Paper 3566-2019

PROC DS2, An Intro

Dari Mazloom, USAA

ABSTRACT

PROC DS2 is a new programming feature of SAS® programming language which introduces concepts of methods, modularity, and encapsulation. A DS2 program encapsulates variables and code. DS2 is included with BASE SAS® language 9.4. based on the DATA step language. DS2 provides new data types which make possible to have better precision for data and better interaction with external databases. DS2 provides three separate but related system-defined methods which run in order. The methods are: INIT(), RUN(), and TERM().
In addition, DS2 provides a capability for multi-threading. It also allows the SAS® practitioners to define their own custom methods called user-defined methods. This paper concentrates on how SAS® programmers can use PROC DS2 to write modularized programs which make for easier testing and maintenance. It also introduces the basic programming instructions(syntax) associated with the construction and use of PROC DS2.

INTRODUCTION

SAS® has introduced new capabilities via PROC DS2 to write user-defined methods. In the following topics, we will discuss the basic constructs of PROC DS2.
This paper covers the following topics:

- Why use PROC DS2?
- Data types in PROC DS2
- The constructs of PROC DS2
- What is User-Define Method?
- What is Variable Scope?
- What is Multi-Threading?
- Appendix A: Variable Scope
- Appendix B: List of Data Types

This paper does NOT cover the following topic(s):

- Interaction with external databases will NOT be covered.
- Object-Oriented Programming is NOT covered.

TOPIC 1: WHY USE PROC DS2?

PROC DS2 extends the power of DATA step into OO programming.

It makes it possible for programs to be multi-threaded. It provides an extensive list of data types which makes the SAS® data compatible with other databases.

TOPIC 2: DATA TYPES IN PROC DS2

PROC DS2 provides an extensive list of data types which extends the current SAS® data types.

The new data types facilitate a seamless interaction among SAS® and external databases.

The new data types are: BIGINT, BINARY(n), CHAR(n), DATE, DECIMAL|NUMERIC(p,s),

DOUBLE, FLOAT, INTEGER, NCHAR(n), NVARCHAR(n), REAL, SMALLINT,

TIME(p), TIMESTAMP(p), TINYINT, VARBINARY(n), VARCHAR(n).

For details of the new data types, please consult: What Are the

Data Types in SAS® 9.4 DS2 Language Reference, Sixth Edition

TOPIC 3: THE CONSTRUCTS OF A DS2 PROGRAM

A DS2 program is comprised of a list of variable declarations together with a list of methods. A programming block is a set of code contained within two key words:

Entire program is within;	DATA…………ENDDATA;
Method is declared within:	METHOD ……END;
Thread is declared within:	THREAD……. ENDTHREAD;
PACKAGE is declared within:	PACKAGE……ENDPACKAGE;

Example 1: Let’s look at and analyze a simple DS2 program.

```
PROC DS2 LIBS=WORK;
      DATA WRK_T1 (OVERWRITE=YES);
        /*The scope of the following data elements is the current
        DS2*/
```

```
            DCL VARCHAR (20) MAKE;
            DCL VARCHAR (20) MODEL;
            DCL INT COUNTER;

            /*The INIT method is run automatically once at the*/
            /*beginning of the program to perform all
              initialization*/
            METHOD INIT();
                  PUT 'METHOD=INIT';
                  COUNTER = 0;
                     MAKE = 'FORD';
                     MODEL = 'F-150';
            END;

            /*The RUN method is run automatically after the
             INIT method.*/
            METHOD RUN();
                  PUT 'METHOD=RUN';
                  COUNTER = COUNTER + 1;
                     PUT MAKE=;
                     PUT MODEL=;
                  OUTPUT WRK_T1;
            END;

            /*The TERM method is run automatically once at the end of*/
           /*the process and it performs house cleaning tasks. */

            METHOD TERM();
                  PUT 'METHOD=TERM';
            END;
      ENDDATA;
RUN;
QUIT;
```

TOPIC 4: USER-DEFINED METHODS IN A DS2 PROGRAM

In addition to the three DS2 system-defined methods (INIT, RUN, TERM), DS2 allows the Programmers to define their own method called User-Defined methods.

Important note: All user-defined methods need to be defined and placed prior to the placement of methods, INIT, RUN, and TERM.

