-out utility must be executed to back out updates made since the last synchronization point. After backing out updates, any updates made by the program prior to the last synchronization point are intact and any made after the last synchronization point are not. When an update program is restarted after an abend, processing must resume at the synchronization point or duplicate updating may occur.

When building synchronization points into an online access region program, keep these things in mind:

- □ If the program updates a large number of database records between synchronization points, the DL/I control region enqueue tables can overflow and cause the online DL/I system to abend.
- □ The DL/I control region dynamic log can also overflow, which can cause the online access region or the whole online system to abend, depending on the online system used.
- □ On the other hand, if synchronization points are too frequent, they can tie up the master console and prevent other IMS messages from being sent.

Your database administration staff can help you determine how frequently synchronization points should be executed.

Examples

Programs that update DL/I databases should be designed to avoid the problem of duplicating updates if a restart is required. A DATA step program can be written to be inherently restartable, as in “Example 1” on page 220 or additional recovery logic can be built in for restarts, as in “Example 3” on page 230.

The CHKP call, which is used in these examples, is discussed in Chapter 9, “Using the SAS/ACCESS Interface to IMS-DL/I DATA Step Interface,” on page 197.

Example 1

This sample program updates the ACCTDBD database with data from wire transactions in the WIRETRN database. (See Chapter 3, “Defining SAS/ACCESS Descriptor Files,” on page 39 for complete database information on the WIRETRN database.) The program takes checkpoints and thereby releases database resources at regular intervals. Because the program is set up with checkpoints, it is appropriate for shared update access.

As you study this example, you will notice that the WIRETRAN segments are deleted from the WIRETRN database as soon as the ACCTDBD segments are successfully updated. There are no synchronization points between the ACCTDBD segment updates and the WIRETRAN deletions. Therefore, if an abend occurs and changes are backed out to the last synchronization point, you know that any WIRETRAN segments remaining in the database have not been processed. There is no danger of duplicating updates, and the program is inherently restartable. No special recovery logic is required for restarts.

