tcVISION 7

tcSCRIPT 7

Technical documentation

Last review: Sep 25, 23
Code Rev. 1610

1 General

tcSCRIPT is a platform-independent module that allows processing of user data. To provide a great deal of flexibility tcSCRIPT is a REXX-like scripting language, hence allowing the implementation of user-specific processing steps.

tcSCRIPT processes raw data coming

- From tcVISION DBMS-Exits
- From DBMS Log files
- Directly from DBMS (Bulk processing) or partly from 'Unload-files'

and the additional processing of the data up to the creation and execution of the corresponding SQL statements or SQL Loader files.

Three types of scripts are available:

- Data scripts: Processing of tcVISION data
- Process control scripts: Processing control of data scripts
- Control scripts: Supplementary functions, e.g. the start of a data script in a mainframe region or partition of its own.

Standard functional processing flows can be defined using parameter values.
Special exit functions are available to deal with special situations. Individual user-defined processing steps can be implemented at certain points. Input parameters can be processed by the exits and output parameters can be set. Additionally, the exit can also perform DBMS accesses, TCP/IP functions and use a variety of available script functions.

All script types are structured as follows:

Prolog: The prolog is used to define and specify the parameter values for the next processing. The parameter can be created using the menu-based tcVISION Dashboard. Or they can be manually created using the tcVISION Script Editor. The prolog can also be used to write start information into the script log.

Processing: The parameter values coming from the prolog are processed. If necessary, user-defined exits can be called at specified exit points during the processing.

Epilog: Use this part to evaluate the returned information and write them into the script log.

The parameter assignment consists of the stem 'PM_I', the name separated by a period, the operator '=' and the parameter value.

Example: `Function = 'CDC'`

Processing starts with the call

```
CDC_rc = CALLCDC()
```

The variable CDC_rc contains the return code.

The parameter Function controls the main function or the script type.
The parameter value is a character string.
The following values are defined and can only be specified separately:

CDC	Data processing script. Data will be processed according to additional parameter values and exit functions.
SCHEDULE	Process control script. Starts and controls additional scripts according to additional parameter values.
CONTROL	Supplementary script for the execution of functions according to additional parameter values and exit functions.

Example: `Function = 'CDC'`

1.1 Analytic Aids and Error Determination

Data consistency always has to be the primary goal for any data synchronization initiative across platforms. It is very important that all variations from an error-free synchronization are documented and traceable. Such variations can always happen and are rarely predictable.

tcVISION, and especially tcSCRIPT, offers a variety of possibilities to document such variations in different protocol and audit datasets.

It is always recommended to specify these protocol files for any processing script that processes changed data as of stage 1.

These protocol files are:

- GOOD-protocol
  This file contains all records that have been successfully implemented into the target system. The protocol is specified with the parameter `OUTPUT_LOG_NAME`.

- BAD-protocol
  This protocol should always be active. All changed records that have been rejected in the target system are logged into this file. The content of the change record and the error message issued by the target system are part of the protocol. If the flag `OUTPUT_FLAGS_IGN_SQL_ERROR` has been specified, the rejected record becomes part of the BAD-protocol, but the processing continues. The protocol is specified with the parameter `OUTPUT_BAD_NAME`.

- INFO-protocol
  This protocol should always be active. The protocol contains information gathered during processing of the change records. Multiple criteria can be specified for the INFO-protocol. Among others, the following can be specified:
  - Changed data is available, Before and After Images are identical
  - Changed data has been propagated to the target system, but no records were affected
  - Wrong input data was corrected by tcVISION

  The protocol is specified with the parameter `Output_Info_Name.`

2 Parameters for Data Processes and Data Scripts

This chapter deals with parameters and functions of data processes and data scripts with Function = 'CDC'.

The terms data process and data script are equivalent.

In a script definition file each parameter has the prefix “PM_I.”.

2.1 Command Line Parameters

Data scripts are started from a command line.

The first parameter value has no keyword and specifies the fully qualified path and name of the data script that should be executed.

In the remaining part of the command line additional parameters can be specified using keywords. They must be separated by a blank character. Parameter values containing blank characters must be enclosed in double quotes ("), e.g. "keyword=parameter-value".
The parameter described in the following chapter can be specified.

2.2 Extended Command Line Using a File

If data scripts are executed on mainframe systems, no parameter line can be defined (IMS/DLI Batch) or the parameter line is limited in length. For this case it is possible to use a file that contains the desired parameters. Refer to example: IMS/DLI Batch Script.

z/OS:
If a definition like

```
//STDPARM  DD …
```

is part of the job stream, the start parameters will be retrieved from the specified file.

2.2.1AGENT=

The parameter Agent= defines the IP name and port of the controlling Agent.
The format is 'a.a.a.a:xxxxx' wheres a.a.a.a is the IP address and xxxxx specifies the desired port. a.a.a.a may also be specified as DNS name.

Example: `Agent=192.168.0.230:4120`

This parameter can be defined. If it is not specified, the data script runs without a controlling Agent.

If the data script is started from a controlling Agent, this parameter will be automatically set.

2.2.2 TCPIP_NAME=

The parameter `TCPIP_NAME=` defines the optional start parameter value for the local IP stack.
The format of the parameter is a character string used during initialization of the local IP stack.

Example: `TCPIP_NAME=07`

This parameter must be specified if the local IP stack requires a value.

If the data script is started from a controlling Agent, this parameter will be automatically set.

2.2.3 EXTFLG=

The parameter `EXTFLG=` with a value of 1 defines an external data script running outside of the controlling Agent, e.g. in another address space/partition or on another system.
External data scripts cannot be canceled by the Agent, because they run outside of the control of the Agent.
External data scripts will not be directly started by the Agent. They will be started by an external JCL or a control script.

Example: `EXTFLG=1`

This parameter must be specified if the data script is not started by the controlling Agent. The data script signs on to the controlling Agent as an external script.

2.2.4 EVENT_PID=

The parameter `EVENT_PID=` is used to create an instance of an external script and is only necessary if multiple instances of the same script run in parallel. Using the value of this parameter will make the internal names that are based on the script names unique.
The value corresponds to the PID (Process Id) of the starting control script. It is an 8-digit hexadecimal value.
The PID will be extracted by a control script with function GETPID(). This value represents the DECIMAL value of the PID and must be converted into an 8-digit HEX value. See the following example:

Example: `EVENT_ID= D2X(GETPID(),8)`

2.2.5 parametername=

The parameter `parametername=` allows passing and overwriting script variables with the stem of PM_I as described in the following chapters. The expressions must be enclosed in double quotes (") if they contain blank characters. A comma must be used at the end of the parameter to separate multiple parameters. The stem PM_I is a fixed value; 'parametername' must be specified with the parameter name that should be set. The parameter values must be specified WITHOUT double quotes.

Example: `"DLI_SSA.1=SLIEFER  .",`

2.3 Mainframe Runtime Parameters

When a script is started, it reads the runtime parameters. The parameter values and the way they are passed to a script depend on the operating system in use:

z/OS:
The parameters are referred to the JCL with file STDENV and must be defined as follows:

```
parameter=value
```

2.3.1 Deactivate the Abend Handler

This parameter deactivates the Abend Handlers.

z/OS
`RT_NA=1`

2.3.2 Create Separate Output Files

Use this parameter to redirect the output of STDOUT (Standard Output), STDERR (Output Error Messages) and STDTRC (Output Runtime Trace) into separate output files.

z/OS
`RT_MT=1`

2.3.3 Deactivate Runtime Trace

Use this parameter to deactivate the Runtime Trace.

z/OS
`RT_NT=1`

2.3.4 Display Runtime Trace on Console

This parameter redirects the output of the Runtime Trace on the system console.

z/OS
`RT_TC=1`

2.3.5 Activate IP Buffer Trace

This parameter activates the output of the IP buffers into the runtime trace.

z/OS
`RT_TI=1`

2.4 Function

The parameter Function defines the script as data script. The parameter value is the character string 'CDC'.

Example: `Function = 'CDC'`

This parameter is mandatory and must be specified.

2.5 Protocol_Prefix

The parameter Protocol_Prefix replaces the path for the protocol files returned from the Agent.

2.6 Protocol_Timestamp

The parameter Protocol_Timestamp replaces the timestamp used for the protocol file names returned from the Agent.
A specification of '(NONE)' results in no timestamp as part of the file name.

2.7 Trace

The parameter Trace controls the internal trace functions and activates the writing of traces into a protocol file.
The following values are defined and can be specified separated by a blank:

CALLS	Calls to subroutines will be logged.
INFOS	Information from the processing steps will be logged.
ERRORS	Error messages and return codes, etc. will be logged.
SHORT_DUMPS	Storage dumps of small (short) storage areas will be logged.
MEDIUM_DUMPS	Storage dumps of medium large storage areas will be logged.
LARGE_DUMPS	Storage dumps of large storage areas will be logged.
REPOSITORY	Repository accesses will be logged.
TRANSLATION	Storage dumps of routines translating between code pages will be logged.

Example: `Trace = 'CALLS INFOS ERRORS SHORT_DUMPS'`

The parameter is optional. If it is not defined, no tracing will be performed.

2.8 Trace_Seg_Size

The parameter Trace_Seg_Size controls the segmentation of trace files.
The parameter value is a number in MB that specifies the maximum size of a trace file in MB. If the current trace file exceeds this size, it is closed and a new segment is opened.

Example:
```
Trace_Seg_Size = 10
```

This parameter is optional. If it is not defined, segmentation will not be performed.

2.9 Trace_Seg_Compress

If trace files are to be segmented (see parameter Trace_Seg_Size), this parameter controls the compression of file segments.
The parameter value is 0 (no compression of files) or 1 (compression activated).

Example:
```
Trace_Seg_Compress = 1
```

This parameter is optional. If it is not defined, compression will not be performed. This parameter has no effect on z/OS systems.

2.10 Language

The parameter Language specifies the language that should be used for system messages.
The parameter value is a character string.

The following values are defined and can only be specified individually:

D	German
E	English
I	Italian

Example: `Language = 'E'`

This is an optional parameter.
The installation language or the language specified for the controlling Agent will be used as default.

2.11 Local_CCSID

The parameter Local_CCSID defines the locally used CCSID (Coded Character Set Identifier).
The parameter value is numeric.

1252 must be specified for workstations, for mainframe systems the assigned CCSID must be used:

Example Windows: `Local_CCSID = 1252`
Example mainframe: `Local_CCSID = 1148`

This parameter is mandatory and must be specified. The only exception is when the script is being executed under the control of an Agent. In this case, the CCSID of the controlling Agent will be used.

2.12 Script_Class

The parameter Script_Class specifies the script class that should be used. The class has been defined in the corresponding tcVISION Agent.
The script classes can be used to control the execution properties (e.g. serialization).
The parameter value is a character from A to Z.
Default is 'A'.

Example: `Script_Class = 'A'`

2.13 Input_Type

The parameter Input_Type defines the type of data source that should be processed.
The parameter value is a character string.

The following values are defined and can only be specified individually:

FILE The data source is a file in the internal format of tcVISION and has been created by a previously executed data script.

HDFS The data source is a file in the internal tcVISION format and has been created by a previously executed data script in a HDFS. The file will be processed.

PIPE The data source is a pipe in the internal tcVISION format that has been created by a previous data script and should now be processed.

REMOTE The data source is another 'sender' script with Output_Type = 'REMOTE'. The data will be sent by the 'sender' script in an internal format and read by the 'receiver' script.

WEBSPHERE_MQ The data source is a WebSphere MQ queue in the internal format of tcVISION and has been created by another data script.

CDC_POOL The data source is a CDC_POOL on a mainframe system. This option is only supported when the script runs as a subtask of a tcVISION Agent on a mainframe system.

VSAM_LOGSTREAM The data source is a VSAM z/OS logstream.

CICS_LOGSTREAM The data source is a CICS Journal in a z/OS logstream.

DB2_LOGREC The data source is a Db2 active log or a Db2 archive log for z/OS or log files for Db2 LUW for Linux, UNIX, or Windows. For the second case, the index option of the parameter Input_Source_Name can be used to specify multiple files.

ORACLE_LOGREC The data source is the Oracle active log or the Oracle archive log for Oracle databases for Linux, UNIX, and Windows.

POSTGRESQL_LOGREADER
The data source is PostgreSQL active log.

MSSQL_LOGREC The data source is the MS SQL Server active log or the MS SQL Server backup log.

MYSQL_LOGREC The data source is the MySQL server active log.

DATACOM_LOGREC The data source is the file containing the log data for CA Datacom/DB. The log data is processed with the RXXREAD interface. With Input_Source_Type ='READCDC' it is also possible to process log data from an active Datacom CDC database.

IMS_LOGREC The data source is an IMS archive log file. Multiple files can be specified using the index option of the parameter Input_Source_Name.

IMS_LOGREADER The data source is an IMS system. The log files (online log files OLDS and archive log files ALDS) are identified using the DBRC interface and processed in the correct sequence. The name of the IMS system can be specified with the parameter Input_Source_Name.

IDMS_JOURNALREC The data source is an IDMS Journal file. The Journal files may also be transferred in a binary format to a workstation and be processed by a data script. Multiple files can be specified using the index option of the parameter Input_Source_Name.

ADA_LOGREC The data source is an Adabas Dual Protection Log, Sequential Log file, or Adabas LUW PLOG files. Multiple files can be specified using the index option of the parameter Input_Source_Name.
Name of the Adabas Dual Protection Log or Sequential Log file, e.g. '`ADABAS.DATA.DB001.SLOG1`'.
For the real-time access to the set (2 to 8) of the active PLOGs, all PLOG files must be specified.
Instead of the active PLOG files, ADABAS on S390 can also specify the DS name of the DDASSOR1 DDN in the Input_Source_Name parameter, e.g. 'ADA842.DB00001.ASSOR1'.
The script can then use the associated active PLOG DSNs from the ADABAS PPT for processing.

In this case, the parameter ADA_INTERNAL_NUCID = n must also be specified for Adabas Parallel/Cluster Services so that the script can read the correct entry in the PPT.
For **Adabas LUW** the directory containing the PLOG files is specified in the parameter Input_Source_Name.
For UNIX Raw-Devices the path of the Raw-Devices containing the PLG.*-files must be specified. If UNIX Raw-Devices are used, the Raw-Device must contain at least two PLOGs.

BATCH_COMPARE A data source that outputs sorted data will be read in batch and the sorted values as well as the CRC value of the data record will be saved to an output 'snapshot' file. In a subsequent run the same data source will again be processed and compared with the 'snapshot' file from the first run as input file. Output are the corresponding UPDATE, INSERT, and DELETE data records.

BULK_TRANSFER A data source will be read for a bulk data transfer. The data will be – based on the type of data source – read in the most efficient way (block-reads, multi fetch, compressed, etc.) and written in blocks. Usually, the output will be another data script on a remote system. The data should be processed in stage 0 for best throughput and performance. The stage 0 output can also be used for a subsequent 'BATCH_COMPARE' as long as the data will be processed in a sorted order.

SMF_DATA The data sources are z/OS SMF data. The data can be processed from SMF Dumps, SMF VSAM, or SMFLogstreams.

CDC_LOGSTREAM The data source is a logstream file created and maintained by tcVISION (z/OS only).

Example: `Input_Type = ' BULK_TRANSFER'`

This parameter is mandatory and must be specified.

2.14 Input_Source_Name

The parameter Input_Source_Name specifies the name of the data source defined in parameter Input_Type.
The parameter value is a character string.

The following values must be specified depending on Input_Type:

FILE Filename(s) for data in the internal format. It is possible to process multiple files in a directory, for example LUW files. If the filename contains a pattern (* or ?), all files matching the pattern will be processed. The filenames are sorted in ascending order and processing starts with the *first* file in that order. The following keywords can be specified:

WAITFORFILES= Wait for new files when all files in the directory have been processed (YES/NO)

TIMETOWAIT= Number of milliseconds that should be waited until a new scan for new files will be performed (default: 1000ms)

Example:

```
Input_Source_Name = 'C:\Temp\*.s3;WAITFORFILES=YES'
```

HDFS Same as FILE, however the file is written into a HDFS (Hadoop File System). The same token and pattern as with FILE are possible. The file name must start with the path name of the HFDS.

Example:
```
Input_Source_Name = 'hdfs://192.168.0.16:8020/tmp/myfile.txt'
```

PIPE Name of the pipe control file
```
Example:
Input_Source_Name = 'C:\Temp\MyPipe.bin'
```

REMOTE Number of the local TCP/IP ports on which the connection is expected by the remote data script. The format is ':ffff-[tttt]' whereas ffff is the desired port. An ascending port range can be specified with the optional [-tttt]. If a port is already occupied by a server in the same TCP/IP stack, the next port within the range will be checked. If it is free, it will be used. The search will be terminated when the end of the port range has been reached.

Example:
```
Input_Source_Name = ':5005'
Input_Source_Name = ':5005,SSL=Y'
Input_Source_Name = ':5005-5010'
```

If no port or port 0 has been specified, the local IP stack will use a free port. Also refer to the parameter Input_Comm_Token. If SSL=Y is specified (optional, delimited by comma from the port), a listener for using SSL/TLS[1] is opened.

WEBSPHERE_MQ Name of the WebSphere MQ queue. Optionally, an MQ Group ID separated by a colon can be specified. In this case only the corresponding records from that queue will be processed.

CDC_POOL Name of the CDC pool.

VSAM_LOGSTREAM Name of the VSAM logstream, e.g. `'VSAM.LOGSTRM'`.

CICS_LOGSTREAM Name of the CICS logstream, e.g. `'CICSC.O2ATS13C.DFHJ02'`.

CICS_JOURNAL Name of the CICS Journal, e.g. 'CICS2.SYSTEM.LOG.A'.

DB2_LOGREC Name of the Db2 Bootstrap file, e.g. `'DSN610.DB1G.BSDS01'`. For Db2 z/VM 'active log', the cuu of the log minidisk must be specified with keyword CUU=cuu. For Db2 LUW for Linux, UNIX, and Windows use keywords 'LOGPATH=' and 'ARCHIVEPATH=' to specify the directories containing the active log files and archived log files.

ORACLE_LOGREC Direct access to log files:
The keywords 'LOGPATH=' and 'ARCHIVEPATH=' specify the directories in which the active log files and the archived log files can be found.
Access using the LOGMINER interface:
The keywords 'DSN=', 'UID=', and 'PWD=' define the connection to the Oracle database.
The keyword 'PREFETCH=' n sets the amount of records fetched at once, whereas n can be a value between 1 and 1000. This parameter only works for an Oracle driver of version 19 or higher.

MSSQL_LOGREC The keywords 'LOGPATH=' and 'ARCHIVEPATH=' specify the directories in which the active log files and the archived log files can be found.

MYSQL_LOGREADER
The access is either performed using the libmysql API and 'mysqlbinlog', or with a direct access to the log data. IP address and port, username, and password must be specified. The password may be specified in encrypted format. The parameter 'log-bin_basename' is described in chapter 8.15.1MySQL Log Processing.
Example:
```
'server=192.168.8.15:3310;uid=postgres;pwd=enc123546789sdf;log_
bin_basename=MYSQLBINLOG;'
```

POSTGRESQL_LOGREADER

[1] This product includes cryptographic software written by Eric A. Young (eay@cryptsoft.com). This product includes software written by Tim J. Hudson (tjh@cryptsoft.com).

	PostgreSQL is accessed using the libpq API and 'logical decoding'. IP address, database name, user name, and password must be specified. The password can be encrypted. Example: `'host=192.168.8.15;dbname=postgres;version=9.4.4;user=postgres; password=enc123546789sdf'`
IMS_LOGREC	Name of the IMS archive log file, e.g. `'IMS.SLDSP.IVP1.D04041.T1354517.V11'`.
IMS_LOGREADER	Optionally, the name of the IMS system with keyword SSID=, e.g. `'SSID=IVP1'.` Optionally, the name of the active WADS may be specified (refer to #8.4.1.IMS Logreader) using the keyword WADS1=, e.g. `'SSID=IVP1;WADS1=DFSD10.WADS0;'` For Data Sharing Logreader, the name of the DRA connection table with DRA= and the PSB name (only one IO-PCB is used) with PSB= can be specified additionally, e.g. `'SSID=IVP1;WADS1=DFSD10.WADS0;DRA=TV;PSB=TCVHBPSB;'`
IDMS_JOURNALREC	Name of the IDMS Journal file. The file can either be a tape file or a disk file. e.g. `'FALOGDB.G0000T001;TAPE_CUU=0580;VOL_SERNR=000004;'`.
ADA_LOGREC	Name of the Adabas Dual Protection Log or Sequential Log file, e.g. `'ADABAS.DATA.DB001.SLOG1'`. All PLOG files must be specified for real-time access to the set (2 to 8) of the active PLOGs. For **Adabas LUW** the directory containing the PLOG files must be specified. For UNIX Raw-Devices, the path of the Raw-Devices containing the PLG.*-files must be specified (currently, multiple Raw-Devices are not supported). If UNIX raw devices are used, the raw device must contain at least two PLOGs.
DATACOM_LOGREC	Depending on parameter Input_Source_Type:
READRXX	Name and unit of log data file, i.e: `'RXX;TAPE_CUU=0183;'`.
READCDC	URT=urt;DBID=dbid;CTLID=A;MUF=mufname, e.g.: `'URT=DBURT999;DBID=2009;CTLID=A;MUF=MUFBOS'.`
BATCH_COMPARE and BULK_TRANSFER	Depending on parameter Input_Source_Type:
FILE	Name of file.
HDFS	Name of file in URL format (e.g.: hdfs://192.168.0.16:8020/tmp/file.txt) (only applies to BULK_TRANSFER).
ADABAS	If the data source is a file in the Adabas-ADAULD compressed format, specify the file name. For BULK_TRANSFER a file in Adabas-ADASAV or Adabas-ADABCK format can be specified. One or more Adabas files can be selected from the database backup with ADABAS_FINR. These files can also be processed on workstations. In this case the files should be transferred to the workstation using FTP with options 'binary' and 'quote site rdw'.
DRDA	DRDA-Connect-string, example: `'HOST=192.168.0.230;PORT=446;DATABASE=S390LOC;UID=USER;PWD=PASSWD;TRACE=N;CTB=Y;PLAN=TCEXPRESS;PACKAGE=TCELF000;CONSTOK=6841446351614450;EOS=N;DEFSYSTAB=Y;UII=Y'`
ODBC	ODBC connect string, e.g. `'DSN=MyDSN;'`
EXASOL	EXASOL connect string, e.g. `'SERVER=address:port;UID=userid;PWD=Password'`
MYSQL	MYSQL connect string, e.g. `'server=127.0.0.1:3310;uid=name;pwd=pwd;'`
POSTGRESQL	POSTGRESQL connect string, e.g. `'host=127.0.0.1;port=5432;dbname=postgres;user=name; password=pwd;'`

ORACLE Oracle OCI connect string, e.g.
```
'DSN=ServiceName;UID=userid;PWD=Password'
```

IDMS IDMS parameter with the following keywords:

SUBSCHEMA=	Name of subschema
RECORD=	Name of record
REC_LEN=	Length of record
AREA=	Area name of record
INDEX=	Name of the index that should be used
INDEX_AREA=	Area name of index
READY_ALL	A READY command will be performed for all IDMS areas.
NO_READY	No READY command will be performed.