Example 1: The following is a DS2 program that contains user-defined methods.

```
PROC DS2 LIBS=WORK;
     DATA WRK_T1 (OVERWRITE=YES);
          DCL VARCHAR(20) MAKE;
          DCL VARCHAR(20) MODEL;
          DCL MSRP DOUBLE;
          DCL INT COUNTER;

            /*User-defined method: Assigns a value to MAKE*/
          METHOD SetMake(VARCHAR(20) MakeName);
                  MAKE = MakeName;
          End;

            /*User-defined method: Returns the value of MAKE to
          /the calling section*/
          METHOD GetMake() RETURNS VARCHAR(20);
                  RETURN MAKE;
          End;

            /*User-defined method: Assigns a value to MODEL*/
          METHOD SetModel(VARCHAR(20) ModelName);
                  MODEL = ModelName;
          End;

            /*User-defined method: Returns the value of MODEL to*/
          /*the calling section*/
         METHOD GetModel() RETURNS VARCHAR(20);
                  RETURN MODEL;
         End;

         METHOD ComputeDealerCost() Returns DOUBLE;
              DCL DOUBLE DealerCost;

              DealerCost = MSRP - (MSRP * 0.30);

              Return DealerCost;
         END;
```

The following is a user-defined method that accepts an input parameter.
It also a different version of the method ComputerDealerCost:

```
         METHOD ComputeDealerCost(DOUBLE CostFactor) RETURNS DOUBLE;

              RETURN (MSRP - (MSRP * CostFactor));
         END;
```

```
          METHOD INIT();
                PUT 'METHOD=INIT';
                COUNTER = 0;
                   MAKE='';
                   MODEL='';
          END;

          METHOD RUN ();
               DCL VARCHAR(20) TempMake;
               DCL VARCHAR(20) TempModel;
               DCL DOUBLE TempMSRP1;
               DCL DOUBLE TempMSRP2;

                PUT 'METHOD=RUN';
                 MSRP = 50000;
                 SetMake('FORD');
                 SetModel('F-150');
                 TempMake = GetMake();
                 TempModel = GetModel();
                 TempMSRP1 = ComputeDealerCost();
                  TempMSRP2 = ComputeDealerCost(0.40);
                 PUT TempMake=;
                 PUT TempModel=;
                 COUNTER = COUNTER + 1;
                 PUT COUNTER=;
                 OUTPUT WRK_T1;
          END;

          METHOD TERM();
                PUT 'METHOD=TERM';
          END;
     ENDDATA;
RUN;
QUIT;
```

TOPIC 5: WHAT IS Variable Scope?

Local Scope: When a variable is referenced, DS2 first looks inside the block in which the reference is made. If the variable in found in that block, DS2 will use the variable. In this case, the variable is local to the block where it is referenced.

```
          METHOD INIT();
              DCL VARCHAR(20) MAKE;
```

```
        DCL VARCHAR(20) MODEL;
        DCL INT COUNTER;

        COUNTER = 0;
        MAKE = 'FORD';
        MODEL = 'F-150';
    END;
```

Global Scope: If the variable in NOT found in that block, DS2 will look outside of the block for the variable. In this case, the variable is global to the block where it is referenced.

```
DCL VARCHAR(20) MAKE;
DCL VARCHAR(20) MODEL;
DCL INT COUNTER;

 METHOD INIT();
    PUT 'METHOD=INIT';
    COUNTER = 0;
    MAKE = 'FORD';
    MODEL = 'F-150';
 END;
```

TOPIC 6: WHAT IS Multi-Threading?

Multi-threading is used for programs that involve intense computational or other intense programming.

Example: First, we create a test dataset outside of PROC DS2.

```
DATA WORK.TEST1;

DO COUNT = 1 TO 500000;

        PRICE = COUNT;

        DISCCOUNT = COUNT * 0.20;

        OUTPUT;

END;
```

```
DROP COUNT;

RUN;
```