The numbered comments following this program correspond to the numbered statements in the program:

```
data _null_;
   length ssa1 $ 43
          ssa2 $ 32
          ssa3 $ 9;
   retain blanks '                   '
          wirenum 0
          chkpnum 0;

❶    infile acctsam dli ssa=(ssa1,ssa2,ssa3) call=func
            pcb=pcbindex status=st segment=seg;


   /*  get hold next WIRETRAN segment
       from WIRETRN database        */
   func = 'GHN ';
   ssa1 = ' ';
   ssa2 = ' ';
   ssa3 = ' ';
❷  pcbindex = 5;
❸  input @1   wiressn  $char11.
           @12  wireacct $char12.
           @24  accttype $char1.
           @25  wiredate mmddyy8.
           @33  wiretime time8.
           @41  wireammt pd5.2
           @46  wiredesc $char40.;

   if st ¬= '  ' then
      if st = 'GB' then
         do;
            _error_ = 0;
            go to reptotal;
         end;
      else
         link abendit;

❹  if wirenum/5 = chkpnum then
      link chkp;

❺  amount = abs(wireammt);


   /*  insert debit or credit segment into
       ACCTDBD database         */

❻  if accttype = 'C' then
      do;
         ssa2 = 'CHCKACCT
          (ACNUMBER= '|| wireacct ||')';
         if wireammt > 0 then
            ssa3 = 'CHCKCRDT';
         else
            ssa3 = 'CHCKDEBT';
      end;
   else
❼     if accttype = 'S' then
         do;
            ssa2 = 'SAVEACCT
             (ACNUMBER= ' || wireacct || ')';
            if wireammt > 0 then
               ssa3 = 'SAVECRDT';
            else
               ssa3 = 'SAVEDEBT';
         end;
❽     else
      do;
         file log;
         put / '***** Invalid ' accttype= 'for '
```

```
           wiressn= wireacct= '*****';
         return;
      end;

9   ssa1 = 'CUSTOMER
      (SSNUMBER= ' || wiressn || ')';
    func = 'ISRT';
    pcbindex = 4;
    file acctsam dli;

10  put @1  amount   pd5.2
        @6  wiredate mmddyy6.
        @14 wiretime time8.
        @22 wiredesc $char40.
        @62 blanks   $char19.;

11  if st ¬= '  ' then
      if st = 'GE' then
         do;
            _error_ = 0;
            file log;
            if seg = 'CUSTOMER' then
               if accttype = 'C' then
                  put / '***** No CHCKACCT segment with '
                       wiressn= wireacct= '*****';
               else
                  put / '***** No SAVEACCT segment with '
                       wiressn= wireacct= '*****';
            else
               put / '***** No CUSTOMER segment with '
                    wiressn= '*****';
            return;
         end;
      else
         link abendit;


   /*  get hold checking or savings segment from
       ACCTDBD database  */

12  ssa3 = ' ';
    func = 'GHU';

    input @1   acnumber $char12.
          @13  balance  pd5.2
          @18  stmtdate mmddyy6.
          @26  stmt_bal pd5.2;

13  if st ¬= '  ' then
      link abendit;


   /*  replace checking or savings segment into
       ACCTDBD database   */
   balance = balance + wireammt;
   ssa1 = ' ';
   ssa2 = ' ';
   func = 'REPL';

   put @1   acnumber $char12.
       @13  balance  pd5.2
       @18  stmtdate mmddyy6.
       @26  stmt_bal pd5.2;

   if st ¬= '  ' then
      link abendit;


   /*  delete WIRETRAN segment from WIRETRN
       database                */

14  func = 'DLET';
    ssa1 = ' ';
    pcbindex = 5;
    put @1   wiressn  $char11.
        @12  wireacct $char12.
        @24  accttype $char1.
        @25  wiredate mmddyy8.
        @33  wiretime time8.
        @41  wireammt pd5.2
        @46  wiredesc $char40.;

15  if st ¬= '  ' then
      link abendit;

16  wirenum +1;
   return;

17  reptotal:
      file log;
      put // 'Number of Wire Transactions Posted ='
          wirenum 5.
           / '       Number of CHKP Calls Issued ='
          chkpnum 5.;
      stop;

18  chkp:
      chkpnum +1;
      func = 'CHKP';
      pcbindex = 1;
      file acctsam dli;
      put @1 'SAS'
          @4 chkpnum z5.;
      if st ¬= '  ' then
         link abendit;
      func = 'GHU ';
      ssa1 = 'WIRETRAN
```

```
          (SSNACCT = ' || wiressn || wireacct || ')';
         pcbindex = 5;
         input;
         if st ¬= '  ' then
            link abendit;
         return;

19    abendit:
         file log;
         put _all_;
         abort;
         run;
```

1 The program uses the ACCTSAM PSB. It contains PCBs for the ACCTDBD database and a PCB for the WIRETRN database, both of which are needed in this program.

2 PCBINDEX is set to point to the WIRETRN PCB.

3 The INPUT statement issues the GHN call to retrieve a WIRETRAN segment. If the call is not successful, and there is a **GB** status code (end-of-database), _ERROR_ is reset to 0 and the program branches to the REPTOTAL subroutine, which prints a summary report. For any other non-blank status code, the program skips to the ABENDIT subroutine, which forces an abend.

4 If the GHN call is successful, the program continues with a test to see if a CHKP call should be issued. Two accumulator variables, WIRENUM and CHKPNUM, are evaluated. WIRENUM is a value that is incremented each time an ACCTDBD database record is successfully updated. CHKPNUM is a value incremented each time a CHKP call is issued.

A CHKP call is issued any time the WIRENUM value divided by five equals CHKPNUM. That is, after five successful updates the program links to the subroutine labeled CHKP to issue the CHKP call. After the CHKP call, the program repositions itself in the database and continues processing the DATA step (see item 18).

5 The program goes on to set up for the REPL call that updates the balance information in the CHCKACCT and SAVEACCT segments of the ACCTDBD database. The absolute value of WIREAMMT is saved.

6 The value of the ACCTTYPE field is checked. If the ACCTTYPE is **C** (checking), a qualified SSA for the CHCKACCT segment is built by concatenating literal values with the value of the WIREACCT variable from WIRETRAN. The value of WIREAMMT is checked to build another, unqualified SSA that specifies the segment type to insert. If WIREAMMT is greater than 0, the SSA specifies the CHCKCRDT segment. If WIREAMMT is less than or equal to 0, the SSA specifies CHCKDEBT.