Example:
```
'SUBSCHEMA=GINA01R;RECORD=RINA-KLIENT-
DB;REC_LEN=400;AREA=AINAB02;INDEX=INA-NDX-
NMADRES;INDEX_AREA=AINAX05;NO_READY'
```

IDMS_BACKUP Name of an IDMS backup file

DATACOM Datacom parameter with the following keywords:

URT=	Name of the URT for the data access
TABLE=	3-digit file name
KEY=	Name of the key to be used (for GSETP)
ELEMENTS=	Element list for the access
REC_LEN=	Record length

Example:
```
' URT=DBURT021; TABLE=LIM;KEY=LIMNR;ELEMENTS=LIM
;REC_LEN=750 '
```

JDBC JDBC connection string with the following keywords:

CLASS=	Name of the Java driver class
UID=	Name of the user to log in
PWD=	User password
DRIVER=	Url of the driver, usually in the form jdbc://...
SUBTYPE=	JDBC subtype. If one exists for the JDBC driver, please use it.

For using the JDBC output target it is mandatory that the .jar for the driver is added in the classpath. The tcvj.jar must either be in the classpath or in the same folder as the tcSCRIPT. A JAVA of version 7 or newer is required.

DCD_UNLOAD Name of the file that contains the Datacom UNLOAD image (e.g. a Datacom EXTRACT).

DB2 Db2 requires the following parameters:

DATABASE=	Name of the Db2 database
UID=	User ID. This parameter is only required if an RRSAF connection to Db2 is used.
PWD=	Password. This parameter is only required if an RRSAF connection to Db2 is used.

Example:
```
'DATABASE=DB8G'
 'DATABASE=XYZG;UID=userid;PWD=password'
```

DB2_ICOPY Fully qualified name of the Db2 image copy file

DB2M_UNLOAD
Fully qualified name of the Db2 unload file. The file name is the key for the identification of the input table definition in the repository in the form *database.creator.table*.

SMF_DATA For SMF dump data, the file names containing the SMF dump data are specified. These files can also be processed on workstations. In this case the files should be transferred to the workstation using FTP with the options 'binary' and 'quote site rdw'.

For SMF VSAM data, the VSAM file names has to be specified (SYS1.MAN1 etc.).

For SMF logstreams, the name of the logstream has to be specified (e.g. IFASMF.S0W1.DEFAULT).

CDC_LOGSTREAM
Fully specified name of the logstream files. This is only available for z/OS.
Example:
```
Input_Source_Name = 'TCVISION.LOGSTR'
```

This parameter has to be specified for Input_Type = 'FILE', 'PIPE', 'REMOTE', 'WEBSPHERE_MQ', 'CDC_POOL', 'CICS_LOGSTREAM', 'VSAM_LOGSTREAM', 'CICS_JOURNAL', 'DB2_LOGREC', 'ORACLE_LOGREC', 'MSSQL_LOGREC', 'MYSQL_LOGREC', 'DATACOM_LOGREC', 'IMS_LOGREC', 'IMS_LOGREADER', 'ADA_LOGREC', 'BATCH_COMPARE', 'SMF_DATA', CDC_LOGSTREAM', 'POSTGRESQL_LOGREADER', or 'BULK_TRANSFER'.

2.14.1 Multiple Input Files

Some data sources support the use of multiple input files with the same structure (refer to parameter Input_Type). The specification of multiple input files with the index option is as follows:

Input_Source_Name.*index* = '...' whereas '*index*' is a number in the range of 1 to 500.

Example: Input_Source_Name = 'first.dataset'
```
Input_Source_Name.1 = 'second.dataset'
Input_Source_Name.2 = 'third.dataset'
```

2.15 SSL_KEYFILE

The parameter SSL_KEYFILE defines the name of the key file for the use of the SSL/TLS listener. The parameter value is a character string.

2.16 Input_Source_Move_To_On_RC0

The parameter Input_Source_Move_To_On_RC0 defines a directory to which the input file should be moved depending on the return code.

Also refer to parameter Input_(INPUT_Flags_MOVE_INPUT_RC0).
The parameter value is a character string.

Example:
```
Input_Source_Move_To_On_RC0 = 'z:\LUWs without errors'
```

The parameter is optional.

2.17 Input_Source_Move_To_On_RCX

The parameter Input_Source_Move_To_On_RCX defines a directory to which the input file should be moved if the return code is greater than or equal 8.
Also refer to parameter Input_Flags (INPUT_FLAGS_MOVE_INPUT_RCX).
The parameter value is a character string.

Example:
```
Input_Source_Move_To_On_RCX = 'z:\LUWs with errors'
```

The parameter is optional.

2.18 Input_Flags

The parameter Input_Flags defines options for the input data.
The parameter value is numeric.

The following values are defined and can be specified individually and in combination:

INPUT_FLAGS_DELET_INPUT_RC0:
(hex value 0x00000001)
For Input_Type = 'FILE'; If the return code is 0, the input file will be deleted.
INPUT_FLAGS_MOVE_INPUT_RC0:
(hex value 0x00000002)
For Input_Type = 'FILE'; If the return code is 0, the input file will be moved to the directory defined with parameter Input_Source_Move_To.
INPUT_FLAGS_MOVE_INPUT_RCX:
(hex value 0x00000004)
For Input_Type = 'FILE'; If the return code is greater than or equal to 8, the input file will be moved to the directory defined with parameter Input_Source_Move_To.
INPUT_FLAGS_CONTINUE_LISTEN:
(hex value 0x00000008)
For Input_Type = 'REMOTE'; After every session with a remote "sender" script that has been successfully closed, another connection attempt of the "sender" script will be waited for.
INPUT_FLAGS_IGN_DB2_NON_DC:
(hex value 0x00000010)
For Input_Type = ' DB2_LOGREC' images without DATA CAPTURE flag that belong to selected tables will be processed.
INPUT_FLAGS_DATA_TO_CRC:
(hex value 0x00000020)

For Input_Type = ' BATCH_COMPARE' only the key values will be processed and a CRC checksum will be calculated for the data. Using the integrated SORT, the data to be sorted can be limited to the key values. This option is available for IMS/DLI, IDMS, and sequential files.

INPUT_FLAGS_DATA_SORT:
(hex value 0x00000040)

For Input_Type = ' BATCH_COMPARE' using this option the files to be processed can be sorted using the internal mainframe DFSORT interface before the actual batch compare takes place.
This option can be used for IMS/DLI HDAM or unsorted files (sequential, VSAM ESDS, etc.).

INPUT_FLAGS_BC_SKIP_DUP_KEY:
(hex value 0x00000080)

For Input_Type = ' BATCH_COMPARE' duplicate key values can be ignored (skipped) in the input file when this option is in use. All data records with a key that has already been processed will be skipped. Optionally, these records can be written to the information protocol file.

INPUT_FLAGS_NRT_WAIT_DB2:
(hex value 0x00000100)

For Input_Source_Type = 'IFI_306' (Processing "real-time") this option specifies whether tcVISION should wait for a Db2 during initialization or during processing if Db2 is not available. If this option is not specified, processing is terminated.

INPUT_FLAGS_NRT_CONTINUE_EOF:
(hex value 0x00000200)

For Input_Source_Type = 'IFI_306' (Processing "real-time") this option specifies whether the processing of the Db2 active log should be terminated when the end-of-file condition has been reached (flag is NOT set) or whether processing should continue (flag has been set). For the latter, tcVISION will wait until new Db2 activities emerge.

For Input_Source_Type = 'DB2L_API' this option specifies whether the processing should be terminated (flag not set) or continued (flag set) when reaching end-of-file or when the highest LSN (Log Sequence Number) has been reached. In this case the script is waiting for new log data.

For Input_Type = 'CICS_LOGSTREAM' or 'VSAM_LOGSTREAM' this option specifies whether the processing should be terminated (flag not set) or continued (flag set) when reaching end-of-file during the processing of the logstream. In this case the script is waiting for new logstream data.

For Input_Source_Type = 'ORA_LOGM' this option specifies whether the processing should be terminated (flag not set) or continued (flag set) when reaching end-of-file or when the highest SCN has been reached. In this case the script is waiting for new log data.

For Input_Source_Type = 'MYSQL_LOGREADER' this option specifies whether processing should stop at an end-of-file (EOF) condition (flag NOT set) or continue (flag is set). For the latter, tcVISION will wait until new records have been written into the log data.

For Input_Source_Type = 'POSTGRESQL_LOGREADER' this option specifies whether processing should stop at an end-of-file (EOF) condition (flag NOT set) or continue (flag is set). For the latter, tcVISION will wait until new records have been written into the log data.

For Input_Type = 'WEBSPHERE_MQ' this option specifies whether processing should stop (flag NOT set) at an end-of-queue condition or continue (flag is set). For the latter, tcVISION will wait until new records have been written into the queue.

For Input_Source_Type = 'READCDC' this option specifies whether processing should be terminated (flag NOT set) when an EOF condition is encountered (reaching the most current entry in the TSN table) or whether processing should continue (flag is set). In this case the script waits for new data records in the TSN table.

For Input_Source_Type = 'PIPE' this option specifies whether processing should be terminated (flag NOT set) when an EOF condition is encountered (reaching the most current entry in the Pipe) or whether processing should continue (flag is set). In this case the script waits for new data records in the pipe.

For Input_Source_Type = 'ADA_LOGREC' and specification of all active PLOGs in parameter Input_Source_Name this option specifies whether the processing should be terminated (flag NOT set) when an EOF condition is encountered (reaching the most current entry in the PLOG) or whether processing should continue (flag is set). In this case the script waits for new data records in the active PLOG file.

For Input_Type = 'SMF_DATA' and specification of all active VSAM files or the active logstream this option specifies whether the processing should be terminated (FLAG NOT set) when an EOF condition is encountered (reaching the most current SMF entry) or whether processing should continue (flag is set). In this case the script waits for new SMF data records.

For Input_Type = 'CDC_LOGSTREAM' and specification of the active logstream this option specifies whether the processing should be terminated (flag NOT set) when an EOF condition is encountered (reaching the most current logstream entry) or whether processing should continue (flag is set). In this case the script waits for new logstream data records.

For Input_Type = 'IMS_LOGREADER' this option specifies whether the processing should be terminated (flag NOT set) when an EOF condition (reaching the most current entry in the online log) is encountered or whether processing should continue (flag is set). In this case the script waits for new logstream data records.

INPUT_FLAGS_MQ_BROWSE:
(hex value 0x00000400)

For Input_Source_Type = 'WEBSPHERE_MQ' this option specifies that an input queue is processed in BROWSE mode and the records should not be deleted in the queue.

INPUT_FLAGS_DB2DATASHARING:
(hex value 0x00000800)

For Db2 z/OS this option specifies that the input RBA/LRSN-fields contains an LRSN.

INPUT_FLAGS_IMSDATASHARING:
(hex value 0x00000800)

For IMS this option specifies that the IMS LogReader processes a member of a Data Sharing Group.

INPUT_FLAGS_DB2_I306_CDCONLY:
(hex value 0x00001000)

For Db2 z/OS: this option specifies that the processing of IFI 306 input only applies to log records that have the Change Data Capture flag. This increases the processing performance, **but no** CREATE-DROP-Table commands are recognized and maintained in the history.
This option is activated automatically when processing data from LOB/XML tablespaces, because the IFI 306 interface discards the log data otherwise.

INPUT_FLAGS_DB2_I306_DECOMP:
(hex value 0x00002000)

For Db2 z/OS this option specifies that the processing of IFI 306 log records containing compressed data are decompressed using the IFI 306 interface. The script-internal decompression is not used.

INPUT_FLAGS_DB2_SYSEVT_LOG:
(hex value 0x00004000)

For Db2 z/OS this option specifies that log records with information related to
- new entries in SYSIBM.SYSCOPY,
- ALTER TABLE ADD COLUMN...
- REORG/RELOAD with LOG(YES) with changed compression dictionaries compared to the current dictionary data are processed in the repository. Also refer to 'CDC_ExitDBControl'.

INPUT_FLAGS_PM_ACTIVE:
(hex value 0x00008000)

With this option the elapsed time consumed for certain processing steps can be determined. The corresponding timings can be evaluated during and after a script execution using script variables. Refer to chapter 'Performance Monitoring'.

INPUT_FLAGS_DB2_ICOPY_ALL_P:
(Hex-Wert 0x00020000)

With this option all partitions of a tablespace are processed for Db2 imagecopy processing.

INPUT_FLAGS_DB2_IGN_CCSID:
(Hex-Wert 0x00040000)

With this option for a Db2-S390 bulk processing the CCSID of the data fields is not taken from the SQLVAR metadata but from the repository.

INPUT_FLAGS_DB2LUW_FULL_LOG:
(Hex-Wert 0x00080000)

With this option long field values are also processed for Db2 LUW processing.

INPUT_FLAGS_POSTGRESQL_DROP_SLOT:
(Hex-Wert 0x00100000)

This option deletes the replication slot after PostgreSQL CDC processing.

INPUT_FLAGS_POSTGRESQL_NO_COPY:
(Hex-Wert 0x00200000)

This option disables the streaming copy mode during PostgreSQL bulk processing.

INPUT_FLAGS_DB2_I306_PRESELECT:
(Hex-Wert 0x00400000)

For Db2 z/OS, this option determines that the log data to be processed is preselected via the IFI 306 WQLS selection.
This option is activated automatically for processing data from LOB/XML tablespaces, otherwise the IFI 306 interface discards this log data.

INPUT_FLAGS_DB2_I306_LUW_FILTER:

(Hex-Wert 0x00800000)
For Db2 z/OS this option determines that the BEGIN and END transaction log data is filtered out for 'empty' LUWs .
This option significantly reduces the data volume if only a small portion of the extensive log data is selected for further processing.
This filtering requires a small additional load on the host.

Example:
```
Input_Flags = INPUT_FLAGS_MOVE_INPUT_RC0
```

This parameter is optional.

2.19 Input_Comm_Token

The parameter Input_Comm_Token defines a connection token that will register the local receiver script with Input_Type = 'REMOTE' at the local Agent and therefore allows a remote sender script to establish a connection using this token.
The parameter value is a character string.

Example:
```
Input_Comm_Token = 'ReceiverScript1'
```

This is an optional parameter.

2.20 Input_Source_Type

The parameter Input_Source_Type defines the data source subtype if Input_Type = 'BATCH_COMPARE', 'BULK_TRANSFER', 'DATACOM_LOGREC' or 'DB2_LOGREC'has been specified.
The parameter value is a character string.

The following values are defined and can only be used individually:

Input_Type = 'BATCH_COMPARE', 'BULK_TRANSFER':

USEREXIT	Data is provided by exit ’CDC_ExitUserRecord’.
FILE	Data is read from a file.
DLI_BATCH	Data is from a DLI database in batch mode.
IDMS	Data is read from an IDMS database in batch mode.
IDMS_SR23	SR2/SR3 data is physically read from an IDMS database.
IDMS_BACKUP IDMS	Data is read from an IDMS backup file.
DATACOM	Data is read from a Datacom database in batch mode.
DRDA	Data is read from a DRDA database (Db2).
ODBC	Data is read from an ODBC database.
EXASOL	Data is read from an EXASOL database.
ORACLE	Data is read from an Oracle database.
MYSQL	Data is read from a MYSQL database.
POSTGRESQL	Data is read from a PostgreSQL database.
ADABAS	Data is read from an Adabas database.
DB2	Data is read from a Db2 direct interface.
DB2_ICOPY	Data is read from a Db2 image copy file. This value is only valid for Input_Type = 'BULK_TRANSFER'.

DB2M_UNLOAD
: Data is read from a Db2 image unload file. Such an unload is created with the Db2 program DSNUPROC and the NOPAD option. This value is only valid for Input_Type = 'BULK_TRANSFER'.

CDC_FILE
: Data is read from a file in stage 0 that is in the internal tcVISION format. This option allows a 'BATCH_COMPARE' with the data of a previous 'BULK_TRANSFER' script.

Input_Type = 'DB2_LOGREC':

DSNJSLR
: Data is read from Db2 archive log files (Db2 must not be started).

IFI_306
: Data is read from a Db2 active log (Db2 must be started). For Db2 z/VM the 'active log' from the log minidisk will be processed.

DB2M_FILE
: Db2 z/OS log is read from a sequential file. This file can either be a Db2 archive log file on a mainframe system or the file can be sent via FTP (binary) to another system and can be processed there by a data script. Multiple files can be specified with the index option of parameter Input_Source_Name.

DB2L_API
: Log files for Db2 LUW for Linux, UNIX, and Windows are processed using the API 'ReadLog'. To connect to the source database, specify the parameter Input_Source_Name and the keywords 'DATABASE=', 'UID=, and 'PWD='. UID= specifies the user id and ‘PWD=’ the corresponding password.

Input_Type = 'DATACOM_LOGREC':

READRXX
: Data is processed by the READRXX interface.

READCDC
: Data is processed of CDC tables TSN and MNT.

Input_type = 'ADABAS_LOGREC'

ADABAS
: Data is processed with the CLOG interface (Command Log).

ADAPLOG
: Data is processed with the PLOG interface (Protection Log).

Input_Type = 'ORACLE_LOGREC':

ORA_LOGM
: Oracle databases for Linux, UNIX, and Windows log files are processed using the LOGMINER interface. The connection to the desired Oracle database can be established by specifying the parameter Input_Source_Name and an ORACLE OCI-connect-string of: `'DSN=ServiceName;UID=userid;PWD=Password'`. No path specifications to the log files are required, because LOGMINER obtains this information directly from the database.

Input_Type = 'MSSQL_LOGREC':

MSSQL_FILE MS-SQL Server log data is read from the corresponding directories. Two directories can be specified with the parameter Input_Source_Name and the keywords 'LOGPATH=' and 'ARCHIVEPATH='. LOGPATH refers to the directory containing the 'active log' for a running or stopped MS SQL Server. ARCHIVEPATH refers to the related 'Backup_Log' directory.

MSSQL_CDC MS SQL Server CDC data is read (SQL Server 2008 R2 and higher). The connection to the SQL Server is defined with the parameter Input_Source_Name.

For Input_Type = 'BATCH_COMPARE', 'BULK_TRANSFER', 'ORACLE_LOGREC', 'MSSQL_LOGREC' or 'DB2_LOGREC' this parameter is mandatory and must be specified.

For Input_Type = 'DATACOM_LOGREC' this parameter is optional.

2.21 Input_Source_Query

The parameter Input_Source_Query defines the SQL query if Input_Type = 'BATCH_COMPARE' or 'BULK_TRANSFER' and Input_Source_Type = 'DRDA', 'DB2','ODBC', 'EXASOL', 'ORACLE', 'MYSQL' or 'POSTGRESQL',has been defined.
The parameter value is a character string.

Example:
```
Input_Source_Query =
'SELECT * FROM DEMO.ARTICLE'
```

This parameter is mandatory for Input_Type = 'BATCH_COMPARE', 'BULK_TRANSFER' and Input_Source_Type = 'DRDA', 'DB2', 'ODBC', 'EXASOL', 'ORACLE', 'MYSQL' or 'POSTGRESQL'.

2.22 Input_Source_Order_by

For the parameter Input_Source_Type = 'DRDA', 'ODBC', 'EXASOL', 'MYSQL', 'POSTGRESQL' or 'ORACLE':

The parameter Input_Source_Order_by defines the field names and sequences of an SQL query that represent the ascending and unique sorting values. The fields must be separated by commas.
The parameter value is a character string.

Example:
```
Input_Order_by = 'Number, Name'
```

This parameter is mandatory for Input_Type = 'BATCH_COMPARE' and must be specified.
This parameter is optional for Input_Type = 'BULK_TRANSFER' if the output file should be used by a subsequent 'BATCH_COMPARE' script.