Next, we create a thread:

```
PROC DS2;

   THREAD COMPUTE / OVERWRITE=YES;

   DCL DOUBLE TOTAL;

   DCL BIGINT TOTAL_ROWS;

   DROP TOTAL_ROWS;

  METHOD INIT();

 TOTAL_ROWS = 0;

END;

METHOD RUN();

   SET WORK.TEST1;

   BY PRICE;

   TOTAL = PRICE – DISCOUNT;

   TOTAL = ROWS + 1;

END;

METHOD TERM();

   PUT ‘Thread=’ ThreadId_ ‘COMPLETED=‘  TOTAL_ROWS  ‘ROWS’;

END;

DATA WORK.TEST1_RESULTS (OVERWRITE=YES);

DCL THREAD COMPUTE MY_THREAD;
```

```
METHOD RUN();

SET FROM MY_THREAD Threads= 3;

END;

ENDDATA;

RUN;

QUIT;
```

Resutlts for three threads:

```
**Thread 1 COMPLETED 194244 ROWS

**Thread 2 COMPLETED 106870 ROWS

**Thread 3 COMPLETED 198886 ROWS

NOTE: Created thread compute in data set work.compute.

NOTE: Execution succeeded. 500000 rows affected.

NOTE: PROCEDURE DS2 used (Total process time):

 Real time  6.07 seconds              CPU time  2.96 seconds
```

CONCLUSION

DS2 language provides a powerful set of techniques to write code that encapsulate data and methods in the same program while providing capabilities for multi-threading. This paper is providing just a brief introduction to the capabilities of DS2. For more details and techniques, the practitioners should consult the following references.

REFERENCES

SAS® 9.4 DS2 Language Reference, Sixth Edition - SAS Support

ABOUT THE AUTHOR

Dari Mazloom is a lead business intelligence analyst with over thirty years of practical experience in a number of programming languages such as SAS®, C#, C++, JAVA, VB, VBA, Smalltalk, COBOL, Easytrieve, JCL.

CONTACT INFORMATION

Your comments and questions are valued and encouraged.

Contact the author at:

Dari Mazloom

USAA

Dariush.Mazloom@USAA.Com

Appendix A: VARIABLE SCOPE

Global Variables

A variable with global scope, a global variable, is declared in one of three ways: in the outermost programming block of a DS2 program, using a DECLARE statement, implicitly declared inside a programming block using a SET statement, or implicitly declared inside a programming block by using an undeclared variable. Variables with global scope can be accessed from anywhere in the program and exist for the duration of the program. Global variables can be used in a THIS expression in any program block.

Local Variables

A variable with local scope, a local variable, is declared within an inner programming block, such as a method or package, by using a DECLARE statement. Variables with local scope are known only within the inner programming block where they are declared.

Global and Local Variables in DS2 Output

Only global variables, by default, are included in the output. Local variables that are used for program loops and indexes do not need to be explicitly dropped from the output. Local variables are always created at the start of a method invocation, such as an iteration of the implicit loop of the RUN method, and are destroyed at the end of each invocation. Therefore, it is not recommended to use local variables as accumulator variables in the RUN method.

All global variables are named in the program data vector (PDV). The PDV is the set of values that are written to an output table when DS2 writes a row.

Example of Global and Local Variables

The following program shows both global (A, B, and TOTAL) and local variables (C):

```
PROC DS2;
      DATA;
         DCL INT A;    /*1 A is a global variable*/
          Mthod INIT();
              DCL INT C; /*2 C is a local variable*/
            A = 1;
```

```
                B = 2; /*3 B is un-declared so it is global */
                  C = A + B;
                   total = a + b + c; /*5 TOTAL is un-declared so it
                                                        is global*/
            END;
ENDDATA;
RUN;
QUIT;
```

- A is a global variable because it is declared in the outermost DS2 program.

- C is a local variable because it is declared inside the method block, METHOD INIT ().

- Because A is a global variable, it can be referenced within the method block, METHOD INIT().

- Because B is not declared in METHOD INIT(), it defaults to being a global variable.
  DS2 assigns B a data type of DOUBLE. B appears in the PDV and the output table.

- THIS.TOTAL simultaneously declares the variable TOTAL as a global variable with the data type of DOUBLE and assigns a value to it based on the values of A, B, and C.

APPENDIX B: LIST OF SAS® DS2 DATA TYPES

Please see: SAS® What are DS2 Data Types?