7 These statements are identical to the preceding group of statements, except that they build SSAs that define a savings account segment path rather than a checking account segment path.

8 If the value of ACCTTYPE is not **C** or **S**, the account type is not valid for the DATA step and an explanatory message is written to the log. Processing returns to the beginning of the DATA step again.

9 A qualified SSA for the CUSTOMER segment is built by concatenating literals with the value of WIRESSN from WIRETRAN. An ISRT call using the ACCTDBD PCB is set up.

10 The ISRT call is issued. Depending on the ACCTTYPE and the value of WIREAMMT, the inserted segment is a CHCKCRDT, CHCKDEBT, SAVECRDT, or SAVEDEBT segment, as specified by the SSAs. Since all four transaction segment types have the same format, only one PUT statement is needed.

11 This series of statements checks the status code after the ISRT call and writes explanatory messages to the SAS log if the status code is **GE** (segment not found). If the status code is a non-blank code other than **GE**, the program skips to the ABENDIT subroutine. Note that a FILE statement is issued, changing the output destination from the DL/I database to the SAS log.

12 If the ISRT call is successful, the account balance must be updated to reflect the amount of the processed transaction. First, a GHU call is set up. The variable SSA3 is set to blank, but SSA1 (for the CUSTOMER segment) and SSA2 (for the CHCKACCT or SAVEACCT segment) are still in effect. The INPUT statement issues the GHU call, which retrieves the parent CHCKACCT or SAVEACCT segment for the segment just added by the ISRT call.

13 If the GHU call fails, the program skips to the ABENDIT subroutine. Otherwise, the program updates the BALANCE value by adding the value of WIREAMMT from the wire transaction and issues a REPL call to replace the CHCKACCT or SAVEACCT segment retrieved by the GHU call. If the REPL call fails, the program branches to the ABENDIT subroutine.

14 If the REPL call is successful, a DLET call is issued for the WIRETRN database. The WIRETRAN segment just used to update the ACCTDBD database (retrieved with a GHN or GHU call earlier) is deleted. Because wire transaction segments are deleted as they are processed, this program can be restarted. That is, if the program stops for some reason (such as a system failure), it can be started again without any danger of duplicate transactions being added to the ACCTDBD database.

15 If the DLET call is not successful, the program links to the ABENDIT subroutine.

16 If the DLET call is successful, the WIRENUM accumulator variable is incremented, and processing returns to the beginning of the DATA step.

17 This subroutine is executed when a get call to the WIRETRN database returns a **GB** (end-of-database) status code (see item 2).

18 This subroutine issues the CHKP call after every fifth update (see item 4). If the CHKP call is not successful, the program links to the ABENDIT subroutine. If the CHKP call is successful, the database position has been lost. Therefore, a GHU call is set up to re-retrieve the WIRETRAN segment that is retrieved by the previous GHN call.

Because the values from the segment are still in the program data vector, the INPUT statement issuing the GHU call does not need to specify variable names.

If the GHU call fails for any reason, the program links to the ABENDIT subroutine. If the call succeeds, the program resumes processing at the assignment statement that follows the LINK CHKP statement.

⓳ These statements are executed when a bad status code is returned by one of the calls in the program. The contents of the program data vector are printed on the SAS log, and the program abends.

Example 2

Unless a program is structured so that it can be restarted without duplicating updates, special recovery logic should be included. The previous example shows a data program designed so that it can be restarted if necessary. The following example is not designed to be restarted and does not include special recovery logic. We include it as an example of the kind of program that should not be used for updating in a shared environment because it could result in erroneous data.

This program updates the ACCTDBD database with wire transactions that are stored in a sequential file rather than in the WIRETRN database. The program is similar to "Example 1" on page 220, but it is not designed to be restarted. Example program 3 illustrates the modifications to this program to add recovery logic.