For Input_Source_Type = 'DATACOM' and Input_Type = 'BATCH_COMPARE':
The parameter Input_Source_Order_by defines positions and lengths of fields that are the ascending and unique sort values. The field names must be separated by commas.
The parameter value is a character string.

Example:

```
Input_Order_by = '0(1),10(1),22(2),34(2)'
```

This parameter is mandatory for Input_Type = 'BATCH_COMPARE'.

2.23 Input_Source_Sort_CCSID

The parameter Input_Source_Sort_CCSID defines the CCSID that has been used for the sort on the source system for Input_Type = 'BATCH_COMPARE'. This parameter must be specified if the local CCSID would result in a different sorting order and the key values would no longer be in ascending order.

Example:

```
Input_Source_Sort_CCSID = 1148
```

For example: If DRDA Db2 data with a CCSID of 1184 from a mainframe system are processed on a PC, this parameter and the value of 1148 (as an example) should be used to ensure that the local process will process the key values with the same CCSID.

2.24 Input_SnapShot

The parameter Input_SnapShot defines the name of an input snapshot file that should be used for comparison with the data source. The input snapshot file has been previously created as an output snapshot file.
The parameter value is a character string.

This parameter is optional for Input_Type = 'BATCH_COMPARE'

2.25 Output_SnapShot

The parameter Output_SnapShot defines the name of an output snapshot file. This will later be used as an input snapshot file.
The parameter value is a character string.

This parameter is optional for Input_Type = 'BATCH_COMPARE'

2.26 Input_RecLen

The parameter Input_RecLen defines the length of data records in the input file if Input_Source. = 'FILE' has been specified and the length cannot be automatically determined by a script (e.g. for a PC file).

Special values:
A value of -1 determines the record length by a CR and LF sequence. The CR and LF sequence itself is not processed.
A value of -2 determines the record length by a CR or LF sequence. The CR or LF sequence itself is not processed.

The parameter value is numeric.

This parameter is optional for Input_Type = 'BATCH_COMPARE', 'BULK_TRANSFER'

2.27 Input_KeyPos

The parameter Input_KeyPos defines the starting position of a field that contains unique sort values. It applies to Input_Source. = 'FILE' and when the starting position cannot be automatically determined by a script (e.g. for a PC file).
The parameter value is numeric.

This parameter is optional for Input_Type = 'BATCH_COMPARE'

2.28 Input_KeyLen

The parameter Input_KeyLen defines the length of a field that contains unique sort values. It applies to Input_Source. = 'FILE' and when the starting position cannot be automatically determined by a script (e.g. for a PC file).
The parameter value is numeric.

This parameter is optional for Input_Type = 'BATCH_COMPARE'

2.29 Input_StartFrom

The parameter Input_StartFrom defines a value that should be used as a start key value for a bulk or batch compare operation. Depending on the type of input source, the value is used as a starting key.

The parameter value is a character string. All characters in the string are taken as they are with the exception of the character combination '/x'. All characters following '/x' are treated as hexadecimal values and their binary value is used. The hexadecimal interpretation ends at the end of the character string or at the next occurrence of character '/'.

Example:

```
Input_StartFrom = "Alpha/x01234C"
```

This parameter can be specified for Input_Type = 'BATCH_COMPARE' and 'BULK_TRANSFER' if Input_Source_Type = 'ADABAS', 'DATACOM', 'DLI_BATCH', 'IDMS', and 'FILE' (VSAM KSDS).

2.30 Input_EndAt

The parameter Input_EndAt specifies the ending key value for a bulk or batch compare operation. A record that exactly matches this key value does not become part of the number of hits.

The parameter value is a character string. All characters in the string are taken as they are with the exception of the character combination '/x'. All characters following '/x' are treated as hexadecimal values and their binary value is used. The hexadecimal interpretation ends at the end of the character string or at the next occurrence of character '/'.

Example:
```
Input_EndAt = "Zulu/xFFFF"
```

This parameter can be specified for Input_Type = 'BATCH_COMPARE' and 'BULK_TRANSFER' if Input_Source_Type = 'ADABAS', 'DATACOM', 'DLI_BATCH', 'IDMS', and 'FILE' (VSAM KSDS).

2.31 Input_SkipCount

The parameter Input_SkipCount specifies the number of records that should be skipped after the start of a bulk transfer (with or without key value).

The parameter value is numeric.

Example:
```
Input_SkipCount = 1000000
```

This parameter can be specified for Input_Type = 'BULK_TRANSFER' if Input_Source_Type = 'ADABAS', 'DATACOM', 'DCD_UNLOAD', 'DLI_BATCH', 'IDMS', and 'FILE'.

2.32 Input_Pipe_Min_Wait

The parameter Input_Pipe_Min_Wait defines the minimum number of milliseconds that should be waited before the pipe is checked for new input data.

The parameter is a numeric value.
Minimal value: 2
Maximal value: 4000
Default value: 10

If no new data is available in the pipe, the specified number of milliseconds is waited for. For every unsuccessful attempt the specified number of milliseconds is increased by the original value until the value in variable Input_Pipe_Max_Wait is reached.

Example:
```
Input_Pipe_Min_Wait = 500
```

2.33 Input_Pipe_Max_Wait

The parameter Input_Pipe_Max_Wait defines the maximum number of milliseconds that should be waited before the pipe is checked for new input data.

The parameter is a numeric value.
Minimal value: 2
Maximal value: 8000
Default value: 20

Example:
```
Input_Pipe_Max_Wait = 500
```

2.34 Input_DatabaseName

The parameter Input_DatabaseName specifies the database that should be used.

Example:
```
Input_DatabaseName = 'SAMPLE'
```

This parameter must be specified for Input_Source_Type = 'DB2V_FILE','DB2L_API','DB2M_FILE','MSSQL_FILE', 'MSSQL_CDC'.

2.35 DLI_PCB

The parameter DLI_PCB defines the offset to be used in a PCB list during an IMS batch processing. The first PCB in the list has an offset of 0.
The parameter value is numeric.

Example:
```
DLI_PCB = 1
```

This parameter is optional for Input_Source_Type = 'DLI_BATCH'

2.36 DLI_SSA

The parameter DLI_SSA defines the SSAs to be used with a DLI batch processing.
Up to 15 SSAs with an index from 1 to 15 can be defined.
If this parameter has not been specified, an unqualified access is performed and all segments defined in the PSB will be processed.
If the parameter has been specified, every entry must be a valid SSA.
The parameter value is a character string.

Example:
```
DLI_SSA.1 = 'PARTROOT .'
```

This parameter is optional for Input_Source_Type = 'DLI_BATCH'

2.37 DLI_DBD

The parameter DLI_DBD defines the databases and segments that are to be processed when reading IMS log data. Multiple databases and segments separated by commas can be specified. During the pre-processing (output stage 1) only the log records relevant to the specified databases and segments will be retrieved from the log blocks.
The format is DBD_Name(Segment_Name,...),....
DBD_Name corresponds to the parameter 'NAME' used at the DBD-Generation.
Segment_Name corresponds to parameter 'NAME' used at the SEGM-Generation. Multiple DBD names can be specified, separated by comma. For every DBD name multiple segment names can be specified in brackets and separated by commas. Segment name can also be *, indicating that all segments of the database should be selected.

Example:

```
DLI_DBD = 'XPIMSDB1(ARTIKEL), XPIMS1(*)'
```

This parameter is specified for Input_Type = 'IMS_LOGREC', 'IMS_LOGREADER', or if the input data comes from a remote data script that outputs the data in the internal format of stage 0.

2.38 IDMS_START_JSNR

The parameter IDMS_START_JSNR defines the desired start JSNR when reading IDMS journal data.
The parameter value is a decimal string of max. 20 bytes.

Example:

```
IDMS_START_JSNR = '138234'
```

This parameter can be specified if Input_Source_Type = 'IDMS_JOURNALREC' or if the input data comes from a remote data script which outputs the data in internal format of stage 0.

2.39 IDMS_END_JSNR

The parameter IDMS_END_JSNR defines the desired last JSNR to be processed when reading IDMS journal data.
The parameter value is a decimal string of max. 20 bytes.

Example:

```
IDMS_END_JSNR = '138934'
```

This parameter can be specified if Input_Source_Type = 'IDMS_JOURNALREC' or if the input data comes from a remote data script which outputs the data in internal format of stage 0.

2.40 IDMS_AREA_RECID

The parameter IDMS_AREA_RECID defines the areas and record IDs to be processed when reading the IDMS Journal files. Multiple areas and record IDs can be specified, separated by a comma.
During the pre-processing (output stage 1) only the journal records that are relevant to the specified areas and record IDs will be retrieved from the journal blocks.
The format is Area_Name(RecordId,...),....
Area_Name corresponds to parameter 'NAME' of ADD AREA NAME IS *name*. Record ID corresponds to parameter ADD RECORD ... RECORD ID IS *recid*. Multiple areas (separated by comma) can be specified. For each area multiple record IDs (separated by comma) can be specified in brackets. RecordIDs can also be specified as ranges.

Example:

```
IDMS_AREA_RECID = 'AREA1(1,2,3,4),AREA2(1-6,1201)'
```

Use this parameter for Input_Source_Type = 'IDMS_JOURNALREC' or if the input data comes from a remote data script that outputs the data in the internal format of stage 0.

2.41 IDMS_KEY_CONNECT_TYPE

The parameter IDMS_CONNECT_TYPE defines the access type for an SQL database.
The SQL database will be used to read the corresponding key fields of the OWNER data records for a particular member data record when the IDMS journal/backup records contain OWNER-DBKEY-values.

The following values are allowed:

DRDA	DRDA database
ODBC	ODBC database
EXASOL	EXASOL database
ORACLE	Oracle OCI database
POSTGRESQL	PostgreSQL database

This parameter must be defined when IDMS journal/backup record member data records and their corresponding Owner data records must be processed from stage 0 to stage >=1.

2.42 IDMS_KEY_CONNECT_NAME

The parameter IDMS_CONNECT_NAME defines the connect string necessary to access the SQL database specified with IDMS_CONNECT_TYPE.

Examples:

DRDA-Connect-String:

```
'HOST=192.168.0.230;PORT=446;DATABASE=S390LOC
;UID=USER;PWD=PASSWD;TRACE=N;CTB=Y;PLAN=TCEXPRESS;PACKAGE=TCELF
000;CONSTOK=6841446351614450;EOS=N;DEFSYSTAB=Y;UII=Y'
```

ODBC-Connect-String: `'DSN=MyDSN;'`

ORACLE OCI-Connect-String:

```
'DSN=ServiceName;UID=userid;PWD=Password'
```

EXASOL Connect-String:

```
'SERVER=address:port;UID=userid;PWD=Password'
```

POSTGRESQL Connect-String:

```
'host=127.0.0.1;port=5432;dbname=postgres;
user=name;password=pwd;'
```

This parameter must be defined if IDMS journal record member data records and their corresponding Owner data records must be processed from stage 0 to stage >=1.

2.43 IDMS_HSLDATE_DEFINITION_FILE

If the product HSLDate is in use, this parameter defines the name of the definition file with the date transformations. The parameter is optional. The following example shows the supported format:

```
*
* ------------------------------------------------------
*
*    SITE: EDUCATIONAL TESTING SERVICES
*
*    PROJECT 2000 HSLDATE RECORD/FIELD ZAP NUMBER:  41
*
*    GENERATED WITH THE FOLLOWING PARAMETERS:
*
*            DATE CLASS:  41
*           RECORD NAME:  APCSYR
*             AREA NAME:  APP-CUST-AREA
*         DATE POSITION:  1
*             DATE TYPE:  C
*      CLASS YEAR BREAK:  N/A
*
* ------------------------------------------------------
*
...
```

2.44 IDMS_DROP_ABRT_RU_MINUTES

The IDMS journal log entry for ABRT does not contain any information, if the related ROLLBACK of the application has been specified with or without CONTINUE-option. Therefore, tcVISION does not automatically delete a Run Unit after an ABRT from the Run Unit list - otherwise possible other DMLs of the application would not find related AREA-informations within the same Run Unit.

Parameter IDMS_DROP_ABRT_RU_MINUTES=n defines the number of minutes after which an open Run Unit - that is ended by a ROLLBACK – is deleted from the active Run Unit list, because no more DMLs will follow for exactly that Run Unit. This delete will only occur, when parameter IDMS_DROP_ABRT_RU_JSNRS permits this.

n is a number of minutes between 0 and 3600, default value is 60.
Value 0 means all applications call ROLLBACK always without CONTINUE option and a Run Unit can be safely deleted from the tcVISION list after an ABRT is done.

Refer to parameter IDMS_DROP_ABRT_RU_JSNRS

2.45 IDMS_DROP_ABRT_RU_JSNRS

The IDMS journal log entry for ABRT does not contain any information, if the related ROLLBACK of the application has been specified with or without CONTINUE-option. Therefore, tcVISION does not automatically delete a Run Unit after an ABRT from the Run Unit list - otherwise possible other DMLs of the application would not find related AREA-informations within the same Run Unit.

Parameter IDMS_DROP_ABRT_RU_JSNRS=n defines a number of further journal-logs after which an open Run Unit - that is ended by a ROLLBACK – is deleted from the active Run Unit list, because no more DMLs will follow for exactly that Run Unit.
This delete only will occur, when parameter IDMS_DROP_ABRT_RU_JSNRS permits this.

n is a number of journal entries between 0 and 10000, default value is 500.
Value 0 means all applications call ROLLBACK always without CONTINUE option and a Run Unit can be safely deleted from the tcVISION list after an ABRT is done.

Also look for parameter IDMS_DROP_ABRT_RU_MINUTES

2.46 CICS_VSAM_Selection

The parameter CICS_VSAM_Selection defines the CICS-FCT names that should be selected when processing a z/OS CICS logstream data. The FCT names must be separated by a comma.

Example:

```
CICS_VSAM_Selection = 'ARTICLE,SUPPLR'
```

This parameter can be specified for Input_Type = 'CICS_JOURNAL' or 'CICS_LOGSTREAM' and during processing from stage 0 to stage >=1.
If this parameter has not been defined, all VSAM files in the journal/logstream will be processed.

2.47 DATACOM_Selection

The parameter DATACOM_Selection specifies the Dbids and Datacom tables needed for the Datacom Logrec processing.

Example:

```
DATACOM_Selection = '*,+*'
DATACOM_Selection = '*,+LIM'
DATACOM_Selection = '10,+LIM,+PFK'
DATACOM_Selection = '10,+LIM,+PFK;11,+OPT'
```

For Input_Type = 'DATACOM-LOGREC' the parameter can be specified for the processing of log data or for the processing of data from stage 0 to stage >=1.
If this parameter has not been specified, all Datacom data of the log file will be processed.

2.48 DATACOM_CDC_EXPIRATION_MIN

The parameter DATACOM_CDC_EXPIRATION_MIN specifies a time interval in minutes. Data in CDC tables TSN and MNT that are older than this time interval are automatically deleted from the CDC database during the active processing.
If the parameter is not specified or has a value of 0, no data is deleted.
If a data script should only delete the expired data without further processing, an output file with a name of 'DUMMY' can be specified for parameter Output_Target_Name.

Default value is 0.
The parameter value is numeric.

Example:

```
DATACOM_CDC_EXPIRATION_MIN = 1440
```

2.49 ADABAS_SVC

The parameter ADABAS_SVC specifies the Adabas-SVC to access Adabas on mainframe systems.
Valid values are 200 – 255 and a value of 0. Value 0 uses the installation defined Adabas SVC.
Default is 0.
The parameter value is numeric.

Example:

```
ADABAS_SVC = 247
```

This parameter can be specified if Input_Source_Type = 'ADABAS'.

2.50 ADABAS_SVC_CHECK

The parameter ADABAS_SVC_CHECK defines defines whether the SVC defined for Adabas access should be checked for validity on the mainframe system.
Valid values are the values 0 and 1. Value 0 deactivates the check of the Adabas SVC.
Default is 1.
The parameter value is numeric.

Example:

```
ADABAS_SVC = 0
```

Notice: The use of an incorrect SVC number with deactivated check will lead to unpredictable results.

2.51 ADABAS_MF_CTR

The parameter ADABAS_MF_CTR defines the Adabas multi fetch blocking factor.
Valid values are 1 to 500.
Default is 200.
The parameter value is numeric.

Example:

```
ADABAS_MF_CTR = 50
```

This parameter can be specified if Input_Source_Type = 'ADABAS'.

2.52 ADABAS_DBNR

The parameter ADABAS_DBNR defines the Adabas database number to be used.
The parameter value is numeric.

Example:
```
ADABAS_DBNR = 1
```

This parameter is mandatory for Input_Source_Type = 'ADABAS' and must be specified.

2.53 ADABAS_FINR

The parameter ADABAS_FINR defines the Adabas file number to be used.
The parameter value is numeric.

Example:
```
ADABAS_FINR = 1
```

This parameter is mandatory for Input_Source_Type = 'ADABAS' or 'ADAPLOG' and must be specified.

For Input_Source_Type = 'ADA_LOGREC' this parameter is used to specify the files that should be processed. Multiple files or areas can be specified, each separated by a comma.
During the pre-processing (output stage 0) starting with PLOG or SLOG, only the log records relevant for the specified Adabas files will be passed from the log blocks. If not specified or if the value is 0, the log blocks will be passed unchanged. Limiting the processing to certain files can drastically reduce the data volume of the log files.

Example:
```
ADABAS_FINR = '1,2,4-7,9'
```

During the processing (output stage 1+) the Adabas files that should be processed can be further reduced by using this parameter in an additional data script.

Example:
```
ADABAS_FINR = '4-7'
```

2.54 PM_I.ADABAS_USERID

The parameter PM_I.ADABAS_USERID defines the Adabas Userid to be used for connecting to ADABAS.
The parameter value is a character string.

Example:
```
ADABAS_CHECKPOINT_FILE = 37
```

2.55 ADABAS_CHECKPOINT_FILE

The parameter ADABAS_CHECKPOINT_FILE defines the file number of the Adabas LUW checkpoint file if the Adabas default number 1 is not used.
The parameter value is numeric.

Example:
```
PM_I.ADABAS_USERID = 'TCVUSER1'
```

2.56 ADABAS_Descriptor

The parameter ADABAS_Descriptor defines the unique Adabas descriptor to be used.
The parameter value is a 2-digit character string.

Example:
```
ADABAS_Descriptor = 'AA'
```

This parameter is mandatory for Input_Source_Type = 'ADABAS' and Input_Type = 'BATCH_COMPARE', and must be specified.

The parameter is optional for Input_Type = 'BULK_TRANSFER'.

The parameter may be specified if Input_Source_Name specifies a file in the Adabas-ADAULD compressed format and if the file has been unloaded in sorting sequence of a unique Adabas descriptor.

2.57 ADA_FromPLOG

The parameter ADA_FromPLOG defines the start PLOG session for Input_Source_Type = 'ADA_LOGREC'. Log data with a PLOG session less than this value will be ignored and skipped.
Default is 0, no selection of log data.
The parameter value is numeric.

Example:
```
ADA_FromPLOG = 123
```

2.58 ADA_ToPLOG

The parameter ADA_ToPLOG defines the end PLOG session for Input_Source_Type = 'ADA_LOGREC'. Log data with a PLOG session higher than this value will be ignored and skipped.
Default is 0, no selection of log data. If the parameter ADA_FromPLOG has been defined, the default is identical to the value of ADA_FromPLOG.
The parameter value is numeric.

Example:
```
ADA_ToPLOG = 123
```

2.59 ADA_FromBLOCK

The parameter ADA_FromBLOCK defines the start PLOG block for Input_Source_Type = 'ADAPLOG'. Log data with a PLOG block less than this value will be ignored and skipped. The start block refers to the Start PLOG session defined with parameter ADA_FromPLOG.
Default is 0, no selection of log data.
The parameter value is numeric.

Example:
```
ADA_FromBLOCK = 6
```

2.60 ADA_ToBLOCK

The parameter ADA_ToBLOCK defines the end PLOG block for Input_Source_Type = 'ADAPLOG'. Log data with a higher PLOG block will be ignored and skipped. The end block refers to the end PLOG session defined with parameter ADA_ToPLOG.
Default is 0, no selection of log data.
The parameter value is numeric.

Example:
```
ADA_ToBLOCK = 9
```

2.61 ADA_FromCP

The parameter ADA_FromCP defines the start checkpoint name for Input_Source_Type = 'ADA_LOGREC'. Log data before this checkpoint is ignored and skipped. The name refers to the start PLOG block defined with parameter ADA_FromBLOCK.
Default is '', no selection of log data.
The parameter value is a 4-digit character string.

Example:
```
ADA_FromCP = 'MYCP'
```

2.62 ADA_ToCP

The parameter ADA_ToCP defines the end checkpoint name for Input_Source_Type = 'ADA_LOGREC'. Log data after this checkpoint is ignored and skipped. The name refers to the end PLOG block defined with parameter ADA_ToBLOCK.
Default is '', no selection of log data.
The parameter value is a 4-digit character string.

Example:
```
ADA_ToCP = 'MYCP'
```

2.63 ADA_PLOG_OBFUSCATE

The parameter ADA_PLOG_OBFUSCATE copies an archived ADABAS PLOG in an output PLOG with the same structure and size, but all user data (before image, after image, descriptor values) are overwritten with hex '00'.

The data of the checkpoint file 19 is not masked and copied.
A file must be defined as output for stage 0. On a S390 System, the output file must be allocated in advance according to the input PLOG.