Data Type	Description
BIGINT	Stores a large signed, exact whole number, with a precision of 19 digits. The range of integers is -9,223,372,036,854,775,8 Integer data types do not store decimal values; fractional portions are discarded.
BINARY(*n*)	Stores fixed-length binary data, where *n* is the maximum number of bytes to store. The maximum number of bytes is required to store each value regardless of the actual size of the value.
CHAR(*n*)	Stores a fixed-length character string, where *n* is the maximum number of characters to store. The maximum number of characters is required to store each value regardless of the actual size

	of the value. If `char(10)` is specified and the character string is only five characters long, the value is right padded with spaces.
DATE	Stores a calendar date. A date literal is specified in the format *yyyy-mm-dd*: a four-digit year (0001 to 9999), a two-digit month and a two-digit day (01 to 31). For example, the date September 24, 1975 is specified as `1975-09-24`. DS2 complies with ANSI SQL:1999 standards regarding dates. However, not all data sources support the full range of date. For example, dates between 0001-01-01 and 1582-12-31 are not valid dates for a SAS® data set or an SPD data set.
DECIMAL|NUMERIC(*p*,*s*)	Stores a signed, exact, fixed-point decimal number, with user-specified precision anc scale. The precision and scale determine the position of the decimal point. The precision is the maximum number of digits that can be stored to the left and righ of the decimal point, with a range of 1 to 52. The scale is the maximum number of digits that can be stored following the decimal point. For example, `decimal(9,2)` stores decimal numbers up to nine digits, with a two-digit fixed-point fractional portion, such as 1234567.89.
DOUBLE	Stores a signed, approximate, double-precision, floating-point number. Allows numbers of large magnitude and permits computations that require many digits of precision to the right of the decimal point.
FLOAT	Stores a signed, approximate, double-precision, floating-point number. Data defined as FLOAT is treated the same as DOUBLE.
INTEGER	Stores a regular size signed, exact whole number, with a precision of ten digits. The range of integers is -2,147,483,648 to 2,147,483,648. Integer data types do not store decimal values; fractional portions are discarded. Note: Integer division by zero does not produce the same result on all operatinc systems. It is recommended that you avoid integer division by zero.
NCHAR(*n*)	Stores a fixed-length character string like CHAR but uses a Unicode nationa character set, where *n* is the maximum number of multibytes to store. Depending on the platform, Unicode characters use either two or four bytes pe character and support all international characters.
NVARCHAR(*n*)	Stores a varying-length character string like VARCHAR but uses a Unicode nationa character set, where *n* is the maximum number of multibytes characters to store. Depending on the platform, Unicode characters use either two or four bytes pei character and can support all international characters.
REAL	Stores a signed, approximate, single-precision, floating-point number.
SMALLINT	Stores a small signed, exact whole number, with a precision of five digits. The range of integers is -32,768 to 32,767. Integer fractional portions are discarded
TINYINT	Stores a very small signed, exact whole number, with a precision of three digits. The range of integers is -128 to 127. Integer data types do not store decimal values; fractional portions are discarded.
VARBINARY(*n*)	Stores varying-length binary data, where *n* is the maximum number of bytes to store. The maximum number of bytes is not required to store each value. If `varbinary(10)` is specified and the binary string uses only five bytes, only five bytes are stored in the column.
VARCHAR(*n*)	Stores a varying-length character string, where *n* is the maximum number o characters to store. The maximum number of characters is required to store each value. If `varchar(10)` is specified and the character string is only five characters long, only five characters are stored in the column.

TIME(*p*)	Stores a time value. A time literal is specified in the format *hh:mm:ss[.nnnnnnnnn]*; a two-digit hour 00 to 23, a two-digit min second 00 to 61 (supports leap seconds), with an optional fraction value. For example, the time 6:30 a.m. is specified as **`06;30:00.`** When supported by a data source, the *p* parameter specifies the seconds precision, which is an optional fraction value that is up to nine digits long.
TIMESTAMP(*p*)	Stores both date and time values. A timestamp literal is specified in the format *yyyy-mm-dd hh:mm:ss[.nnnnnnnnn]*: a four-digit year 0001 to 9999; a two-digit month 01 to 12, a two-digit day 01 to 31, a two-digit hour 00 to 23, a two-digit minute 00 to 59, and a two-digit second 00 to 61(supports leap seconds), with an optional fraction value. For example, the date and time September 24, 1975 6:30 a.m. is specified as **`1975-09-24 06:30:00.`** When supported by a data source, the *p* parameter specifies the seconds precision, which is an optional fraction value that is up to nine digits long.