The numbered comments following this sample program correspond to the numbered statements in the example:

```
filename tranin '<your.sas.tranin>' disp=shr;
data _null_;
   length ssa1 $31
          ssa2 $32
          ssa3 $9;
   retain blanks '                   '
          wirenum 0
          chkpnum 0;


   /*  get data from TRANIN flatfile      */

❶  infile tranin eof=reptotal;
   input  @1   cust_ssn $char11.
          @12  acct_num $char12.
          @24  accttype $char1.
          @25  wiredate mmddyy8.
          @33  wiretime time8.
          @41  wireammt pd5.2
          @46  wiredesc $char40.;
   if _error_ then
      link abendit;

❷   if wirenum/5  = chkpnum then
      link chkp;

❸   amount = abs(wireammt);
❹   if accttype = 'C' then
```

```
      do;
         ssa2 = 'CHCKACCT
           (ACNUMBER =' || acct_num || ')';
         if wireammt < 0 then
            ssa3 = 'CHCKCRDT';
         else
            ssa3 = 'CHCKDEBT';
       end;
   else
      if accttype = 'S' then
         do;
            ssa2 = 'SAVEACCT
              (ACNUMBER =' || acct_num || ')';
            if wireammt < 0 then
               ssa3 = 'SAVECRDT';
            else
               ssa3 = 'SAVEDEBT';
         end;
      else
         do;
            file log;
            put / '***** Invalid ' accttype= 'for '
               cust_ssn=  acct_num= '*****';
            return;
         end;

   /*  insert debit or credit segment into
       ACCTDBD database        */

5   infile acctsam dli ssa=(ssa1,ssa2,ssa3) call=func
        pcb=pcbindex status=st segment=seg;
   ssa1 = 'CUSTOMER(SSNUMBER =' || CUST_SSN || ')';
   func = 'ISRT';
   pcbindex = 4;
   file acctsam dli;
   put  @1  amount   pd5.2
        @6  wiredate mmddyy6.
        @14 wiretime time8.
        @22 wiredesc $char40.
        @62 blanks   $char19.;

6   if st ¬= '  ' then
      if st = 'GE' then
         do;
            _error_ = 0;
            file log;
            if seg = 'CUSTOMER' then
               if accttype = 'C' then
                  put / '***** No CHCKACCT segment with '
                        cust_ssn= acct_num= ' *****';
            else
               put / '***** No SAVEACCT segment with '
                     cust_ssn= acct_num= ' *****';
            else
            put / '***** No CUSTOMER segment with
                  ' cust_ssn= '*****';
         return;
      end;
   else
      link abendit;


/*  get hold checking or savings segment from
    ACCTDBD database  */

ssa3 = '  ';
```
❼
```
 func = 'GHU';
  input @1  acnumber $char12.
        @13 balance  pd5.2
        @18 stmtdate mmddyy6.
        @26 stmt_bal pd5.2;
  if st ¬= '   ' then
     link abendit;
  balance = balance + wireammt;


/*  replace checking or savings segment into
    ACCTDBD database   */

ssa1 = ' ';
ssa2 = ' ';
func = 'REPL';
```
❽
```
 put @1  acnumber $char12.
     @13 balance  pd5.2
     @18 stmtdate mmddyy6.
     @26 stmt_bal pd5.2;
if st ¬= '   ' then
   link abendit;
```
❾
```
 if wireammt > 0 then
   debtnum +1;
else
   crdtnum +1;
```
❿
```
 wirenum +1;
return;

reptotal:
   file log;
   put // 'Number of debit  transactions posted ='
        debtnum 8.
        / 'Number of credit transactions posted ='
        crdtnum 8.;
   stop;
```
⓫
```
 chkp:
   chkpnum +1;
```

```
      func = 'CHKP';
      pcbindex = 1;
      file acctsam dli;
      put @1 'SAS'
          @4 chkpnum z5.;
      if st ¬= '  ' then
         link abendit;
      return;

      abendit:
         file log;
         put _all_;
      abort;
run;
filename tranin clear;
```

❶ The standard INFILE statement specifies the external sequential file containing the data to update ACCTDBD. The fileref is TRANIN. When the end-of-file condition is set, the program branches to the REPTOTAL subroutine to print a summary report. The standard INPUT statement reads a record from TRANIN. If any error occurs, the program links to the ABENDIT subroutine.

❷ As in the previous example, this program issues CHKP calls after every fifth update. If the value of WIRENUM divided by five is equal to the value of CHKPNUM, the program links to a section that issues the CHKP call.

❸ The DATA step sets up for the REPL call that will update balance information in the CHCKACCT and SAVEACCT segments of the ACCTDBD database. The absolute value of WIREAMMT is saved.