This parameter is only to be used for problem solving in consultation of BOS Support.

Example:

```
ADA_PLOG_OBFUSCATE = 1
```

2.64 ADA_PLOG_HEARTBEAT

The parameter ADA_PLOG_HEARTBEAT defines a tcVISION Adabas file number that is used for the NRT logreader. This Adabas file has to be defined with this file number as follows:

ADACMP FNDEF='1,TV,250,A'

Example:

```
ADA_PLOG_HEARTBEAT = 150
```

This parameter has to be specified for the NRT processing.
This ADABAS file does not have to be initialized. It only contains one record with ISN=1. That record is added, if it does not exist yet, or updated.

2.65 ADA_PLOG_GAP

The parameter ADA_PLOG_GAP specifies the maximum allowed difference of PLOG sessions when switching to the next PLOG session, block 1. If this difference is exceeded, a corresponding error message is displayed and processing is terminated.

A difference in the PLOG session after an Adabas shutdown and startup is caused by the use of ADASAV database, which increases the PLOG session by one per call at the next Adabas startup.

The default value is 2 and therefore allows an ADASAV database.

Example:

```
ADA_PLOG_GAP = 5
```

In this example, four calls of ADASAV database between an Adabas shutdown and startup are tolerated.

2.66 ADA INTERNAL_NUCID

The parameter ADA_INTERNAL_NUCID defines for ADABAS Parallel/Cluster Services the Internal Nucleus ID to be processed with values from 0 - 32 and is only used for scripts which directly access the ADABAS PLOGs.
The value corresponds to the ADABAS Internal Nucleus ID, not the External!

The ADABAS PPT (Parallel Participant Table) contains the corresponding information and can be printed using 'ADACHK PPTPRINT':

Example:
```
CHK001I, Print ASSO Block 8031 (x'00001F5F') thru 8062 (x'00001F7E') - PPT
 DB   00001 PPT AT RABN      00001F60
 DB   00001 PPT BLOCK NUMBER 02
 DB   00001 PPH  +0000            Number of entries: 03
 DB   00001 PPH  +0001            Nucleus indicator: 02
 DB   00001 PPH  +0002               External NUCID: 03E9
 DB   00001 PPH  +0004        Internal ADARES field: 00000233
 DB   00001 PPH  +0008      Last session nr written: 006D
 DB   00001 PPH  +000A         Actual header length: 0020
 DB   00001 PPH  +000C   Last PLOG block nr written: 00000005
 DB   00001 PPH  +0010           PPT init indicator: 40
 DB   00001 PPH  +0011      CLCOPY jobs in progress: 00
 DB   00001 PPH  +0012      PLCOPY jobs in progress: 00
 DB   00001 PPE  +0000          Length of PPT entry: 0028
 DB   00001 PPE  +0002   Timestamp first PLOG block: DA83AF08ABE3BF00
 DB   00001 PPE  +000A              PPT status flag: 00
...
```

The 'PPT BLOCK NUMBER' corresponds to the Internal Nucleus ID for the External Nucleus ID 0x03E9 or 1001 in decimal.

Single Nucleus Instances:
The ADA_INTERNAL_NUCID parameter is not used.

Parallel/Cluster Services Instances:
For scripts that process the active PLOG files, the ADA_INTERNAL_NUCID parameter must be used to specify the Internal Nucleus ID of the corresponding nucleus.
For scripts that process the merged PLOG files the Internal Nucleus ID of the corresponding nucleus can be specified to selectively process only the log data of this nucleus.

2.67 Input_Parms

The parameter Input_Parms will be filled with parameter values of the start command line during the prolog processing. They can be changed or enhanced. The content of this parameter allows the predefined parameter values to be dynamically modified during script start.
The parameter value is a character string.

Example:
`Input_Parms` = '`Trace=CALLS INFOS ERRORS SHORT_DUMPS'`

The above example shows how the value specified in the script for Trace will be overwritten with the new value of '`CALLS INFOS ERRORS SHORT_DUMPS'`.

2.68 Input_Retry_Times

The parameter Input_Retry_Times defines the number of retries to establish a connection with a remote sender script.
The parameter value is numeric.
This parameter is optional for Input_Type = 'REMOTE'.

2.69 Input_Retry_Interval

The parameter Input_Retry_Interval defines the time interval for which the listener port of the receiver script is kept open before a new connection attempt is made.
The parameter value is numeric.

This parameter is optional for Input_Type = 'REMOTE'.

2.70 Input_From_Timestamp

The parameter Input_From_Timestamp defines the timestamp that should be used as a starting point to process records from the data source that contains timestamps.
The parameter value is in timestamp format 'yyyy-mm-dd.hh.mm.ss.ssssss'.

This parameter is optional for Input_Type = 'PIPE', 'CICS_LOGSTREAM' 'VSAM_LOGSTREAM', 'ADA_LOGREC', 'DATACOM_LOGREC', 'IDMS_JOURNALREC', 'ORACLE_LOGREC', DB2V_FILE', and 'SMF_DATA'.

For parameter Input_Source_Type = 'DB2_ICOPY' the timestamp must be defined when the image copy has been created (Field SYSIBM.SYSCOPY.TIMESTAMP).

2.71 Input_To_Timestamp

The parameter Input_To_Timestamp defines a timestamp that should be used as a ceiling value to read data records from a data source that contains timestamps.
The parameter value is in timestamp format 'yyyy-mm-dd.hh.mm.ss.ssssss'.

This parameter is optional for Input_Type = 'PIPE', 'CICS_LOGSTREAM', 'VSAM_LOGSTREAM', 'ADA_LOGREC', 'DATACOM_LOGREC', 'IDMS_JOURNALREC', 'ORACLE_LOGREC', 'DB2V_FILE', and 'SMF_DATA'-.

2.72 State_Save_File_Out

The parameter State_Save_File_Out defines the name of the status file that will be written at the end of a log file processing.
This file contains the current level of open LUWs and of partial undo/redo log data. With the next script execution for the subsequent log file of this source database this file can be referenced with parameter State_Save_File In to correctly connect these two log files. Also refer to the description of parameter State_Save_File_In.
Both parameters can reference the same file. In this case the file will be read at the beginning of the processing and it will be written to at the end of the processing.
The parameter is a character string.

Example:

```
Save_State_file_Out = 'c:\cdc\Save_State_IDMS_Journal'
```

The parameter can be specified when multiple log files that belong to a closed DBMS session should be processed. The current state of the internal processing will be saved and reread at the start of the processing for the next log file.

The parameter is also important during restart processing. If this parameter has been specified in a receiver script and the corresponding sender script has the flag OUTPUT_FLAGS_RESTART_REMOTE specified, the defined file contains information relevant to a restart processing.

2.73 State_Save_File_In

The parameter State_Save_File_In defines the name of the status file that has been created during the processing of the previous log file with parameter State_Save_File_Out.
Also refer to the description of parameter State_Save_File_Out.
The parameter is a character string.

Example:

```
Save_State_file_In = 'c:\cdc\Save_State_IDMS_Journal'
```

The parameter is also important during restart processing. If this parameter has been specified in a receiver script and the corresponding sender script has the flag OUTPUT_FLAGS_RESTART_REMOTE specified, the defined file contains information relevant to a restart processing.

2.74 State_Save_Directory

The parameter State_Save_Directory defines where the status file is to be created during the processing of the previous log file with parameter State_Save_File_Out.
Without this parameter the file is created in the local Agent's working directory.

The parameter is a character string.

Example:

```
Save_State_Directory = '/tmp/restart/'
```

2.75 Process_Flags

The parameter Process_Flags defines data processing options.
The parameter value is numeric.

The following values are defined and can be specified individually or combined:

PROCESS_FLAGS_DATA_CORRECTION:
(hex value 0x00000001)
This value activates the automatic correction of invalid field contents.
PROCESS_FLAGS_DISCARD_NO_OUTPUT_TAB:
(hex value 0x00000002)
Data for input tables that are not linked to output tables is discarded.
PROCESS_FLAGS_IGN_NO_START_LUW:
(hex value 0x00000004)
LUW data with no corresponding LUW start is discarded.
PROCESS_FLAGS_IGN_NO_IDMS_AREA:
(hex value 0x00000008)
Data IDMS records with no registered area is discarded.

PROCESS_FLAGS_DISCARD_NO_REP_ENTRY:
(hex value 0x00000016)
Changed data for an input table that is not defined in the repository is discarded.
PROCESS_FLAGS_PRESERVE_TRAILING_BLANKS:
(hex value 0x00000032)
Preserve trailing spaces when converting fixed character fields to variable length character fields.

2.76 Output_Type

The parameter Output_Type defines the type of the target data source that should be written to. The parameter value is a character string.

The following values are defined and can only be specified individually:

FILE	The data will be written to a file.
PIPE	The data is written in the internal tcVISION format into a tcVISION pipe.
REMOTE	The data will be transferred in the internal tcVISION format to a remote 'receiver' script with Input_Type = 'REMOTE'.
WEBSPHERE_MQ	The data will be written into a WebSphere MQ queue.
CDC_POOL	The data will be written into a CDC pool in the internal tcVISION format.
ODBC	The data will be written into an ODBC object.
DRDA	The data will be written into a DRDA object.
EXASOL	The data will be written into an EXASOL object.
ORACLE	The data will be written into an Oracle object.
POSTGRESQL	The data will be written into a PostgreSQL object.
XML	The data will be written into an XML object.
DB2	The data will be written into a Db2 direct interface.
REPOSITORY	The data will be written to the object defined in the repository for the corresponding table.
ADABAS	The data will be written into an Adabas object.
DLI	The data will be written into a DLI object.
DATACOM	The data will be written into a Datacom object.
VSAM	The data will be written into a VSAM object.
IDMS	The data will be written into an IDMS object.
JDBC	The data will be written into a JDBC object.
MSSQL	The data will be written into a MS SQL Server object.
MYSQL	The data will be written into a MYSQL Server object.
HDFS	The data will be written into a HDFS (Hadoop distributed file system).
MONGODB	The data will be written into a MongoDB object.
TERADATA	The data will be written into a Teradata object.
BIGDATA	The data will be written into a BigData object.
ELASTICSEARCH	The data will be written into an ElasticSearch object.
GCP	The data will be written into a Google Cloud Platform object.

Example: `Output_Type = 'REMOTE'`

This parameter is mandatory and must be specified.
Exception: For Input_Type = 'BATCH_COMPARE' and if no output file is necessary, the parameter can be omitted (e.g. only an output snapshot file is needed).

2.77 Output_Comm_Token

The parameter Output_Comm_Token defines a connection token that the local sender script reads from the remote Agent when Output_Type = 'REMOTE'. Using the token value it is possible to establish a connection with the remote receiver script.
The parameter value is a character string.

Example: `Output_Comm_Token = 'ReceiverScript1'`

2.78 Output_File_Type

The parameter Output_File_Type defines the type of data that should be written to the target data source for Output_Type = 'FILE' or 'WEBSPHERE_MQ'.
If the parameter has been omitted or set to 0, the data will be written in the internal tcVISION format and can be processed by a subsequent script.
The parameter value is numeric.

The following values are defined and can be specified individually or in combination:

OUTPUT_FTYPE_RAW_DATA: (hex value 0x00000001)
Raw data is written for Output_Stage = 0.

OUTPUT_FTYPE_BEFORE_IMAGE: (hex value 0x00000002)
Before images will be written for Output_Stage = 1 or 2.

OUTPUT_FTYPE_AFTER_IMAGE: (hex value 0x00000004)
After images will be written for Output_Stage = 1 or 2.

OUTPUT_FTYPE_SQL_DML: (hex value 0x00000008)
SQL DML statements will be written for Output_Stage = 3.

OUTPUT_FTYPE_SQL_LOADER: (hex value 0x00000010)
If Output_Stage = 3, the data that must be inserted for Input_Type = 'BULK_TRANSFER' will be written into a tabular SQL loader format without INSERT commands. A subsequent control script can directly start the corresponding SQL loader.

OUTPUT_FTYPE_FIXED_FORMAT: (hex value 0x00000020)
The tabular data in the SQL loader format will be written with fixed field lengths.

OUTPUT_FTYPE_NUM_BINARY: (hex value 0x00000040)
The numeric fields in the SQL loader format will be written in binary. Field types SMALLINT, INTEGER, and DECIMAL are affected (also refer to options OUTPUT_FTYPE_LITTLE_ENDIAN and OUTPUT_FTYPE_BIG_ENDIAN).

OUTPUT_FTYPE_SEMI_COLON: (hex value 0x00000080)
If bit OUTPUT_FTYPE_SQL_DML is active, all SQL DML statements are terminated with a semicolon (;).

OUTPUT_FTYPE_LITTLE_ENDIAN: (hex value 0x00000100)

If bit OUTPUT_FTYPE_SQL_LOADER and OUTPUT_FTYPE_NUM_BINARY is active, all binary numeric fields will be written in format 'little-endian'. If OUTPUT_FTYPE_BIG_ENDIAN and OUTPUT_FTYPE_LITTLE_ENDIAN are not set, the 'endianess' of the local processor will be used.

OUTPUT_FTYPE_BIG_ENDIAN: (hex value 0x00000200)
If bit OUTPUT_FTYPE_SQL_LOADER and OUTPUT_FTYPE_NUM_BINARY is active, all binary numeric fields will be written in the 'big-endian' format. If OUTPUT_FTYPE_BIG_ENDIAN and OUTPUT_FTYPE_LITTLE_ENDIAN are not set, the 'endianess' of the local processor will be used.

OUTPUT_FTYPE_NULL_INDICATOR: (hex value 0x00000400)
If bit OUTPUT_FTYPE_NULL_INDICATOR is active, a NULL indicator will be sent to the output for all fields that can be NULL.

OUTPUT_FTYPE_ORACLE_LOADER: (hex value 0x00000800)
The loader output file will be written in the Oracle format.

OUTPUT_FTYPE_DB2UDB_LOADER: (hex value 0x00001000)
The loader output file will be written in the Db2 LUW format.

OUTPUT_FTYPE_ORACLE_DML: (hex value 0x00002000)
The SQL DML statements will be written in the Oracle SQL syntax.

OUTPUT_FTYPE_MSSQL_LOADER: (hex value 0x00004000)
The SQL DML statements will be written in the MS SQL Server Syntax.

OUTPUT_FTYPE_REPOSITORY_LOADER: (hex value 0x00008000)
The SQL DML statements will be written based on the output target specified in the repository.

OUTPUT_FTYPE_ZIP_DATA: (hex value 0x00020000)
The output data is transferred in the internal compressed tcVISION format

Example:
```
Output_File_Type = OUTPUT_FTYPE_BEFORE_IMAGE + OUTPUT_FTYPE_AFTER_IMAGE
```

This parameter is optional for Function = 'CDC'.

2.79 Output_Flags

The parameter Output_Flags defines options for the data that should be written into the target data sources.
The parameter value is numeric.

The following values are defined and can be specified in combination:

OUTPUT_FLAGS_USE_SQL_PARAMS: (hex value 0x00000001)
For Output_Stage = 3 the SQL DML statements will contain placeholders and the field values are supplied as parameters when writing to ODBC, DRDA, EXASOL, Oracle and PostgreSQL. For repository output target this flag can be overwritten.

OUTPUT_FLAGS_LENGTH_FIELD: (hex value 0x00000002)
For Output_File_Type <> 0 the data records have a 4 byte length field as prefix.

OUTPUT_FLAGS_CRLF: (hex value 0x00000004)
For Output_File_Type <> 0 the data records have a CR/LF suffix. For repository output target this flag can be overwritten.

OUTPUT_FLAGS_NO_SQL_COMMENTS: (hex value 0x00000008)
For Output_Stage = 3 the SQL DML statements will be generated without comments. For repository output target this flag can be overwritten.

OUTPUT_FLAGS_SCHEDULE_SCRIPT: (hex value 0x00000040)
For Output_Type = 'FILE' a data script will be immediately started after the close of an output file using the start parameter Input_Source_Name = *filename*. The parameter *filename* is equivalent to the dynamic file name as described with Output_Target_Name. The name of the script that should be started is defined with parameter Schedule_Script.

OUTPUT_FLAGS_FILE_EVENT: (hex value 0x00000080)
For Output_Type = 'FILE' a message will be sent to the scheduler script after the close of the output file. The message can initiate the start of an additional data script. This script can then process the output file and may propagate it into a target DBMS.

OUTPUT_FLAGS_NO_TRANSACTIONS: (hex value 0x00000100)
For Output_Stage = 3 all SQL transaction commands like 'COMMIT', ROLLBACK', etc. will be suppressed with this flag. For repository output target this flag can be overwritten.

OUTPUT_FLAGS_IGN_SQL_ERROR: (hex value 0x00000200)
For Output_Stage = 3 this flag suppresses all SQL error conditions that have been reported by an SQL target database when processing the SQL commands.
This prevents the cancellation of a process by isolated SQL messages. For repository output target this flag can be overwritten.

OUTPUT_FLAGS_RESTART_REMOTE: (hex value 0x00000400)
Specify this flag if a connection should be re-established after a connection loss. The attempt to reconnect is performed equal to the value specified with parameter Output_Restart_Count. If this flag has been specified in a sender script, the corresponding restart script must have RESTART files defined (refer to State_Save_File_In or State_Save_File_Out).

OUTPUT_FLAGS_LOWVAL2NULL: (hex value 0x00000800)
This flag determines that LowValues in CHAR, DATE, TIME, and TIMESTAMP fields set the output value to NULL.

OUTPUT_FLAGS_ARRAY_DML: (hex value 0x00002000)

This flag specifies that multiple data records can be propagated to the target database in blocks. The option can only be used with option OUTPUT_FLAGS_USE_SQL_PARAMS. The maximum number of data records per block can be specified in parameter DML_ARRAY_SIZE (default value: 30). Currently this option is available for output types Oracle, EXASOL, DRDA, and ODBC. For repository output target this flag can be overwritten.

OUTPUT_FLAGS_GENERATE_LDRCTL: (hex value 0x00004000)
This flag defines that the required loader control files should be automatically created for Output_Stage = 3 and Output_Type = 'FILE' and creation of loader files.

OUTPUT_USE_IDENTIFIERQUOTES: (hex value 0x00008000)
This flag defines that for Output_Stage = 3 the identifier quotes reported by the data source should not be used. Instead, the values defined with parameter IDENTIFIER_QUOTE_BEFORE and IDENTIFIER_QUOTE_AFTER should be used.
For repository output target this flag can be overwritten.

OUTPUT_FLAGS_INSTCHK_SCRIPT: (hex value 0x00020000)
This flag is evaluated during the start of a script after an output file has been created (OUTPUT_FLAGS_FILE_EVENT). If the flag has been set, a check is performed whether the script that is about to be started is already active. If it is active, no new script will be started. The script that is to be started should have wildcards defined for the input files in the input directory (e.g. *.bin) and should have either a time interval specified or the number of processed files before a new scan of the input directory is performed (refer to WAITFORFILES or TIMETOWAIT for parameter Input_Source_Name on page 17.

Example:
```
Output_Flags = OUTPUT_FLAGS_CRLF + OUTPUT_FLAGS_NO_SQL_COMMENTS
```

OUTPUT_FLAGS_BLOCKDMLTEXT: (hex value 0x00100000)
This flag is used during the output of SQL DML into a target database. If this flag is active, the individual SQL DMLs are packaged in text format and sent to the target database. The packaging is especially important for remote target databases with limited performance capacity to achieve a good performance.
The preparation and sending of the SQL DMLs is performed in separate threads to further improve the performance.

The following target databases are currently supported:

MSSQL

The following additional parameters can be used in the connection string:

MaxStatements=
The maximum number of SQL DMLs that can be sent in one request to the target database. The minimum value is 1, the maximum value is 5000, default value is 500.

MaxEntries=

The maximum number of elements in the processing batch of a session which will be sent asynchronously to the target database. Minimum value is 1, maximum value is 10485760, default value is 5120. If this value is reached during an active session, the processing will wait for a free spot.

MaxMemory=
The maximum allowed storage size of all elements in the processing batch of a session which will be sent asynchronously to the target database. Minimum value is 1, maximum value is 2147483648, default value is 524288000. If this value is reached during an active session, the processing will wait for a free spot.

Using the following additional code in the epilog (CDC_Epilog:) the most important thread runtime parameters can be displayed after the processing:

```
IF DATATYPE(PM_0.ThreadCount,'W') = 1 THEN DO
ThreadDetails = 1
SAY ' '
DO ThreadIndex = 1 TO PM_0.ThreadCount
  Thread = PM_0.ThreadName.ThreadIndex
  SAY "Statistics for Thread    '"Thread"'"
  SAY '  Number of Threads:     ' PM_0.Threads.ThreadIndex
  SAY '  Entries used:          ' PM_0.EntryCountUsed.ThreadIndex
  SAY '  HW Entries:            ' PM_0.EntryMaxUsedPercent.ThreadIndex'%'
  SAY '  HW Memory:             ' PM_0.MemoryMaxUsedPercent.ThreadIndex'%'
  SAY '  Time Active:           ' PM_0.TimeActivePercent.ThreadIndex'%'
  SAY '  Waits for Entries:     ' PM_0.EntryCountWait.ThreadIndex
  IF PM_0.Threads.ThreadIndex > 1 & ThreadDetails = 1 THEN DO
    DO ThreadPoolIndex = 1 TO PM_0.Threads.ThreadIndex
       IF PM_0.EntryCountUsed.ThreadIndex.ThreadPoolIndex = 1 THEN DO
       SAY '  Pool Thread:           ' ThreadPoolIndex 'to' PM_0.Threads.ThreadIndex 'unused'
       LEAVE
      END
      ELSE DO
        SAY '  Pool Thread:           ' ThreadPoolIndex
        SAY '    Entries used:          ' PM_0.EntryCountUsed.ThreadIndex.ThreadPoolIndex
        SAY '    HW Entries             ' PM_0.EntryMaxUsedPercent.ThreadIndex.ThreadPoolIndex'%'
        SAY '    HW Memory:             ' PM_0.MemoryMaxUsedPercent.ThreadIndex.ThreadPoolIndex'%'
        SAY '    Time Active:           ' PM_0.TimeActivePercent.ThreadIndex.ThreadPoolIndex'%'
        SAY '    Waits for Entries:     ' PM_0.EntryCountWait.ThreadIndex.ThreadPoolIndex
        SAY '    User Info:             ' PM_0.UserInfo.ThreadIndex.ThreadPoolIndex
      END
    END
  END
  SAY ' '
END
END
```

2.80 Output_Target_Name

The parameter Output_Target_Name defines the name of the data source that should receive the output data.
The parameter value is a character string.

The following values can be specified depending on Output_Type:

FILE File name. If the file should be created using multiple different names, the character '&' can be used as a separation character.
`Example: C:\Temp\filea.bin&C:\Temp\fileb.bin`
Refer to chapters 2.80.1, Dynamic File Names and *2.80.2New Datasets on z/OS Systems*

PIPE Name of the PIPE control file.
`Example: C:\Temp\MyPipe.bin`

REMOTE IP name and port of the remote data script. The format is 'a.a.a.a:xxxxx' whereas a.a.a.a is the IP address and xxxxx is the desired port. a.a.a.a can also be a DNS name.
`Example: Input_Source_Name = '192.168.0.230:5555'`
If SSL=Y is specified (optional, delimited by comma from the port), a connection for using SSL/TLS is opened.
Also refer to parameter Output_Comm_Token.

WEBSPHERE_MQ Name of the WebSphere MQ queue. Optionally an MQ GroupId can be separated by a colon and a dynamic placeholder can be specified. The MQ Group ID can only be used if the flags LUW_PROCESSING_ACTIVE and LUW_TO_FILE are active.
The following placeholders can be used for dynamic Group IDs:
<luwname> Group ID is the current name of the LUW.
<timestamp> Group ID is the current timestamp.

CDC_POOL Name of the CDC pool

DRDA DRDA connect string, e.g.
`'HOST=192.168.0.230;PORT=446;DATABASE=S390LOC;UID=FRANK;PWD=XXX X;TRACE=N;CTB=Y;PLAN=TCEXPRESS;PACKAGE=TCELF000;CONSTOK=6841446 351614450;EOS=N;DEFSYSTAB=Y;UII=Y'`

ODBC ODBC connect string, e.g. `'DSN=MyDSN;'`

MARIADB MARIADB Server connect string, e.g.
`'SERVER=sv;UID=u;PWD=p;'`

MSSQL SQL Server connect string, e.g.
`'SERVER=sv;DATABASE=db;UID=u;PWD=p;'`

MYSQL MYSQL Server connect string, e.g.
`'SERVER=sv;UID=u;PWD=p;'`

ORACLE Oracle OCI connect string, e.g.
`'DSN=ServiceName;UID=userid;PWD=Password'`

POSTGRESQL PostgreSQL connect string, e.g.
`'host=127.0.0.1;port=5432;dbname=postgres;user=name;password=pw d;'`

EXASOL EXASOL connect string, e.g.
`'SERVER=address:port;UID=userid;PWD=Password'`

MONGODB MongoDB connect string, e.g.
`'HOST=192.168.0.31;PORT=27017;DATABASE=SAMPLE'`

BIGDATA BigData connect string, e.g.
`'Protocol=JSON;Layer=Kafka;PNV=1;LF=3;Topic=test1;Brokers=local host:9092;'`
with the following keywords:
Protocol=
selects the protocol. Options are JSON, AVRO, and CSV.

JSON:
The output is structured as JSON objects.

Each operation gets represented by one JSON Document which contains the table name, the Operation type and a data document including the fields of the output table.
JSON specific connection Parameters are:
LF: This value is a combination of flag bits, the values 1 to 4 extend the field format, higher ones add information to the outer part of the JSON document.
1 = Include the field type
2 = Include the position of the field in the key
4 = Include info about the maximum length in chars this field can have in a JSON document

If one of these first 3 is set it expands the individual fields from a key-value pair into it's own document, changing the data document into a data array. Using this is not advised as it leads to storing information that you only need once per table in every document. An alternative would be using a JSON Schema where these information can be stored. Also this extended format leads to a format that can't be used as a source for JSON&Kafka as source.

8 = the timestamp of the source data is included in the outer document
16 = the timestamp of the source luw (commit) is included in the outer document
32 = the timestamp of processing (creation of the JSON document by tcVISION) is included in the outer document
64 = the URID (unique recovery id) is included in the outer layer of the JSON document
128 = the SQL Comment is included in the outer layer of the JSON Document.

So e.g. an LF of 40 would mean that the timestamp of the luw and the timestamp of processing are included but non of the other options.

PNV: If set to 1 Null fields with null values get printed. Otherwise they are left out.

JFF: Only exists for JSON with File. If not set or set to 0 the JSON documents for all records in the file are collected in one large record document, so the whole document is a JSON document. If set to 1 then after each JSON document for an operation a new line begins.

AVRO:
The implementation was done according to AVRO 1.8.1.
The schema is created based on the structure of the processed table.
A requirement for AVRO is that the name of the topic/file of the layer contains the <table> token, otherwise the records will not match the schema.
One record contains the information if it is an Update/Insert/Delete, a unique key (concatenation of key fields), and a record containing the fields of the table.
In the schema, there are - in addition to the *name* and avro *type* of each field of - also the attributes *keypos* and *sqlfieldtype* which contain information about the position in the primary key and the original sql type respectively.

When handling BLOBs and CLOBs, it should be noted that with CDC there is usually no before & after image available in the log files unless the BLOB/CLOB changes.
If there is no data for such a field in either the before or after image, the value "<unchanged>" is used here. This means that no changes have been made to the value of the BLOB/CLOB and not that the BLOB/CLOB has the value "<unchanged>".
If the layer is *file* or *hdfs*, the output is structured as an Avro file. A block is one transaction if transactions are enabled. Otherwise a block is one undo/redo record. The only exception is a key value update where one block without transactions would contain two records as it is handled as one delete and one insert.

AVRO specific connection string parameters are:
LF: As in JSON this parameter is a collection of Flag bits with the following meanings:
1 = Include a field for the table name in the outer record.
4 = All fields use the AVRO Type string, the sql type is still written based on the output field type in the schema.
8 = the timestamp of the source data is included in the outer record
16 = the timestamp of the source luw (commit) is included in the outer record
32 = the timestamp of processing (creation of the AVRO record by tcVISION) is included in the outer record
64 = the URID (unique recovery id) is included in the outer layer of the JSON record
128 = the SQL Comment is included in the outer layer of the AVRO record.

If AVRO is combined with Kafka, Confluent Cloud, Microsoft Event Hub or other messaging systems a place is needed to put the schema to. In this case one of the following options can be set:
SCHEMATOPIC= Name of a topic, where the schema definition is written to, in combination with the schema name.
SCHEMAREGISTRY= Url:Port of the Confluent Schema Registry.
HDFREGISTRY= Url:Port of the Hortonworks Schema Registry.

If AVRO is combined with a file, the schema is always at the start of the file. However there are other options:
COMPRESSION= If this parameter is set to snappy then the AVRO blocks of the file are snappy compressed otherwise no compression is used.
AVROBLOCKTARGETSIZE= How large a block should ideally be. Default is 5M (5 Megabyte). Supports K for kilobyte , M for megabyte or G for gigabyte.

CSV:
HL= If set to 1 the csv headline is printed at the start of the file, otherwise no headline is printed.
DECIMALPOINT= Specifies the decimal point used in the csv format, options are . or ,
DELIMITER= Specifies the delimiter used in the csv format, can be up to 10 characters long.
QUOTES= Specifies the quotes used in the csv format, can be up to 10 characters long.

Unlike for AVRO and JSON in the CSV format the data is written as is without any extra fields. Due to this only Inserts can be processed. If a CDC

replication should be established a journal replication should be selected in the repository and an extra field for the replication type should be added.

Layer=
Selects the layer. Available options are File, *HDFS* and *Kafka*.

File:
The only parameter is FILE= specifying the file name.
The name can contain most dynamic tokens. Only LUWtime / LUWtime_UTC or Counter is not supported. It is important that the user has write access to the indicated path.

HDFS:
Apart from the file name, connection information to the HDFS interface is required here.
PATH=
The path to the hdfs file including the url, e.g. 192.168.0.78:8020/tmp/test.csv
UH=
Sets the dfs.client.use.datanode.hostname to true, forcing it to use the name instead of the ip address if set to 1. Otherwise it stays at the default (true)
KEYTAB=
Path to the used keytab for kerberos, if kerberos is to be used.
PRINCIPAL=
Principal for the kerberos authentification, if kerberos is to be used.

Kafka:
The name of the topic and the connection string for the broker is required.
The use of the <table> token is recommended for the topic in order to only need one output target. The <table> token is replaced by the name of the output table in the metadata repository.
There is also a topic for schemas and the option to synchronize the messages. This ensures the sequence of the transactions to remain unchanged, but the performance is decreased.
This option is disabled by default, because Kafka is characterized by not having to wait for the answer. However, this means that the proper sequence of the transactions is not guaranteed and this might lead to a breach of integrity in the replication.
The group “target of the Avro schemas” is only available if Avro is used as protocol. This defines where the schemas for the respective output targets should be stored. There are three possibilities:
- In the same topic as the data
- In a separate topic
- In the Confluent schema registry

If the schema is stored in the same topic as the data, the schema is sent to the respective topic before the first operation, then the data is sent. In order to do this, there has to be a <table> token in the data topic. Otherwise the data cannot be matched with the respective schema.
If the schema is stored in a separate topic, the schema is sent to that topic with a unique identifier as message key. The identifier is composed as follows:
2 byte length (big-endian): Length of the identifier
8 byte todclock: Timestamp of the last change of the output table in the repository

N-10 byte table name: Remaining length, name of the output table in the metadata repository

This identifier is used as key when sending the message to the respective topic. The message begins with the identifier, followed by the Avro record. The schema is sent to the topic for schemas if an undo/redo record for the output table is sent for the first time. It is sent again in the running script if the output table in the metadata repository is changed and the metadata cache for the running script is flushed.
Messages in the topic for schemas with the same message key have the same content. Therefore it is recommended to enable the log compacting for the selected script. Then only the most recent message is stored if there are two or more messages with the same key.
If the Confluent Schema Registry is used, the schemas are sent to the Confluent Schema Registry where they are assigned a unique ID.
It is recommended to turn off the compatibility in the registry, because the schemas may not be compatible anymore after adjusting the replication. This can lead to error messages in the schema registry.
The first 5 byte in the messages to the data topic are the identifier. The first byte is always a 0 byte. This is followed by a 4 byte big-endian integer. The identifier is followed by the Avro record.

The connection string of Kafka supports the following Parameters:
TOPIC=
Name of the Kafka topic. Allows for the tokens <table> or <target> which get replaced by the name of the output table for <table> or the output target for <target>.
BROKERS=
Url:Port for the broker. For multiple brokers the brokers are separated by a ',' The list is only relevant to establish the initial connection, after that librdkafka retrieves all brokers participating in the kafka cluster from the broker it connected to.
SERVICENAME=
Name of the service when signing up using kerberos.
KEYTAB=
Path to the keytab to be used when signing up using kerberos.
PRINCIPAL=
Name of the principal to be used with the keytab when signing up with kerberos.
SSLCLIENTCERT=
SSL Certificate to be used when connecting to the kafka broker using SSL.
SSLKEY=
SSL Key to be used when connecting to the kafka broker using SSL
SSLKEYPWD=
Password for the SSL Key used when connecting to the kafka broker.
SSLCERT=
Certificate used to verify the broker certificate during the SSL handshake.
IDEM =
If set to 1 enables the idempotent broker for Kafka. Default is 1 if Transactions are enabled and 0 if transactions are disabled. Transactions are only possible with an idempotent broker.
TRANSACTIONTIMEOUTMS=
Sets the parameter transaction.timeout.ms for the connection to kafka, meaning it defines the amount of time in milliseconds the transaction coordinator will wait for a status update from the producer before aborting the transaction. Default is one minute (60000 ms)

If SSL is used to connect to kafka and to the schema registry, but the later should have different parameters use the following options:
SSLREGCERT =
SSL Certificate used to verify the certificate of the schema registry during the SSL Handshake

SSLREGCLIENTCERT =
SSL Certificate sent to the schema registry for the SSL Handshake.
SSLREGKEY =
SSL Key used for the SSL hand shake with the schema registry.
SSLREGKEYPWD=
Password for the SSL Key used for the SSL hand shake with the schema registry.
If SSL should only be used to Kafka use REGNOSSL=1 and if Kerberos should not be used to the Schema registry use REGNOKERB=1.

TERADATA

Teradata connect string, e.g.
`'host=192.168.0.103;user=dbc;password=enc1234567893543;MaxStatements=500;Sessions=16;'`
uses the following keywords:

host=	IP name of the Teradata server
user=	Logon user ID
password=	Logon password

MaxStatements=
The maximum number of SQL DMLs that can be sent to the Teradata server in one request. Minimum value is 1, maximum value is 500, default value is 50.

Sessions=
The maximum number of parallel sessions that can be established to the Teradata server. Minimum value is 1, maximum value is 32, default value is 1.

MaxEntries=
The maximum number of elements in a processing batch of a session that can be passed to the target database asynchronously. Minimum value is 1, maximum value is 10485760, default value is 5120 for bulk scripts and 512000 for CDC scripts. If this value is reached during an active session, a new session will be started for bulk scripts. In case of a CDC script, processing waits for a free slot.

MaxMemory=
The maximum allowed storage size of all combined elements in a processing batch of a session that can be passed to the target database asynchronously. Minimum value is 1, maximum value is 2147483648, default value is 524288000. If this value is reached during an active session, a new session will be started for bulk scripts. In case of a CDC script, processing waits for a free slot.

NoXact=

This parameter allows the processing in Teradata without transactions. Each SQL statement is finally applied to the target database. This technique prevents the creation of deadlocks which may sporadically be created because of the Teradata hash locks. Possible values are 0 and 1. With a specification of 1 the integration of data is performed in auto commit mode. The integrity of the individual source transactions cannot be guaranteed.

Notes:
The number of sessions specified with parameter Sessions= is started as required to avoid waiting times on already started sessions.

CDC Data Script:
If Sessions > 1, the data script does NOT wait for the confirmation of COMMIT/ROLLBACK commands from the Teradata server and immediately continues processing for the next transaction to allow the parallel processing of multiple transactions. The integrity of the individual transactions is guaranteed, however the correct sequence of the transaction is not ensured. Based on the types of SQL DMLs as part of the transactions processing, errors may occur (for example: UPDATE of a not yet existing record, violated constraints, etc.) and wrong data in the target database. Compared to single session processing, the processing speed can be considerably increased. The CDC script sends the data of a transaction to a single session. The data of the following transaction is passed to the next session. It may be necessary to increase the values of MaxEntries= and MaxMemory= to pass large transactions to the processing batch of a session without waiting times.

Bulk/Batch Compare Data Script:
If Sessions > 1, the data script attempts to assign the sessions to the individual output tables to use the specified number of rows per request (parameter MaxStatements=) in the case of bound parameters. If necessary, multiple sessions per output table will be started to avoid waiting times on a session.

DB2

Db2 requires the following parameters:

DATABASE= Name of the Db2 database
UID= User ID. This parameter is only required if a RRSAF connection to Db2 is used.
PWD= Password. This parameter is only required if a RRSAF connection to Db2 is used.
RREPL=1 This parameter establishes a RRSAF connection with option 'SET_REPLICATION'.

Example:

```
'DATABASE=DB8G'
  'DATABASE=XYZG;UID=userid;PWD=password'
```

ADABAS

SVC= Specifies the Adabas-SVC to access Adabas on the mainframe.
DBNR= Specifies the Adabas database number to access Adabas.
USERID= Specifies the user ID to access Adabas.
TNAE= Set non-activity time limit on OP for Adabas/LUW

ICDLY= A connection is closed after n seconds of inactivity in Adabas/LUW. The value n should be lower than TNAE= or the default TNAE ADARUN parameter.

DLI

The following parameters are only required when using DBCTL to access IMS on a z/OS system. If the script process runs in a partition/region of its own, the necessary DLI access mechanisms will be used.

DRA= Specifies the name of the DRA connection table to access IMS/DB on a z/OS mainframe system.

PSB= Specifies the PSB to access IMS/DB on a z/OS mainframe system.

DATACOM

URT= Specifies the name of the URT to access Datacom on the mainframe.

VSAM

CICS= If the output operation should be performed using the tcVISION CICS Listener, the IP address and the port of the Listener must be specified.

TRAN= Transaction ID to use if the output operation should be performed using the tcVISION CICS Listener.

UID=/PWD= Signon information for the tcVISION CICS Listener.

IDMS

MODULE= Specifies the name of the IDMS access module. If this parameter is not specified, 'IDMS' is used.

SUBSCHEMA= Specifies the name of the IDMS subschema for the IDMS access. The specified subschema should contain all involved records.

DBNAME= Optional parameter that specifies a special database name used for the IDMS command '@BIND SUBSCH'.

DBNODE= Optional parameter that specifies a special database node used for the IDMS command '@BIND SUBSCH'.

READY_ALL Optional parameter for the execution of IDMS command '@READY ALL' and subsequent ignore of area definitions in the repository.

NO_READY Optional parameter to suppress all IDMS commands '@READY'.

JDBC

JDBC connection string with the following keywords:

CLASS= Name of the Java driver class

UID= User name for signing on to JDBC target

PWD= User password for signing on to JDBC target

DRIVER= Url of the driver, mainly in the form jdbc://... The exact syntax depends on the selected JDBC driver.

SUBTYPE= JDBC Subtype. If there is one for the JDBC driver, please use it.

To use the JDBC output target, the JDBC driver has to be added to the CLASSPATH. The tcvj.jar must be in the same directory as the tcSCRIPT or in the CLASSPATH. A Java version 7 or higher is required.

AWS

AWS connection string, e.g.

```
'AWS=Kinesis;StreamName=bos1;Region=us-east-1;
CredentialFile=mycredentials_file.txt;ChunkSize=100;
ProxyHost=localhost;ProxyPort=8888;MaxConnections=20;'
```

The keywords are:
AWS=
Name of the AWS service to which the data is transferred. The options are 'S3 Bucket', 'Aurora MySQL', 'Aurora PostgreSQL', and 'Kinesis' and 'Redshift'. The rest of the parameters depend on the AWS service selected. For 'Aurora MySQL', 'Aurora PostgreSQL' and 'Aurora MariaDB' the Parameters are the same as for MySQL, PostgreSQL and MariaDB.

S3 Bucket:

BUCKET=
Name of the S3 bucket the file should be written to.
FILE=
Name of the S3 file. This file is also used to write the temporary file in the local file system before moving it to the s3 bucket.
PROTOCOL =
The used protocol. Options are JSON, CSV and AVRO.
Parameters for the protocol are the same as listed for the protocols for the BigData output target.

Kinesis: The target is an AWS Kinesis stream. The parameters are:

StreamName=
Name of an existing Kinesis stream. Mandatory parameter.

Region=
AWS region name. Mandatory parameter.

CredentialFile=
Path to the file containing the AWS access key ID and AWS secret access key. This information is expected to be in the following format:

```
aws_access_key_id=YourAccessKeyID
aws_secret_access_key=YourSecretKey
```

If you have the AWS CLI installed on your machine, you can also use the corresponding credentials file. If a relative path is used for files, its root directory is the tcVISION working directory.
The parameter is optional. If it is not set, it is assumed that the script is running on an EC2 instance and there is an IAM role assigned to the instance, which controls access to Kinesis streams (and possibly to other AWS resources). In that case no credentials need to be stored on an EC2 instance. Instead, temporary security credentials will be requested from IAM service in the background. These temporary credentials are used to sign Kinesis requests.

ProxyHost=
Address of the proxy server if you use one. Optional parameter.

ProxyPort=

Port of the proxy server if you use one. Optional parameter.

ChunkSize=
It is possible to send multiple records to a Kinesis stream in a PUT request, i.e. in packets or chunks. This parameter specifies the maximum number of the records which are sent as a packet. Please note that the actual request size depends on the size of the transaction in the data source.
Optional parameter. The default value is 500.

MaxConnections=
Maximum number of simultaneous connections to the Kinesis server.
Optional parameter. The default value is 1024.

VerbosityLevel=
Influences what trace is written while the INFORMATION_ENTRIES are enabled. Goes from 0 to 3 adding more detail as the number increases.

Redshift:

The Parameters after the AWS=REDSHIFT are the ODBC Redshift connection string parameters. The finished string is just passed on to AWS.

CONFLUENT CLOUD

Confluent Cloud is a Kafka as a service platform managed by confluent. Most parameters are the same as for Kafka with BigData but the process to sign on works differently.

The following parameters are in the confluent cloud connection string:
PROTOCOL=
The protocol the data is written in. Either JSON, AVRO or CSV working the same as described in the BigData section.