❹ Depending on the value of ACCTTYPE, SSAs are built for the CHCKACCT and either the CHCKDEBT or CHCKCRDT segments, or for the SAVEACCT and either the SAVEDEBT or SAVECRDT segments.

❺ The DL/I INFILE statement specifies the ACCTSAM PSB. An ISRT call for the ACCTDBD database is formatted and issued. Depending on the account type and transaction type, a new CHCKCRDT, CHCKDEBT, SAVECRDT, or SAVEDEBT segment is inserted.

❻ This section checks status codes and prints explanatory messages on the SAS log if the status code is **GE** (segment not found). For other non-blank status codes, the program links to the ABENDIT subroutine.

❼ If the ISRT call is successful, a GHU call is issued to retrieve the parent of the added segment. The status code is checked after the call and, if it is not successful, the program links to the ABENDIT routine.

❽ If the GHU call is successful, the account balance is updated by a REPL call. The status code is checked after the call and, if it is not successful, the program links to the ABENDIT routine.

❾ Accumulator variables count the number of debits and credits posted by the program. These values are used to print a summary report.

⑩ The WIRENUM variable is incremented. It is used to determine whether or not a CHKP call is needed (see item 2).

⑪ This section is like the one in “Example 1” on page 220, but no GHU call is issued to re-establish database position because there is no database position to maintain. (This is because the wire transactions are not coming from an IMS database on which the program can reposition.)

Example 3

This example is a modified version of “Example 2” on page 226. The modifications consist of the recovery logic added to allow the program to be restarted. The same sequential file is used to update the ACCTDBD database.

The numbered comments following this program describe the statements added to allow a restart:

```
filename tranin '<your.sas.tranin>' disp=shr;
❶ filename restart '<your.sas.restart>' disp=shr;
data _null_;
   length ssa1 $31
          ssa2 $32
          ssa3 $9
          chkpnum 5;
   retain wireskip
          wirenum 0
          chkpnum 0
          first 1
          debtnum
          crdtnum
          errnum 0
          blanks '                    ';

   infile restart eof=process;
   input @1  chkpid   5.
         @6  chkptime datetime13.
         @19 chkdebt  8.
         @27 chkcrdt  8.
         @35 chkerr   8.;

   wireskip = chkdebt + chkcrdt + chkerr;

   file log;
   put 'Restarting from checkpoint ' chkpid
       'taken at ' chkptime datetime13.
       ' to bypass ' wireskip 'trans already processed';

   do while(wireread < wireskip);
      infile tranin;
      input  @1  cust_ssn $char11.
             @12 acct_num $char12.
             @24 accttype $char1.
             @25 wiredate mmddyy8.
             @33 wiretime time8.
             @41 wireammt pd5.2
             @46 wiredesc $char40.;
       wireread + 1;
    end;

    debtnum = chkdebt;
    crdtnum = chkcrdt;
    wirenum = debtnum + crdtnum;
    errnum = chkerr;

❷     process:
       infile tranin eof=reptotal;
       input  @1  cust_ssn $char11.
              @12 acct_num $char12.
              @24 accttype $char1.
              @25 wiredate mmddyy8.
              @33 wiretime time8.
              @41 wireammt pd5.2
              @46 wiredesc $char40.;
       if _error_ then
          link abendit;

       if wirenum/5 = chkpnum or first =1 then
          do;
             link chkp;
             first =0;
          end;

       amount = abs(wireammt);

       if accttype = 'C' then
          do;
             ssa2 = 'CHCKACCT
              (ACNUMBER= ' || acct_num || ')';
             if wireammt < 0 then
                ssa3 = 'CHCKCRDT';
             else
                ssa3 = 'CHCKDEBT';
          end;
       else
          if accttype = 'S' then
             do;
              ssa2 = 'SAVEACCT
               (ACNUMBER= ' || acct_num || ')';
                if wireammt < 0 then
                   ssa3 = 'SAVECRDT';
                else
                   ssa3 = 'SAVEDEBT';
             end;
❸           else
             do;
                file log;
                put / '***** Invalid ' accttype= 'for '
                   cust_ssn= acct_num= '*****';
                go to outerr;
             end;
```

```
infile acctsam dli ssa=(ssa1,ssa2,ssa3) call=func
       pcb=pcbindex status=st segment=seg;

ssa1 = 'CUSTOMER(SSNUMBER= ' || cust_ssn || ')';
func = 'ISRT';
pcbindex = 4;
file acctsam dli;
put  @1  amount   pd5.2
     @6  wiredate mmddyy6.
     @14 wiretime time8.
     @22 wiredesc $char40.
     @62 blanks   $char19.;

if st ¬= '  ' then
   if st = 'GE' then
      do;
         _error_ = 0;
         file log;
         if seg = 'CUSTOMER' then
            if accttype = 'C' then
              put / '***** No CHCKACCT segment with '
                   cust_ssn= acct_num= '*****';
            else
              put / '***** No SAVEACCT segment with '
                   cust_ssn= acct_num= '*****';
         else
           put / '***** No CUSTOMER segment with '
                cust_ssn= '*****';
         go to outerr;
      end;
   else
      link abendit;

ssa3 = ' ';
func = 'GHU ';
input  @1  acnumber $char12.
       @13 balance  pd5.2
       @18 stmtdate mmddyy6.
       @26 stmt_bal pd5.2;
if st ¬= '  ' then
   link abendit;

balance = balance + wireammt;
ssa1 = ' ';
ssa2 = ' ';
func = 'REPL';
put  @1  acnumber $char12.
     @13 balance  pd5.2
     @18 stmtdate mmddyy6.
     @26 stmt_bal pd5.2;
if st ¬= '  ' then
   link abendit;
   if wireammt > 0 then
      debtnum = debtnum +1;
   else
      crdtnum = crdtnum +1;
   wirenum = wirenum +1;
   return;

   reptotal:
      file log;
      put // 'Number of debit  transactions posted ='
          debtnum 8.
           / 'Number of credit transactions posted ='
          crdtnum 8.;
      stop;

❹   chkp:
      chkpnum +1;
      chkptime = datetime();
      file log;
      put @1  'Next checkpoint will be'
          @25 chkpnum
          @30 chkptime datetime13.
          @43 debtnum
          @51 crdtnum
          @59 errnum;
      func = 'CHKP';
      pcbindex = 1;
      file acctsam dli;
      put @1 'SAS'
          @4 chkpnum z5.;
      if st ¬= '  ' then
         link abendit;
      return;

   outerr:
      errnum = errnum +1;
      return;

   abendit:
      file log;
      put _all_;
      abort;
run;
filename tranin clear;
filename restart clear;
```