BOOTSTRAPSERVER=
URL:Port to connect to the confluent cloud.

CLUSTERKEY=
The cluster key of the key:secret pair created to connect to the confluent cloud.

CLUSTERSECRET=
The cluster secret of the key:secret pair.

CACERT=
A CA cert is needed to verify the confluent cloud platform during the ssl handshake. Most platforms have one installed by default. If a valid one is not in the standard directories you can specify one using this parameter.

If AVRO is used as a protocol the following settings are needed to connect to the registry:

SCHEMAREGISTRY=
Url:Port of the confluent cloud schema registry.

REGISTRYKEY=
The registry key of the key:secret pair created to connect to the schema registry.

REGISTRYSECRET=
The registry secret of the key:secret pair created to connect to the schema registry.

MSA

Microsoft Azure Connect String, e.g.

```
'MSA=AzureR;AccountName=myazurestorage;SharedKey=mysharedkey;
Container=mycontainer;ProxyHost=blob.core.windows.net;
BlobName=myfolder/myblob;Protocol=JSON;;'
```

The keywords are:
MSA=
Name of the Azure services to which the data is transferred.
The possible options are:

AzureM:	Azure database for MySQL
AzureP:	Azure database for PostgreSQL
AzureS:	Azure SQL database
AzureE:	Event Hub
AzureR:	Blob storage

The further parameters vary depending on the selected MSA service (for MySQL, PostgreSQL, and MSSQL the parameters are explained earlier in this chapter).

Blob Storage: The target is an Azure Blob storage. The parameters are:

AccountName=
Name of an existing Blob storage account. Mandatory parameter.

The three parameters SharedKey=, ASAS=, and SSAS= are listed as mandatory, but only one of these three parameters is necessary for authentication (either SharedKey=, ASAS=, or SSAS=).

SharedKey=
The shared key to this Blob storage. Mandatory parameter.

ASAS=
The token of an account SAS (Shared Access Signature) to be used.
Token format (to be created with an Azure administration program, for example the Azure portal or the Microsoft Azure Storage Explorer):
?sv=...&ss=b&srt=sco&sp=rwlac&se=...&st=...&spr=https&sig=...
Mandatory parameter.

SSAS=
The token of the service SAS (Shared Access Signature) to be used.
Token format (to be created with an Azure administration program, for example the Azure portal or the Microsoft Azure Storage Explorer):
?sp=rwl&st=...&se=...&sv=...&sig=...&sr=c
Mandatory parameter.

Container=
A container in this Blob storage that should hold the Blob to be transferred. Mandatory parameter.

ProxyHost=
Address of the proxy server. Mandatory parameter (the default value is ‚blob.core.windows.net').

MBS=
Maximum size in MB (4 – 100) of the Blob block to be written.

BlobName=
The name (including the optional path) of the Blob to be transferred. The three tokens <name>, <timestamp>, and <timestamp_utc> are available to make the name variable. The tokens are solved at runtime. Mandatory parameter.

The three parameters CR=, CT=, and CS= are only required if the output target should be used for a CDC script. Such a script can theoretically run endlessly, so a Blob should be completed regularly, so that the data of the Blobs is available for further processing. The parameter CS= is an addition to the parameters CR= and CT=. The Blob is completed depending on what happens first (x records/transactions or y seconds without activity). The name should contain a token (<timestamp> or <timestamp_utc>) that is solved at runtime, so that the Blob is not constantly overwritten.

CR=
Number of records after which the Blob is completed (committed). Optional parameter.

CT=
Number of transactions after which the Blob is completed (committed). Optional parameter.

CS=
Number of seconds without activity after which the Blob is completed (committed). Optional parameter.

Protocol=
Specification of the protocol to be used. The options are CSV, JSON, and AVRO. Depending on the selected protocol, further parameters are possible (see **BIGDATA** earlier in this chapter).

EventHub

EventHub is only implemented using the kafka compatibility method.
Its connection string options are as follows:

EVENTHUB=
Name of the eventhub (equivalent to kafka topic) the data is written to.

BOOTSTRAPSERVER=
In the format URL:Port as displayed on the web page where the Eventhub was configured.

ENDPOINT=
The endpoint as displayed in the azure cloud portal where the endpoint was setup.

SHAREDACCESSKEYNAME=
Name of the access key used to connect to the event hub.

SHAREDACCESSKEY=
Value of the access key used to connect to the event hub.

PROTOCOL=
Used Protocol for the data written to the EventHub.
Can be JSON, CSV or AVRO. For Detail please look at the BigData Output Target.

ELASTICSEARCH

Elasticsearch connect string e.g.
URL=192.168.0.93:9300;SYNCM=1;BC=50

The keywords are:
URL=
IP address of the Elasticsearch instance, including port.
SYNCM=
Transfer mode
1=Synchronous
2=Asynchronous
For an asynchronous transfer, multiple connections are opened and the data is transferred without waiting for the result.
An asynchronous transfer (SYNCM=2) does **not** guarantee referential integrity.
In internal tests synchronous transfers via batch were the most efficient, even for bulks. Therefore it is recommended to use synchronous transfers instead of the asynchronous method.
BC=
Maximum number of commands that can be sent in a bulk.
This parameter has to be adjusted depending on the environment. The default value is 50.
This parameter is only used in synchronous transfers.
At the end of a transaction, all the operations collected up until then are transferred, even if the maximum number of connections per bulk has not been reached.
HC=
Maximum number of Handles that can be open simultaneously.
This number should be lower than the maximum number defined in Elasticsearch because other applications can also open connections there.
The default value here is 195, since Elasticsearch allows 200 connections by default. If the number is exceeded, tcSCRIPT terminates with an error message.
The parameter HC= is only used for asynchronous transfers.
AUTH=
Token used for the authentification.
Passed to the libcurl as CURLOPT_XOAUTH2_BEARER.
USER=
PWD=

Data for the authentification with user name and password. Passed to the libcurl as CURLOPT_USERPWD.
SSLCLIENTCERT=
SSL client certificate. Passed to the libcurl as CURLOPT_SSLCERT.
SSLKEY=
SSL key file. Passed to the libcurl as CURLOPT_SSLKEY.
SSLKEYPWD=
Password of the SSL key file. Passed to the libcurl as CURLOPT_KEYPASSWD.
SSLCERT=
SSL CA certificate. Passed to the libcurl as CURLOPT_CAINFO.

GCP

Google Cloud Platform Connect String, e.g.
'GCP=S;'

The keywords are:
GCP=
Type of the Google Cloud Platform database to which the data is transferred.
The possible options are:
M: Google Cloud Platform database for MySQL
N: Google Cloud Platform database for MariaDB
P: Google Cloud Platform database for PostgreSQL
S: Google Cloud Platform SQL database
B: Google Cloud Platform cloud storage (Bucket)
The further parameters vary depending on the selected GCP database (for MySQL, PostgreSQL, and MSSQL the parameters are explained earlier in this chapter).

Cloud Storage: The target is a Google cloud storage. The parameters are:

SA=
Email address of the used service account. Mandatory parameter.

ProjectId=
The project ID of the respective Google Cloud Platform account. Mandatory parameter.

PK=
The private key used for encrypting when interacting with the cloud Storage. As this key is encrypted itself, it can only be set with the help of the tcVISION control board. Mandatory parameter.

PKF=
A file containing the private key used for encrypting when interacting with the cloud Storage. This file is only needed if the resulting connection string would exceed the maximum allowed length in the repository due to the very long private key.

Bucket=
The name of the bucket that should be used. Mandatory parameter.

SC=
The storage class of the bucket (‘standard’, ‘nearline’ or ‘coldline’). Mandatory parameter.

Location=
The location of the bucket. Mandatory parameter.

MBS=
Maximum size in MB (4 – 100) of the Object block to be written.

Object=
The name of the object. The three tokens <name>, <timestamp>, and <timestamp_utc> are available to make the name variable. The tokens are solved at runtime. Mandatory parameter.

The three parameters CR=, CT=, and CS= are only required if the output target should be used for a CDC script. Such a script can theoretically run endlessly, so an object should be completed regularly, so that the data of the objects is available for further processing. The parameter CS= is an addition to the parameters CR= and CT=. The object is completed depending on what happens first (x records/transactions or y seconds without activity). The name should contain a token (<timestamp> or <timestamp_utc>) that is solved at runtime, so that the object is not constantly overwritten.

CR=
Number of records after which the object is completed (committed). Optional parameter.

CT=
Number of transactions after which the object is completed (committed). Optional parameter.

CS=
Number of seconds without activity after which the object is completed (committed). Optional parameter.

Protocol=
Specification of the protocol to be used. The options are CSV, JSON, and AVRO. Depending on the selected protocol, further parameters are possible (see under **BIGDATA** earlier in this chapter).

The parameter Output_Target_Name is mandatory for Output_Type = 'FILE', 'REMOTE', 'WEBSPHERE_MQ', 'CDC_POOL', 'ODBC', 'EXASOL', 'MYSQL', 'POSTGRESQL', 'ORACLE', 'DRDA', 'ADABAS', 'DLI', 'DATACOM', 'VSAM', 'IDMS', 'AWS', 'MSA', 'GCP'

2.80.1 Dynamic File Names

The following placeholders can be used for dynamic file names:

<luwname>	Is replaced by the actual name of the LUW. This placeholder can only be used with the flags LUW_PROCESSING_ACTIVE and LUW_TO_FILE active.
<scriptname>	Is replaced by the actual script name

<scriptid>	Is replaced by the actual script ID
<timestamp>	Is replaced by the actual timestamp
<counter>	Is replaced by a running counter starting with 1
<token>	Is replaced with the actual communication token
<CDC_ExitDBControl>	Exit 'CDC_ExitDBControl' should be used to dynamically specify the output target name

Example:
Output_Target_Name = `'Trace_of.<scriptname>.at.<timestamp>'`

2.80.2 New Datasets on z/OS Systems

The following parameters can be used for allocation of new datasets on z/OS systems:

RECFM=	Record format [F, V, U, B. Ex. 'F', 'FB', 'VB', 'U']
LRECL=	Logical Record Size
BLKSIZE=	Block Size
SPACE=	Space type [TRK, CYL, BLK]
PRI=	Primary space
SEC=	Secondary space
DIR=	Directory blocks
UNIT=	UNIT
VOLSER=	VOLSER
RLSE=	RLSE=1 Release unused space at close
DISP=	Disposition Open [NEW, MOD, OLD, SHR]
DISPTERM=	Disposition termination [KEEP, DELETE, CATLG, UNCATLG]

Example:
Output_Target_Name =
```
'QUAL1.QUAL2.DATA;RECFM=VB;LRECL=32000;BLKSIZE=32004;SPACE=CYL;PRI=10;UNIT=3390;
VOLSER=OS39M1;DISP=NEW;DISPTERM=CATLG'
```

2.81 Output_File_Mode

The parameter Output_File_Mode defines the mode for opening the target file.
The parameter value is a character string.

The following values are defined and can only be specified individually:

APPEND	The data will be appended to an existing file (default).
WRITE	Existing data records in the file will be overwritten.
NO OVERWRITE	The file will be created. An existing file will not be overwritten. Processing terminates with an error.

This parameter is optional for Output_Type = 'FILE' and 'PIPE'.

2.82 Output_Segment_Interval

The parameter Output Segment_Interval allows the segmentation of the output into multiple output files. The specified value defines the number of data records per segment. If the parameter has not been defined or assigned a value of 0, no segmentation will be performed. Also refer to chapter *2.80.1, Dynamic File Names*.

The parameter value is numeric.

This parameter can be specified when Output_Type = ’FILE’ and when Output_Stage is equal to 3.

2.83 Output_Commit_Interval

The parameter Output Commit_Interval allows an LUW segmentation of the output to multiple LUWs so that the transaction management of the target database does not have to manage large transaction areas. The specified value defines the number of data records per LUW segment. If the parameter has not been defined or assigned a value of 0, no segmentation will be performed.
The parameter value is numeric.

This parameter can be specified for Output_Stage = 3.

2.84 Output_Target-Object

The parameter Output_Target_Object defines the name of the SQL target object for the creation of SQL DML statements.
The parameter value is a character string.

Example: `Output_Target_Object = 'MyTable'`

This parameter is mandatory and must be defined for Input_Type = 'BATCH_COMPARE' or 'BULK_TRANSFER', and Input_Source_Type = 'DRDA', 'ODBC', 'EXASOL', 'ORACLE' or 'POSTGRESQL'.

2.85 Output _Retry_Times

The parameter Output Retry_Times defines the number of retries to establish a connection with a remove receiver script.
The parameter value is numeric.

This parameter is optional for Output_Type = 'REMOTE'.

2.86 Output _Retry_Interval

The parameter Output_Retry_Interval defines the wait interval to reconnect after an unsuccessful connection attempt.
The parameter value is numeric.

This parameter is optional for Output Type = 'REMOTE'.

2.87 Output_Restart_Count

The parameter Output Restart_Count defines the number of attempts to re-establish a connection to a remote receiver script after a connection loss.

The parameter value is numeric.

Specify this parameter if the flag OUTPUT_FLAGS_RESTART_REMOTE has been set.

2.88 Output _Stage

The parameter Output_Stage defines the desired output stage up to which the data will be prepared before it is passed to the output.
The parameter value is numeric and can contain the values 0, 1, 2, or 3.
The meaning of values 0 to 3 is described in the following chapter.

This parameter is mandatory and must be specified.

2.88.1 Output Stage 0

Stage 0 records are in a data source-specific format. The data records can be physically and/or logically blocked and compressed. Data from DBMS log files are in the internal format of the corresponding DBMS.
This stage is recommended for the transfer between two scripts running on remote systems.

2.88.2 Output Stage 1

Stage 1 records are unblocked and decompressed. They are also divided into before and after images. A lot of data source-specific information like DBMS name, user ID, etc. have been prepared.
If existing, transaction-relevant information like start of a transaction, COMMIT, ROLLBACK, end of the transaction, etc. have been evaluated.
The before and after images are still in the data source-specific format.

2.88.3 Output Stage 2

Stage 2 records consist of before and after images that have been prepared to become SQL data rows using the structure information provided by the structure import wizard.

2.88.4 Output Stage 3

Stage 3 records consist of SQL DML statements that have been created based on the data from stage 2. Optional comments with additional information can be created. This information is based on the underlying DBMS.
These SQL DML statements can be passed directly to the target SQL DBMS to apply the changes.

2.89 Output_Log_Name

The parameter Output_Log_Name defines an optional protocol file for DML commands. All commands that have been successfully executed will be written to this file in text format. Refer to *2.92 Output_Bad_Name*.

Example:

```
Output_Log_Name = 'YES'
```

This parameter can be specified in conjunction with 'Output_Stage = 3'.

2.90 Output_Log_Seg_Size

The parameter Output_Log_Seg_Size controls the segmentation of protocol files for successfully executed DML commands.
The parameter value is a number in MB that specifies the maximum size of a protocol file in MB. If the current protocol file exceeds this size, it is closed and a new segment is opened.

Example:
```
Output_Log_Seg_Size = 10
```

This parameter is optional. If it is not defined, segmentation will not be performed.

2.91 Output_Log_Seg_Compress

If protocol files for successfully executed DML commands are to be segmented (see parameter Output_Log_Seg_Size), this parameter controls the compression of file segments.
The parameter value is 0 (no compression of files) or 1 (compression activated).

Example:
```
Output_Log_Seg_Compress = 1
```

This parameter is optional. If it is not defined, compression will not be performed. This parameter has no effect on z/OS systems.

2.92 Output_Bad_Name

The parameter Output_Bad_Name defines an optional protocol file for DML command errors. All commands that have not been executed successfully will be written to this file in text format together with the error messages. Refer to *2.89 Output_Log_Name*.

Example:
```
Output_Bad_Name = 'YES'
```

The parameter can be defined in conjunction with 'Output_Stage = 3'.

2.93 Output_Bad_Seg_Size

The parameter Output_Bad_Seg_Size controls the segmentation of protocol files for not successfully executed DML commands.
The parameter value is a number in MB that specifies the maximum size of a protocol file in MB. If the current protocol file exceeds this size, it is closed and a new segment is opened.

Example:
```
Output_Bad_Seg_Size = 10
```

This parameter is optional. If it is not defined, segmentation will not be performed.

2.94 Output_Bad_Seg_Compress

If protocol files for not successfully executed DML commands are to be segmented (see parameter Output_Bad_Seg_Size), this parameter controls the compression of file segments. The parameter value is 0 (no compression of files) or 1 (compression activated).

Example:
```
Output_Bad_Seg_Compress = 1
```

This parameter is optional. If it is not defined, compression will not be performed. This parameter has no effect on z/OS systems.

2.95 Output_Info_Flags

The parameter Output_Info_Flags activates the writing to an optional information protocol file. Information concerning the data processing is written depending on the parameter Output_Info_Flags.

The following values can be defined and combined. They specify what type of information is written to the information protocol file:

INFO_SUPPRESSED_SQL_DML:
(hex value 0x00000001)
SQL UPDATE commands that did not change any data will be written to the information log file.

INFO_NOT_SELECTED_DB2_OBJ:
(hex value 0x00000002)
Input objects that have not been processed, because they did not meet the selection criteria will be written to the information log file.

INFO_ROWS_AFFECTED_NE_1:
(hex value 0x00000004)
UPDATE and DELETE commands with a key definition in the WHERE clause that have not affected exactly one single row will be written to the information log file.

INFO_BC_DUP_KEY_VALUE:
(hex value 0x00000008)
Duplicate key values detected during a batch compare will be written to the information log file.

INFO_GENERAL_INFO:
(hex value 0x00000010)
General information will be written to the information log file.

INFO_IDMS_NO_AREA:
(hex value 0x00000020)
IDMS data records with no AREA reference will be written to the information log file.

INFO_NO_START_LUW:

(hex value 0x00000040)
: UPDATEs without a corresponding LUW start record are logged to the info file.

INFO_CORRECTINVALID:
(hex value 0x00000080)
: Invalid data will be corrected and logged to the INFO file. This includes character data containing values less than x'40', invalid packed values, invalid date and time fields, etc.
If this option is not specified, the processing is terminated when invalid data has been detected for packed values and date and time fields. For character data the values will be propagated unchanged[2].

INFO_REPODISCARDED:
(hex value 0x00000100)
: Data that has been rejected because of processing rules is logged to the info file. The processing rules are defined in the repository.

Example:

```
Output_Info_Flags = INFO_GENERAL_INFO
```

[2] The specification of 'Use bound parameter' is mandatory for character strings that may contain LOW-VALUE.

2.96 Output_Info_Seg_Size

The parameter Output_Info_Seg_Size controls the segmentation of information protocol files. The parameter value is a number in MB that specifies the maximum size of a protocol file in MB. If the current protocol file exceeds this size, it is closed and a new segment is opened.

Example:

```
Output_Info_Seg_Size = 10
```

This parameter is optional. If it is not defined, segmentation will not be performed.

2.97 Output_Info_Seg_Compress

If information protocol files are to be segmented (see parameter Output_Info_Seg_Size), this parameter controls the compression of file segments.
The parameter value is 0 (no compression of files) or 1 (compression activated).

Example:

```
Output_Info_Seg_Compress = 1
```

This parameter is optional. If it is not defined, compression will not be performed. This parameter has no effect on z/OS systems.

2.98 QUALIFIER_SCHEMA

The parameter QUALIFIER_SCHEMA specifies the schema name that should be used for processing. The parameter value overwrites the specification of '=schema=' in the output table definition of the repository.

Example:

```
QUALIFIER_SCHEMA = 'MYSCHEMA'
```

2.99 QUALIFIER_CREATOR

The parameter QUALIFIER_CREATOR specifies the creator name that should be used for the processing. The parameter value overwrites the specification of '=creator=' in the output table definition of the repository.