❶ This group of statements initiates the restart, if a restart is necessary. The standard INFILE statement points to a file with fileref RESTART. The RESTART file has one record, a "control card" with data that will determine where processing should resume in the sequential input file. The data in the RESTART file are taken from the last checkpoint message written on the SAS log by the program that ended before completing processing. The message includes the number and time of the last checkpoint, and the values of the accumulator variables counting the number of debit transactions posted (CHCKDEBT), credit transactions posted (CHCKCRDT), and the number of bad records in the TRANIN file (CHKERR).

The RESTART DD statement can be dummied out to execute the program normally (not as a restart). If RESTART is dummied out in the control language, end-of-file occurs immediately, and the program skips to the PROCESS subroutine (see item 6), as indicated by the EOF= option.

The WIRESKIP variable is the sum of CHCKDEBT, CHCKCRDT, and CHKERR; that is, WIRESKIP represents the number of records in TRANIN that were processed by the program before the last checkpoint.

A message is written to the SAS log that shows the checkpoint from which processing resumes.

To position itself at the correct TRANIN record, the program reads the number of records indicated by the WIRESKIP variable. In other words, the program re-reads all records that were read in the first execution of the program, up to the last checkpoint.

The values of DEBTNUM, CRDTNUM, WIRENUM, and ERRNUM are reset so that the final report shows the correct number of transactions. Otherwise, the report would show only the number of transactions processed in the restarted execution.

❷ These statements are the same as the statements in “Example 2” on page 226 except that they are labeled "PROCESS." If the program is not being restarted, end-of-file for the INFILE RESTART occurs immediately, and the program branches to this subroutine.

❸ If the value of ACCTTYPE is anything but **C** or **S**, the TRANIN record is a bad record. The program prints a message on the SAS log and branches to the OUTERR subroutine, which increments the ERRNUM accumulator variable.

❹ The CHKP call is issued by this group of statements. This group is like that in “Example 2” on page 226 except that a message about the checkpoint is also printed on the SAS log. This message provides the necessary information for a restart.

Note that the message is written to the SAS log before the CHKP call is actually issued, so it is possible that a system failure could occur between the time the message is written and the time the call is issued. Therefore, if a restart is necessary, you should verify that the last checkpoint referenced in the SAS log is the same as the last checkpoint in the DL/I log. This can be done by comparing checkpoint IDs.

Using SSAs in IMS-DL/I DATA Step Programs

When a DATA step program uses qualified calls, you designate variables containing the SSAs with the SSA= option in the DL/I INFILE statement. The values of SSA variables do not have to be constants. They can be built by the program using SAS assignment statements, functions, and operators. You can construct SSAs conditionally and change SSA variable values between calls.

The Concatenation Operator

One of the techniques for building an SSA is to incorporate the value of another variable in the SSA variable's value. This can be accomplished with the concatenation operator (||), as in this example:

```
ssa1='CUSTOMER(SSNUMBER ='||ssn||')';
```

This statement assigns a value to SSA1 that consists of the literal CUSTOMER(SSNUMBER =, the current value of the variable SSN, and the right parenthesis. If the current value of SSN is 303-46-4887, the SSA is

```
CUSTOMER(SSNUMBER =303-46-4887)
```

Note: The concatenation operator acts on character values. If you use a numeric variable or value with the concatenation operator, the numeric value is converted automatically to character using the BEST12. format. If the value is less than 12 bytes, it is padded with blanks and, if longer than 12 bytes, it could lose precision when converted. If you want to insert a numeric value via concatenation, you should explicitly convert the value to character with the PUT function (described in the next section). △

The PUT Function

SSA variables in a DATA step program must be character variables. However, you may sometimes need to qualify an SSA with a numeric value. To insert a numeric value in an SSA character variable, you can use the SAS PUT function.* For more information on the PUT statement, see *SAS Language Reference: Dictionary*.

The PUT function's form is

PUT(*argument1, format*)

where *argument1* is a variable name or a constant, and *format* is a valid SAS format of the same type (numeric or character) as *argument1*. The PUT function writes a character string that consists of the value of *argument1* output in the specified format. The result of the PUT function is always a character value, regardless of the type of the function's arguments. For example, in this statement

```
newdate=put(datevalu,date7.);
```

the result of the PUT function is a character string assigned to the variable NEWDATE, a character variable. The result is a character value even though DATEVALU and the DATE7. format are numeric. If DATEVALU=38096, the value of NEWDATE is:

```
newdate='20APR64'
```

Using the PUT function, you can translate numeric values for use in SSAs. For example, to select WIRETRAN segments with WIREAMMT values less than $500.00, you could construct an SSA like this:

```
maxamt=500;
ssa1='WIRETRAN(WIREAMMT <'||put(maxamt,pd5.2)||')';
```

First, you assign the numeric value to be used as the search criterion to a numeric variable. In this case, the value 500 is assigned to the numeric variable MAXAMT.

* The PUT function can also be used to format a character value with any valid character format.

Then you construct the qualified SSA using concatenation and the PUT function. The PUT function’s result is a character string consisting of the value of MAXAMT in PD5.2 format.

Consider a more complicated example using the ACCTDBD database. In this case, you want to select all checking accounts for which the last statement was issued a month ago today or more than 31 days ago.

The following SAS statements illustrate one approach to constructing an SSA to select the appropriate accounts. The numbered comments after this example correspond to the numbered statements:

```
data _null_;
❶   tday = today();
❷   d = day(tday);
   m = month(tday);
   y = year(tday);

❸   if d = 31 then
      if m = 5 or
         m = 7 or
         m = 10 or
         m = 12 then
         d = 30;
❹   if m = 3 then
      if d < 28 then
         d = 28;
   if m = 1 then
      do;
         m = 12;
         y = y - 1;
      end;
   else
      m = m - 1;

❺   datpmon = mdy(m,d,y);
❻   datem31 = tday - 31;

❼   ssa1 = 'CHCKACCT
       (STMTDATE= ' || put(datpmon,mmddyy6.) ||
       '| STMTDATE> ' || put(datem31,mmddyy6.) || ')';
   stop;
run;
```

❶ Use the SAS function TODAY to produce the current date as a SAS date value and assign it to the variable TDAY.