Example:

```
QUALIFIER_CREATOR = 'MYCREATOR'
```

2.100 JCLSKEL

The parameter JCLSKEL specifies the name of a mainframe JCL job that should be used to execute a script (submit). The JCL is stored on the Virtual Disk of tcVISION. A typical example is a

BULK_TRANSFER for IMS or DL/I using a BMP job. The JCL can contain variables that are replaced with the current values when the job is submitted. The following variables are available:

<MODULE>	Name of a tcSCRIPT module depending on the code page used (TCSCRIPT or TCSCRIPTU).
<SCRIPTNAME>	Name of the script
<AGENTLINK>	Connect string to the tcVISION Agent
<PARMS>	Parameters passed to the script

Example z/OS - Start of an external script:

```
//PMHSCRP JOB ,'TCSCRIPT',CLASS=A,MSGCLASS=A
//STEP1    EXEC PGM=<MODULE>,REGION=0M
//STEPLIB  DD DISP=SHR,DSN=TCVISION.V500.LOADLIB
//         DD DISP=SHR,DSN=SERVICE.TVSM500D.LOADLIB
//         DD DISP=SHR,DSN=IMS910.SDFSRESL
//         DD DISP=SHR,DSN=DSN810.SDSNLOAD
//TCCMDSK  DD DISP=SHR,DSN=TVSM.V500PMH.DISK.RRDS
//STDOUT   DD SYSOUT=*
//STDERR   DD SYSOUT=*
//STDTRC   DD SYSOUT=*
//STDENV   DD *
RT_NT=0
RT_NA=0
RT_TI=0
//SYSPRINT DD SYSOUT=*
//SYSABEND DD SYSOUT=*
//STDPARM  DD *
 "TCCMDSK:CONFIG:SCRIPTS/<SCRIPTNAME>"
 AGENT=192.168.0.238:4141,EXTFLG=1
```

Example z/OS - Start of an External DL/I BMP Script:

```
//DLISCRPH JOB 'A',MSGCLASS=A,CLASS=A
//* SPECIFY THE CORRECT PSB IN THE // EXEC STMT
//STEP1    EXEC PGM=DFSRRC00,REGION=0M,
//             PARM='BMP,<MODULE>,DFHSAM04,,,,,,,,,,,IVP'
//STEPLIB  DD DISP=SHR,DSN=IMS910.SDFSRESL
//         DD DISP=SHR,DSN=TCVISION.V500.LOADLIB
//         DD DISP=SHR,DSN=SERVICE.TVSM500D.LOADLIB
//IMS      DD  DSN=IMS910.PSBLIB,DISP=SHR
//         DD  DSN=IMS910.DBDLIB,DISP=SHR
//TVSMDSK  DD DISP=SHR,DSN=TVSM.V500PMH.DISK.RRDS
//IEFRDER  DD DISP=(,DELETE),DSN=&&LOGFILE,UNIT=VIO,SPACE=(CYL,(1,1,1))
//PRINTDD  DD SYSOUT=*
//DFSVSAMP DD DSN=IMS910.PROCLIB(DFSVSMDC),DISP=SHR
//STDOUT   DD SYSOUT=*
//STDERR   DD SYSOUT=*
//STDTRC   DD SYSOUT=*
//STDENV   DD *
RT_NA=1
RT_NT=1
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//STDPARM  DD *
 "TVSMDSK:CONFIG:SCRIPTS/<SCRIPTNAME>"
 AGENT=192.168.0.238:4141 EXTFLG=1 TCPIP_NAME=TCPIP
```

2.101 Output_Info_Flags_Ext

The parameter Output_Info_Flags_Ext defines special treatments for information that has been written into the information protocol file.

The following values are defined and can be specified individually or linked:

INFO_SILENCE_ON_ALL_ZEROES:
(hex value 0x00000001)
After the automatic correction of invalid field contents, fields with a former content of LOW-VALUE are not written to the information log.

2.102 Output_Target_Selection

The parameter Output_Target_Selection defines the output targets that should be used as replication selection for this script execution. Without this parameter all output targets defined in the repository for the input source are used.

The parameter value is a character string consisting of a comma separated list of output target names.

Example:

```
Output_Target_Selection = "TARGET A,TARGET B"
```

2.103 Output_Target_Deselection

The parameter Output_Target_Deselection defines the output targets that should not be used as replication selection for this script execution. Without this parameter all output targets defined in the repository for the input source are used.

The parameter value is a character string consisting of a comma separated list of output target names.

Example:

```
Output_Target_Deselection = "TARGET C"
```

2.104 Output_Idle_Timeout

The parameter Output_Target_Idle_Timeout defines a timeout for RDBMS output targets in seconds. If the connection to the output target has been inactive for that period, the connection to the output target is closed. The connection is re-established when the target is used the next time.

Example:

```
Output_Target_Idle_Timeout = 600
```

2.105 Output_Pipe_File_Size

The parameter Output_Pipe_File_Size defines the size in KB of the pipe data file.

The parameter value is numeric.
Minimum value: 1
Maximum value: 1000000
Default value: 20000

Example:

```
Output_Pipe_File_Size = 450
```

2.106 Output_Pipe_Max_File_Nr

The parameter Output_Pipe_Max_File_Nr defines the maximum number of pipe data files.

The parameter value is numeric.
Minimum value: 10
Maximum value: 9999999
Default value: 9999999

Example:

```
Output_Pipe_Max_File_Nr = 100
```

2.107 Output_Pipe_Keep_Expired_Files

The parameter Output_Pipe_Keep_Expired_Files defines the number of pipe files that are not logically required anymore but have not been removed and are further available for possible analysis purposes.
The parameter value is numerical.
Minimum value: 0
Maximum value: Output_Pipe_Max_File_Nr
Default value: 0

Example:

```
Output_Pipe_Keep_Expired_Files = 10
```

2.108 Output_Pipe_Expiration_Min

The parameter Output_Pipe_Expiration_Min defines the number of minutes backwards from the current time. Younger pipe data is never logically deleted in the pipe.
A value of 0 means that data should be deleted by different criteria.

The parameter value is numeric.
Minimum value: 0
Maximum value: 5256000
Default value: 0

Example:

```
Output_Pipe_Expiration_Min = 100
```

2.109 Output_Pipe_Zip_Min

The parameter Output_Pipe_Zip_Min defines the number of minutes from the current time. Older pipe data is saved to the pipe in compressed format.
A value of 0 means that no data is compressed.

The parameter value is numeric.
Minimum value: 0
Maximum value: 5256000
Default value: 0

Example:

```
Output_Pipe_Zip_Min = 1440
```

2.110 Output_Pipe_FSYNC

The parameter Output_Pipe_FSYNC defines the use of the fsync() call. fsync() can be used if the pipe data consistency has to be ensured even in the event of a server crash/power failure, etc. The use of fsync() can cause considerable throughput losses depending on the data storage used and must therefore be tested in advance.

A value of 0 means that fsync() is not called.
A value of means that fsync() is called.
The parameter value is numeric.
Minimal value: 0
Maximal value: 1
Default value: 0

Example:

```
Output_Pipe_FSYNC = 1
```

2.111 OUTPUT_BLOCK_STORAGELIMIT = [n|nKB|nMB|nGB]

For mainframe pool sending scripts:

Each sent data block is saved so that it can be used in case of a restart. If the number of saved blocks exceeds the storage capacity, this parameter can be used to specify the maximum number of blocks sent without reconfirmation. If this value is exceeded, the sending – hence the intermediate saving – of new blocks will be delayed until the processing of old blocks has been confirmed from the receiver and the storage usage is below the specified limit. A specification of 0 (default) deactivates this mechanism.

2.112 OUTPUT_STORAGELIMIT_MESSAGE_MIN_TIME = n (milli seconds)

As soon as the previously described mechanism is activated, the delayed sending of data results in console message TCS0239W. The message will be issued after the wait time has exceeded the specified time frame. A specification of 0 (default) deactivates the display of the messages.

2.113 OUTPUT_STORAGELIMIT_MESSAGE_MAX_NUMBER = n

The logging of those wait times is terminated after n occurrences of the message. This is indicated by the message TCS0240W. A new logging of the wait times takes place after a new start of the sending script.

2.114 LUW_AUDIT_LOG

The parameter LUW_AUDIT_LOG defines an optional log file. The file contains detailed processing information about a logical unit of work (LUW).

Example:

```
LUW_AUDIT_LOG=C:\log\audit.log
```

2.115 LUW_Manager

The parameter LUW_Manager defines the desired options for the LUW Manager. The parameter will only be processed for Output_Stage >= 1.

During the transition of stage 0 to a higher stage the LUW Manager should be activated. If multiple scripts are participating in the processing, the LUW Manager should only be activated at the level where the transition takes place from stage 0 to a higher stage.

The parameter value is numeric.

The following values are defined and can be specified in combination:

LUW_PROCESSING_ACTIVE:
(hex value 0x00000001)
: All data belonging to an LUW is being processed as package. The data will be saved temporarily and processed in a batch at the end of the transaction. The options described next are only active when LUW_PROCESSING_ACTIVE has also been specified.

LUW_TO_FILE:
(hex value 0x00000002)
: For Output_Type = 'FILE' the data that belongs to an LUW will be written to a new output file. The file name will be dynamically generated from the pattern defined in parameter Output_Target_Name. The placeholder <luwname> will be replaced by the actual name of the LUW. Also refer to the description of parameter Output_Target_Name.
Example:

```
Output_Target_Name  = 'C:\LUWs\<luwname>.bin'
```

This parameter is optional for Function = 'CDC'.

2.116 LUW_StorageThreshold

If this parameter has been specified and the storage requirements of all data saved by the LUW Manager exceeds this value, the data of the LUW is saved to DataSpace areas for mainframe systems. On workstation systems the data is saved to files. Swapped out LUW data is no longer included in the storage calculation.
The size can be specified with a numerical value followed by “KB”, “MB”, or “GB” (kilobyte, megabyte, gigabyte). The default value is 500KB.

2.117 LUW_StorageLimit

If this parameter has been specified and the total storage requirements of all LUW Manager data exceeds this value, the processing is terminated with an error message. If the parameter LUW_StorageThreshold has been specified, the parameter LUW_StorageLimit defines the sizes of the DataSpace areas per LUW.
The size can be specified with a numerical value followed by “KB”, “MB”, or “GB” (kilobyte, megabyte, gigabyte).

2.118 LUW_StoragePagePath

This parameter specifies the place where the swapped out LUW data will be saved on workstation systems.

2.119 PARALLEL_APPLY_MAX_SLOTID

The 'Parallel Apply' method can be used for the following two types of processing:
1.) CDC processing with activated LUW manager and output to a pipe in stage 1.
2.) Bulk processing.

The parameter PARALLEL_APPLY_MAX_SLOTID = n determines the maximum SlotID for the 'Parallel Apply' procedure.
Each LUW or each bulk record is assigned a SlotID from 1 to the maximum SlotID using the 'round robin' procedure.
Possible input values are from 0 to 255.
If the parameter is not specified or specified with n = 0, then all LUWs or bulk records receive the SlotID = 0.
If it is specified with n = 1, all LUWs or bulk records receive the SlotID = 1.

2.120 PARALLEL_APPLY_CHECK_SLOTID

The parameter PARALLEL_APPLY_CHECK_SLOTID determines the SlotIDs to be processed for the 'Parallel Apply' procedure:
1.) LUWs the pipe reader reads from the input pipe (stage 1, LUW manager active).
2.) Bulk records for bulk processing.

Several SlotIDs or ranges can be specified - each separated by a comma.
If the parameter is not specified or is specified with an empty string, no selection is made and all SlotIDs are processed.

Example:
```
PARALLEL_APPLY_CHECK_SLOTID = '1,2,4-7,9'
```

2.121 INST_SUFFIX

The INST_SUFFIX parameter allows instantiation of a script name with a suffix to run multiple parallel instances of a script.
This parameter is a string and is normally passed as a dynamic start parameter or created dynamically in the Prolog.
The parameter value is appended to the script name and the script starts under this instantiated script name.

2.122 SMF_Selection

This parameter selects the SMF data records that should be processed. Multiple tables can be specified separated by semicolon or comma.
During the pre-processing (output stage 1) only records are selected from the log data or image copies that are relevant for the specified tables.

The format is `'[+,-][RecTyp[-ToRecTyp][(SubRecTyp[-ToSubRecTyp][,SubRecTyp[-ToSubRecTyp]])]];...`

Examples:
```
SMF_SELECTION = '+30(1);'
      Select RecordType=30 and SubRecordType=1

SMF_SELECTION = '+1-255;'
      Select RecordType=1-255, all   SubRecordTypes

SMF_SELECTION = '+1-255;-3-5;'
      Select RecordType=1-255, without RecordTypes 3,4, and 5

SMF_SELECTION = '+100-102(+1-4);'
      Select RecordType=100, 101, 102, but only SubRecordTypes 1- 4

SMF_SELECTION = '+100-102(+1,+4);'
      Select RecordType=100, 101, 102, but only SubRecordTypes 1& 4
```

2.122.1 SMF_IFCID_Selection

This parameter additionally selects the SMF-Db2 data records based on the IFCID. Multiple IFCIDs separated by a semicolon can be specified.

The format is `'[+,-][IFCID[-ToIFCID];...`

Examples:
```
SMF_IFCID_SELECTION = '+140-144;'
      Select IFCID 140 – 144 (Auditing)
```

2.123 Start_Remote_Script

The parameter Start_Remote_Script defines the name of the remote receiver script that should be started automatically.
This parameter can be used for Output_Type = 'REMOTE' and if parameter Output_Comm_Token is used.
The remote receiver script will automatically receive parameter Input_Comm_Token as start parameter with the current value of Output_Comm_Token.
The parameter is a character string.

Example:
```
Start_Remote_Script = 'My Receiver Script'
```

2.124 Schedule_Script

Schedule_Script defines the name of the script described with option OUTPUT_FLAGS_SCHEDULE_SCRIPT.
The parameter value is a character string.

Example:
```
Schedule_Script = 'LUW_Script'
```

This parameter is mandatory and must be specified if OUTPUT_FLAGS_SCHEDULE_SCRIPT is active.

2.125 Structure_Repository

Structure_Repository defines the repository that contains the structure information.
The parameter is a character string.

Example:
```
Structure_ Repository = 'OCI       DSN=//192.168.0.73:1521/DBL10'
```

The first 10 digits define the type of data source (ODBC|DRDA|DB2|OCI|AGENT). Starting from position 11 follow the required connection string information. If this parameter is omitted, the repository defined at the controlling Agent will be used.

2.126 Structure_Prefix

The parameter Structure_Prefix defines the group for the repository information.
The parameter value is a character string.

Example:
```
Structure_Prefix = 'GROUP'
```
If this parameter is not specified, group 'DEFAULT' will be used.

2.127 Structure_Slots

The parameter Structure_Slots specifies the number of cache slots for structure information.
The cache allows the multiple use of structure information without re-reading them.

The parameter value is a numeric value between 1 and 1000. A specification of 0 deactivates the caching of structure information.

Example:
```
Structure_Slots = 100
```

2.128 Structure_Target_Selection

For an environment where CDC data must be distributed to multiple output targets, parameter Structure_Target_Selection can be used to only select certain targets for the current replication. The specification is optional. The parameter format is a comma separated list of output targets.

2.129 Structure_Target_Deselection

For an environment where CDC data must be distributed to multiple output targets the parameter Structure_Target_Deselection can be used to deselect certain targets for the current replication. The specification is optional. The parameter format is a comma separated list of output targets.

2.130 Log_Print_File

The parameter Log_Print_File defines an optional protocol file where data source-specific information will be logged during processing from stage 0 to stage 1.

Example:
```
Log_Print_File = 'YES'
```

2.131 DB2_Start_LSN

The parameter DB2_Start_LSN specifies the start LSN when processing log files for Db2 LUW for Linux, UNIX, and Windows.
The parameter value is a hexadecimal character string in the length of 16 bytes.

Example:
```
DB2_Start_LSN = '000000000000F000'
```

This parameter must be specified for Input_Source_Type = 'DB2L_API' or can be specified if the data is received in the internal stage 0 format from a remote data script.

2.132 DB2_End_LSN

The parameter DB2_End_LSN specifies the highest end LSN (Log Sequence Number) when processing Db2 LUW for Linux, UNIX, and Windows log files.
The parameter value is a hexadecimal character string in the length of 16 bytes.

Example:
```
DB2_End_LSN = '00000000000BF000'
```

This parameter can be specified for Input_Source_Type = 'DB2L_API' or if the data is received in the internal stage 0 format from a remote data script.

2.133 DB2_Start_LRI

The parameter DB2_Start_LRI specifies the start LRI for the processing of Db2 LUW for Linux, UNIX, and Windows V10.1+ log files.
The parameter value is a hexadecimal character string in the length of 16.16 bytes.

Example:
```
DB2_Start_LRI = '0000000000006F81.000000000007ADCC'
```

This parameter can be specified for Input_Source_Type = 'DB2L_API' or if the data is received in the internal stage 0 format from a remote data script.

2.134 DB2_End_LRI

The parameter DB2_End_LRI defines the end LRI for the processing of Db2 LUW for Linux, UNIX, and Windows V10.1+ log files.
The parameter value is a hexadecimal character string in the length of 16.16 bytes.

Example:
```
DB2_End_LRI = '0000000000006F81.0000000000800000'
```

This parameter can be specified for Input_Source_Type = 'DB2L_API' or if the data is received in the internal stage 0 format from a remote data script.

2.135 DB2_Start_RBA

The parameter DB2_Start_RBA defines the start RBA that is needed when reading Db2 log files. When Db2 z/OS data sharing is used, this is the start LRSN.
The parameter value is a hexadecimal character string of 12 bytes.

Example:
```
DB2_Start_RBA = '00000000F000'
```

This parameter must be specified if Input_Type = 'DB2_LOGREC' or can be specified if the input data comes from a remote data script that outputs the data in internal format in stage 0.

2.136 DB2_End_RBA

The parameter DB2_End_RBA defines the end RBA that might be needed when reading Db2 log data. If Db2 z/OS data sharing is used, this is the end LRSN.
The parameter value is a hexadecimal character string of 12 bytes.

Example:
```
DB2_End_RBA = '0000000BF000'
```

This parameter must be specified if Input_Type = 'DB2_LOGREC' or can be specified if the input data comes from a remote data script that outputs the data in internal format in stage 0.

After successful processing of the Db2 active log, the parameter PM_O.DB2_End_RBA will contain the RBA/LRSN that corresponds to the highest used RBA/LRSN value from Db2 (current Db2-RBA/LRSN EOF condition + 1). As an example, this value can be saved in the epilog of the script. During the next start of the script it can be read in the script prolog and be used as new start RBA/LRSN. This ensures that processing continues starting from the last processed RBA/LRSN.

2.137 DB2_LRSN_DELTA

The parameter DB2_LRSN_DELTA defines an LRSN delta when reading datasharing Db2 z/OS log data if the log data is read from an archive log and the data script has no possibility to read the LSRN delta directly from the IFI 306 interface.
This parameter overwrites the LSRN delta from the IFI 306 interface for processing.
The parameter value is a hexadecimal string of 20 bytes.

Example Db2 V11+:
DSNJU004 output: STCK TO LRSN DELTA 0005C51F37D8A2000000

```
DB2_End_RBA = '0005C51F37D8A2000000'
```

Example Db2 V10:
DSNJU004 output: STCK TO LRSN DELTA 05C51F37D8A2

```
DB2_End_RBA = '0005C51F37D8A2000000'
```

2.138 DB2_START_LRSN_DELTA

The parameter DB2_START_LRSN_DELTA defines at setup for the IFI 306 LogReader during datasharing whether the user-defined start and end LRSN is effectively a timestamp which must be added to the LRSN_Delta before setup to get a correct start and end LRSN.

The parameter value is a number with the value 0 or 1.

If the value is 1, the addition is executed.

2.139 ORACLE_Start_SCN

The parameter ORA_Start_SCN defines the start SCN that is needed when reading Oracle log files (Linux, UNIX, and Windows).

The parameter value is a hexadecimal character string of 12 bytes.

Example:

```
ORACLE_START_SCN = '0000009419C0'
```

This parameter may be used with Input_Source_Type = 'ORA_LOGM' or if the input data comes from a remote data script that sends the data in the internal stage 0 format.

2.140 ORACLE_End_SCN

The parameter ORA_End_SCN defines the end SCN that might be needed when reading Oracle log data for Linux, UNIX, and Windows.
The parameter value is a hexadecimal character string of 12 bytes.

Example:
```
ORACLE_END_SCN = 'FFFFFFFFFFFF'
```

This parameter may be used with Input_Source_Type = 'ORA_LOGM' or if the input data comes from a remote data script that sends the data in the internal stage 0 format.

After successful processing of the log data, the parameter PM_O.ORA_End_SCN will contain the SCN that corresponds to the highest used SCN value (current SCN EOF condition + 1). As an example, this value can be saved in the epilog of the script. During the next start of the script it can be read in the script prolog and be used as new start SCN. This ensures that processing continues starting from the last processed SCN.

2.141 MSSQL_Start_LSN

The parameter MSSQL_Start_LSN defines the start LSN that is required when processing MS SQL Server.
The parameter value is a hexadecimal character string that consists of three fields, all separated by a ':': File_ID, Block_ID, and Slot_Id.

Example:
```
MSSQL_START_LSN = '00000018:00000062:0028'
```

This parameter can be specified for Input_Source_Type = 'MSSQL_FILE', Input_Source_Type = 'MSSQL_CDC' or if the input data comes from a remote data script that sends the data in the internal stage 0 format.

2.142 MSSQL_End_LSN

The parameter MSSQL_End_LSN defines the end LSN when processing MS SQL Server log data.
The parameter value is a hexadecimal character string that consists of three fields, all separated by a ':': File_ID, Block_ID, and Slot_Id.

Example:
```
MSSQL_END_LSN = '00000018:00000062:0028'
```

This parameter can be specified for Input_Source_Type = 'MSSQL_FILE' or if the input data comes from a remote data script that sends the data in the internal stage 0 format.

After successfully processing the log data, the script sets the value of the parameter PM_O.MSSQL_End_LSN to an LSN that is equal to the highest used LSN.

2.143 MSSQL_SP_REPLDONE

The parameter MSSQL_SP_REPLDONE specifies the number of COMMITS after which a 'EXEC sp_repldone ...' should be executed (with the corresponding LSN). This is performed during the processing of online MS SQL Server log files.
The parameter is a numeric value between 0 and 9999.

Example:
```
MSSQL_SP_REPLDONE = 10
```

This parameter can be specified with Input_Source_Type = 'MSSQL_LOGREC'.

If objects have been defined in a publication for a database, a pointer to the last propagated LSN is maintained in the 'distribution' database. With tcVISION as the only propagation this pointer is not automatically maintained and results in an undesired growth of the log files, because the SQL Server cannot perform a 'log truncation' due to transactions that are supposedly not propagated yet.

If the parameter MSSQL_SP_REPLDONE has a value of n > 0, the script always performs a 'EXEC sp_repldone ...' after the complete retrieval of n transactions with objects defined in a publication. The result is that MS SQL Server now can perform the necessary 'log truncation', e.g. during 'BACKUP LOG...'.