❷ Use the SAS functions DAY, MONTH, and YEAR to extract the corresponding parts of the current date and assign them to appropriate variables.

❸ Modify D values to adjust when previous month has fewer than 31 days.

❹ Modify the month variable (M) to contain the prior month value.

❺ Assign the SAS date value for last month, the same day as today, to the variable DATPMON.

❻ Subtract 31 from the SAS date representing today's date and assign the value to the variable DATEM31.

❼ To build the SSA, concatenate these elements:

- □ a literal that is composed of the segment name (CHCKACCT), a left parenthesis, search field name (STMTDATE), and the relational operator =.
- □ a character string consisting of the value of DATPMON output in the MMDDYY6. format. The character string is the result of the PUT function.
- □ a literal consisting of the Boolean operator | (or), the search field name (STMTDATE), and the relational operator >.
- □ a character string consisting of the value of DATEM31 output in the MMDDYY6. format. The character string is the result of the PUT function.
- □ a literal consisting of a right parenthesis.

If these statements are executed on 28 March 1995, the value of SSA1 is

```
CHCKACCT(STMTDATE =02/28/95|STMTDATE >02/28/95)
```

Setting SSAs Conditionally

Using SAS IF-THEN/ELSE statements, SSA variables can be assigned values conditionally. Consider “Example 2” on page 226 in which the ACCTDBD database is updated with transaction information stored in a standard sequential file with fileref TRANIN. Each TRANIN record contains data for one deposit or withdrawal transaction for a checking or savings account. The program uses the TRANIN records to construct new CHCKDEBT, CHCKCRDT, SAVEDEBT, or SAVECRDT segments and then inserts the new segment in the ACCTDBD database. Notice that the concatenation operator (||) is used to incorporate the value of the ACCT_NUM variable in the SSA.

The program first reads a record from the TRANIN file and then determines whether the data are for a checking or a savings account by evaluating the value of the variable ACCTTYPE. If ACCTTYPE='C', the program constructs a qualified SSA for a CHCKACCT segment. Next, the program determines whether the record represents a debit or credit transaction and builds an unqualified SSA for a CHCKDEBT or CHCKCRDT segment, as appropriate.

If ACCTTYPE='S', a qualified SSA for a SAVEACCT segment is built, and then an unqualified SSA for a SAVEDEBT or SAVECRDT segment is set up.

Changing SSA Variable Values between Calls

A DATA step program can issue multiple calls within a DATA step execution, and the value of an SSA variable can be changed between each call. An example of this is the following code, which is used in “Example 4: Issuing REPL Calls” on page 180 in Chapter 8, “Introducing the IMS-DL/I DATA Step Interface,” on page 151:

```
data _null_;
   set ver6.newaddr;
   length ssa1 $31;
   infile acctsam dli ssa=ssa1 call=func status=st
     pcbno=4;
   ssa1 = 'CUSTOMER(SSN =' || ssn || ')';
```

```
      func = 'GHU ';
      input;
      if st = '  ' then
         do;
            func = 'REPL';
            ssa1 = ' ';
            file acctsam dli;
            put _infile_ @;
            put @52 newaddr1  $char30.
                @82 newaddr2  $char30.
                @112 newcity  $char28.
                @140 newstate $char2.
                @162 newzip   $char10.;
            if st ¬= '  ' then
               link abendit;
         end;
      else
         if st = 'GE' then
            do;
               _error_ = 0;
               stop;
            end;
         else
            link abendit;
      return;

      abendit:
         file log;
         put _all_;
         abort;
   run;
```

These statements are part of a program that updates CUSTOMER segments in the ACCTDBD database with information from the SAS data set VER6.NEWADDR. CUSTOMER segments are retrieved using GHU calls with a qualified SSA, SSA1. Once a segment is retrieved, the data from the SAS data set are overlaid on the old values of the segment and a REPL call is issued. Since a REPL call acts on a segment retrieved previously, no SSA is needed. Therefore, the value of the SSA1 variable is changed to blanks before the REPL call is issued.

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