Possible messages and errors during the execution of 'EXEC sp_repldone ...' will only be written to the info file specified in Output_Info_Name. They do not prematurely terminate the processing.

2.144 MYSQL_START_POS

The parameter MYSQL_START_POS defines the desired start position when reading MySQL CDC data.
The parameter value is a two-part number in the format 'f,p', whereas f is the log file number and p is the position in the respective log file.

Example:
```
MYSQL_START_POS = '5,0'
```

This parameter can be specified if Input_Source_Type = 'MYSQL_LOGREADER'.

2.145 MYSQL_END_POS

The parameter MYSQL_END_POS defines the desired stop position when reading the MySQL CDC data.
The parameter value is a two-digit number in the following format 'f,p'. f represents the log file number, p represents the position in the log file.

Example:
```
MYSQL_END_POS = '6,0'
```

This parameter is valid if Input_Source_Type = 'MYSQL_LOGREADER' has been specified.

2.146 MARIADB_START_POS

The parameter MARIADB_START_POS defines the desired start position when reading MariaDB CDC data.
The parameter value is a two-part number in the format 'f,p', whereas f is the log file number and p is the position in the respective log file.

Example:
```
MARIADB_START_POS = '5,0'
```

This parameter can be specified if Input_Source_Type = 'MARIADB_LOGREADER'.

2.147 MARIADB_END_POS

The parameter MARIADB_END_POS defines the desired stop position when reading the MariaDB CDC data.
The parameter value is a two-digit number in the following format 'f,p'. f represents the log file number, p represents the position in the log file.

Example:
```
MARIADB_END_POS = '6,0'
```

This parameter is valid if Input_Source_Type = 'MARIADB_LOGREADER' has been specified.

2.148 POSTGRESQL_Start_LSN

The parameter POSTGRESQL_Start_LSN defines the start LSN when reading PostgreSQL CDC data.
The parameter is a two-part string in the format x/x whereas x is a 32-bit hexadecimal number.

Example:
```
POSTGRESQL_Start_LSN = '0/01EBB148'
```

This parameter can be specified if PM_I_Input_Source_Type = 'POSTGRESQL_LOGREADER'.

2.149 POSTGRESQL_End_LSN

The parameter POSTGRESQL_End_LSN defines the end LSN when reading PostgreSQL CDC data.
The parameter is a two-part string in the format x/x whereas x is a 32-bit hexadecimal number.

Example:
```
POSTGRESQL_Start_LSN = '0/01EBB148'
```

This parameter can be specified if PM_I_Input_Source_Type = 'POSTGRESQL_LOGREADER'.

2.150 POSTGRESQL_AMEND_LSN_GAP

When reading PostgreSQL CDC data, the POSTGRESQL_AMEND_LSN_GAP parameter defines the maximum allowed gap between the slot-restart-lsn and the current system-wal-lsn.

Such a gap can occur when a CDC process does not receive CDC data from its associated database for a long time, but at the same time the system-wal-lsn of processes on other databases is increased. As soon as this gap becomes too large, archiving of the log files is no longer possible.

The parameter value is a decimal number with values between 0 and 2147483647. Default value is 16777215.

A value of 0 disables this function.

A value that is too low will result in unnecessarily excessive communication between the CDC process and the database server.

Example:

```
POSTGRESQL_AMEND_LSN_GAP = 1024
```

This parameter can be specified, if Input_Source_Type = 'POSTGRESQL_LOGREADER'.

2.151 TABLE_SELECTION

The generic parameter TABLE_SELECTION specifies the names of the tables for the processing of SQL data.
Multiple tables separated by semicolon or comma can be specified.
During the pre-processing stages (stage 0 and 1) only data relevant for the specified tables is processed if possible.

The format is `creator1,+tname1,+tname2,...;creator2,+tname3;....`

The first value is the creator of the table followed by table names that should be processed for this creator. The table names are separated by commas. By using a + or - sign in front of the table name tables can be selected or deselected. Multiple creators can be separated by a semicolon and multiple table names within a creator are separated by commas. Table names can also specified generically. An example would be +SYS*.

Example:

```
TABLE_SELECTION = 'SYSIBM,+SYSTABLES,+SYSCOLUMNS;'
TABLE_SELECTION = 'SYSIBM,+SYS*,-SYSTABLES;'
```

This parameter can be specified for Input_Source_Type = 'IFI_306', 'ORA_LOGM', 'DB2L_API', 'DB2_ICOPY', 'DB2V_FILE', 'DB2M_FILE', 'MSSQL_FILE', 'MSSQL_CDC', or if the input data comes

from a remote data script that reads the data from the corresponding source and transfers the data in stage 0 or stage 1 format.
In this case the parameters 'DB2_SELECTION', 'ORACLE_SELECTION', and 'MSSQL_SELECTION' are no longer required.

2.152 DB2_SELECTION

The parameter DB2_SELECTION defines the tables that should be processed when reading Db2 log files and Db2 image copy data.

Multiple tables can be specified, separated by semicolon or comma.

During the pre-processing (output stage 1) only the records from the data or image copies relevant to the specified tables are retrieved.

The format is `creator1,+tname1,+tname2,...;creator2,+tname3;....`

The first value is the creator of the table and the following values (separated by commas) are the desired table names of that creator. Tables can be selected or deselected with a + (plus) or – (minus) sign. Multiple creators can be separated by a semicolon and multiple tables within a creator are separated by a comma. Table names can also be generically defined, e.g. +SYS*.
If this parameter is not specified, the script reads the Db2 tables that should be selected with the database name from Input_DatabaseName from the repository.

Example:

```
DB2_SELECTION = 'SYSTEM,+SYSCATALOG,+SYSCOLUMNS;'
DB2_SELECTION = 'SYSTEM,+SYS*,-SYSCATALOG;'
```

2.153 DB2_XML_STRING_RESYNC

The parameter DB2_XML_STRING_RESYNC defines the syncronization between the SYSIBM.SYSXMLSTRINGS and the repository table DB2XMLStrings when reading Db2 XML log data and Db2 imagecopy data.

The parameter value is numeric with the values 0 – 4 and is only used for CDC scripts.
The values have the following meaning:

0:
No synchronization

1:
The two tables are synchronized if the max(Stringid) in the repository table is smaller than in SYSIBM.SYSXMLSTRINGS. Then the CDC processing is continued.

2:
Both tables are synchronized and the CDC processing is continued.

3:
The two tables are synchronized if the max(Stringid) in the repository table is smaller than in SYSIBM.SYSXMLSTRINGS. Then the processing is finished.

4:

The two tables are synchronized and then the processing is finished.

The default value is 1.

2.154 ORACLE_SELECTION

The parameter ORACLE_SELECTION defines the tables that should be processed when reading log files for Oracle for Linux, UNIX, and Windows.

Multiple tables can be specified, separated by semicolon or comma.

During the pre-processing (output stage 1) only the records relevant to the specified tables will be retrieved from the log data.

The format is `creator1,+tname1,+tname2,...;creator2,+tname3;....`

The first value is the creator of the table and the following values (separated by commas) are the desired table names of that creator. Tables can be selected or deselected with a + (plus) or – (minus) sign. Multiple creators can be separated by a semicolon. Multiple tables within a creator are separated by a comma. Table names can also be generically defined, e.g. +SYS*.

Example:

```
ORACLE_SELECTION = 'SYSTEM,+ALL_TAB,+MUL_TAB;'
ORACLE_SELECTION = 'SYSTEM,+MUL*,-MUL_TAB;'
```

This parameter may be used with Input_Source_Type = ''ORA_LOGM', Input_Type = 'ORACLE_LOGREC', or if the input data comes from a remote data script that sends the data in the internal stage 0 format.

2.155 MSSQL_SELECTION

The parameter MSSQL_SELECTION specifies the tables that should be processed for the MS SQL Server log data.
Multiple tables can be specified, separated by a semicolon or comma.
During the processing (output stage 1) only data relevant to the specified tables is forwarded.

The format is `creator1,+tname1,+tname2,...;creator2,+tname3;....`

The first value is the creator of the table followed by the tables for this creator. The tables are separated by a comma. Using a + or – sign tables can be selected or deselected. Multiple creators can be specified separated by a semicolon, multiple tables per creator separated by a comma. Table names can also be generically specified, e.g. +SYS.

Example:

```
MSSQL_SELECTION = 'dbo,+*;'
```

This parameter can be specified for Input_Type = 'MSSQL_LOGREC', Input_Source_Type = 'MSSQL_FILE', or when the input data comes from a remote data script that sends the data in the internal stage 0 format.

2.156 DATABASE_CONNECTSTRING

The parameter DATABASE_CONNECTSTRING specifies the connect string to the MS SQL Server when calling the extended stored procedure 'tcVISION_Logreader_xp.dll'.
This procedure is required to process active log data.

2.157 TCPIP_Name

The parameter TCPIP_Name defines additional information for the local TCP/IP stack and depends on the actual runtime environment.
The parameter value is a character string

Example for z/OS:

```
TCPIP_Name = 'TCPIP'
```

If this parameter has been defined, it will be used for all TCP/IP operations.

If this parameter is not defined, the value specified for the controlling Agent will be used.

2.158 TCPIP_Threaded

The parameter TCPIP_Threaded defines whether the TCP/IP traffic between process parts should be handled in separate threads, so that waiting times for e.g. file accesses can be used for sending already read data.

The parameter value is numeric:

0	Threading not enabled
1	Threading enabled

Example:

```
TCPIP_Threaded = 1
```

This parameter can be defined for individual process parts.

Without specification the TCP/IP traffic is handled in the main thread.

2.159 TCPIP_Channels

The parameter TCPIP_Channels defines how many connections should be opened for TCP/IP traffic between process parts. It is used only if TCPIP_Threaded is enabled.

The parameter value is numeric. Values from 1 to 99 (inclusive) are allowed.

Example:

```
TCPIP_Channels = 15
```

This parameter can be defined for individual process parts.

Without specification the TCP/IP traffic is handled with one connection.

2.160 TCPIP_Blocking_Mode

The parameter TCPIP_Blocking_Mode determines the maximum size of packets for TCP/IP traffic between process parts.

The parameter value is numeric;
- 0 Packet size is determined by the internal logical record size.
- 1 Packet size is determined by the TCP/IP stack reported SendBuffer size.

Example:
```
TCPIP_Blocking_Mode = 1
```

If not specified, the internal logical record size is used.

2.161 QMmgrName

The parameter QMmgrName defines the name of the optional WEBSPHERE_MQ Manager name that should be used for a connection to the WEBSPHERE_MQ Manager.
The parameter value is a character string.

Example for z/OS:
```
QMmgrName = 'MyManager'
```

This parameter is optional for Input_Type or Output_Type = 'WEBSPHERE_MQ'.

2.162 HP_Exit_Name

HP_Exit_Name defines the name of an external high-performance load module that contains user-defined exit functions in compiled format. The name must not have operating system-specific suffixes like '.dll', '.so', etc.
Also refer to chapter 4.3 HP Exit.

Example:
```
HP_Exit_Name = 'USREXIT'
```

2.163 Log_Rec_Size

The parameter Log_Rec_Size defines the maximum length of logical or physical data records.
The parameter value is numeric and has a default value of 66000 bytes.

Example:
```
Log_Rec_Size = 150000
```

2.164 SQL_DA_Size

The parameter SQL_DA_Size defines the maximum length of internal SQL structure tables.
The parameter value is numeric and has a default value of 66000 bytes.

Example:
```
SQL_DA_Size = 50000
```

2.165 SQL_DML_Size

The parameter SQL_DML_Size defines the maximum length of SQL statements.
The parameter value is numeric and has a default value of 66000 bytes.

Example:
```
SQL_DML_Size = 50000
```

2.166 SQL_COMMENT_Size

The parameter SQL_COMMENT_Size defines the maximum length of SQL comments.
The parameter value is numeric and has a default value of 2000 bytes.

Example:
```
SQL_COMMENT_Size = 5000
```

2.167 DML_Array_Size

The parameter DML_Array_Size specifies the maximum number of data records in a block that is propagated to the target database. The parameter is activated with the option OUTPUT_FLAGS_ARRAY_DML. The parameter is numeric and the default value is 30.

Example:
```
DML_ARRAY_SIZE = 200
```

2.168 Decimal_Point

The parameter Decimal_Point defines the character that should be used as the decimal point. Default is '.'.

Example:
```
Decimal_Point = ','
```

2.169 Quote_Char

The parameter Quote_Char defines the character that should be used to enclose text variables for the output. Default is no character.

Example:
```
Quote_Char = '"'
```

2.170 Delimiter_Char

The parameter Delimiter_Char defines the character that separates variables in the output. Default is no character.
This parameter is only used if bit OUTPUT_FTYPE_SQL_LOADER in parameter Output_File_Type has been set.

Example:

```
Delimiter_Char = ','
```

2.171 Max_Input_Blocks

The parameter Max_Input_Blocks defines the maximum number of input data blocks that should be read. Processing terminates normally when this threshold has been reached.
A value of 0 allows any number of blocks.
Default is 0.

Example:

```
Max_Input_Blocks = 5
```

2.172 Max_Input_Records

The parameter Max_Input_Records defines the maximum number of input data records. Processing terminates normally when this threshold has been reached.
A value of 0 allows any number of blocks.
Default is 0.

Example:

```
Max_Input_Records = 5
```

2.173 NULL_Date

The parameter NULL_Date defines the default value for output fields with an attribute of DATE. This value is used if no input value is available.

Default is `'1970-01-01'`.

Example:

```
NULL_Date = '2010-01-18'
```

2.174 NULL_Time

The parameter NULL_Time defines the default value for output fields with an attribute of TIME. This value is used if no input value is available.

Default is `'00.00.00'.`

Example:

```
NULL_Time = '14.15.36'
```

2.175 NULL_Timestamp

The parameter NULL_Timestamp defines the default value for output fields with an attribute of TIMESTAMP. This value is used if no input value is available.

Default is `'1970-01-01-00.00.00.000000'.`

Example:
```
NULL_Timestamp = '2010-01-18-14.15.43.000000'
```

2.176 Collect_Statistics

The parameter Collect_Statistics activates the counting of accesses (Insert, Update, Delete) for input and output objects. In addition, the control statements (Nocommit, Commit, and Rollback) are counted for the output targets. The resulting values are issued at the end of the script run in a tabular form.

Default is `0`.

Example:
```
Collect_Statistics = 1
```

2.177 AssignAutoValueStart

The parameter AssignAutoValueStart assigns the start values for the replication processing function “Insert automatic value“.

Default is 1.

Example:
```
AssignAutoValueStart = 1000
```

2.178 LOADER_CTL_FILE

The parameter LOADER_CTL_FILE contains the suffix to be used for the file that contains the loader control statements. This suffix is appended to the output file name.

Example:

```
LOADER_CTL_FILE = '.LDR'
```

2.179 DDL_Flags

The parameter DDL_Flags defines how DDL changes (i.e. structural changes in the source database) are processed. A DDL change could be, for example, adding a column to a table, changing the column data type, dropping a table, etc.

The parameter value is numeric. The following values are defined and can be specified in combination:

DDL_ACTIVATED
(hex value 0x0001)
The DDL processing is activated.

DDL_PROTOCOL
(hex value 0x0002)

Turn on protocoling of the DDL events. A DDL protocol file is created under the same naming rules as the trace or info files; its name ends with '-ddl.txt'.
Note: DDL protocoling is only available on workstations.

DDL_FORMAT_TXT
(hex value 0x0004)

DDL events are protocoled in TXT format.

DDL_FORMAT_JSON_COMPACT
(hex value 0x0008)

Protocol DDL events in JSON format; each DDL event is represented by a one-line JSON string in a compressed, non human-friendly format.

DDL_FORMAT_JSON_FORMATTED
(hex value 0x0010)

Protocol DDL events in JSON format; DDL events are represented by pretty-printed, human-readable text.

DDL_FORWARD
(hex value 0x0040)

SQL statements for DDL events will be generated and forwarded to the target.
Note1: If the target is set to REPOSITORY (i.e. Output_Type is 'REPOSITORY'), this flag is ignored and DDL forwarding is controlled via target settings.
Note2: This feature is only available on workstations.

Example:

```
DDL_Flags = DDL_ACTIVATED + DDL_PROTOCOL + DDL_FORMAT_JSON_FORMATTED
```

2.180 PM_O.RetCode

The parameter PM_O.RetCode contains the return code value of the script execution.

2.181 PM_O.RetMessage

The parameter PM_O.RetMessage contains the returned message from the script execution.

2.182 PM_O.LB_InCount

The parameter PM_O.LB_InCount contains the number of input data blocks.

2.183 PM_O.LB_OutCount

The parameter PM_O.LB_OutCount contains the number of output data blocks.

2.184 PM_O.LR_InCount

The parameter PM_O.LR_InCount contains the number of input data records.

2.185 PM_O.LR_OutCount

The parameter PM_O.LR_OutCount contains the number of output data records.

2.186 PM_O.LC_InCount

The parameter PM_O.LC_InCount contains the number of input transaction commands.

2.187 PM_O.LC_OutCount

The parameter PM_O.LC_OutCount contains the number of output transaction commands.

2.188 PM_O.LUW_Count

The parameter PM_O.LUW_Count contains the total number of processed transactions (LUWs) for the entire script execution.

2.189 PM_O.LUW_MaxCount

The parameter PM_O.LUW_MaxCount contains the number of parallel processed transactions.

2.190 PM_O.BUBC_Inserted

The parameter PM_O.BUBC_Inserted contains the number of INSERT statements generated during a batch compare processing.

2.191 PM_O.BUBC_Deleted

The parameter PM_O.BUBC_Deleted contains the number of DELETE statements generated during a batch compare processing.

2.192 PM_O.BUBC_Updated

The parameter PM_O.BUBC_Updated contains the number of UPDATE statements generated during a batch compare processing.

2.193 PM_O.BUBC_Unchanged

The parameter PM_O.BUBC_Unchanged contains the number of data records in which no changes have been detected during a batch compare processing.

2.194 PM_O.BUBC_SkipDupKey

The parameter PM_O. BUBC_SkipDupKey contains the number of data records in which identical key values have been detected during a batch compare processing.

2.195 PM_O.MemoryAllocated

The parameter PM_O.MemoryAllocated contains the value of the maximum storage allocation during the script execution.

2.196 PM_O.Trace_Output

The parameter PM_O.Trace_Output returns then name of the currently created name of the trace protocol file if one has been created.

2.197 PM_O.Output_Log_Name

The parameter PM_O.Output_Log_Name returns the currently created name of the SQL command protocol file if one has been created.

2.198 PM_O.Output_Bad_Name

The parameter PM_O.Output_Bad_Name returns the currently created name of the SQL commands in error if one has been created.

2.199 PM_O.Output_Info_Name

The parameter PM_O.Output_Info_Name returns the name of the currently created name of the information protocol file if one has been created.

2.200 PM_O.Log_Print_File

The parameter PM_O.Log_Print_File returns the currently created name of the protocol file for data source-specific information if one has been created.

3 Exit Procedures for Data Scripts

This chapter covers the predefined exit procedures for data scripts.

Exit calls will be activated either by inserting procedures with the predefined procedure name as part of the function call

```
CDC_RC = CALL_CDC()
```

or by setting the name of an ExitCode that is defined in the repository in the corresponding input variable.

Input parameters are available to all exit procedures as well as output parameters to control the subsequent processing.
The parameter name consists of a stem and a name separated by a period.

Example: `X_CS_I.ErrorMessage`

Procedures must be terminated with a RETURN instruction.
The RETURN instruction will pass information about error situations in the procedure internal processing to the calling function. The RETURN parameter consists of a return code and an optional error text that is separated by a comma.

RETURN Code	0	No error in procedure, processing continues according to additional output parameters, e.g. X_CS_O.ReturnFlags.
	<>0	Error in the procedure, processing terminates. An error message will become part of the error text of the data-script

Examples:

```
RETURN '12, Error in CDC_ExitSenderError: xxxxx'
RETURN '0'
```

The next chapter covers the predefined exit procedures and the individual input and output parameters.

3.1 CDC_ExitPostInitialization

The exit procedure CDC_ExitPostInitialization will be called after the input parameters have been processed and before the actual processing takes place.

The difference to a PROLOG exit is that a connection to the tcVISION Agent has already been established.

3.2 CDC_ExitPreTermination

The exit procedure CDC_ExitPreTermination will be called after the processing has been completed.
The difference to an epilog exit is that a connection to the tcVISION Agent still exists.

3.3 CDC_ExitStartReceiver

The exit procedure CDC_ExitStartReceiver will be called before a script is started on a remote Agent.

Parameter Stem:

Input:	X_CS_I
Output:	X_CS_O

Input Parameter:

X_CS_I.ScriptName	Name of the script that should be started

Output Parameter:

X_CS_O.ReturnFlags	Return value 0 Start script and continue processing. 1 Do not start script but continue processing.

Example:

```
CDC_ExitStartReceiver:
SAY 'About to start Script: 'X_CS_I.Scriptname

X_CS_O.ReturnFlags = 0
RETURN '0, No Error'
```

3.4 CDC_ExitSenderError

The exit procedure CDC_ExitSenderError will be called if an error occurs during establishing a connection between the sender script and the remote receiver script, and the connection terminates.

Parameter Stem:

Input:	X_CS_I
Output:	X_CS_O

Input Parameter:

X_CS_I.ErrorMessage	Error message allocation of a connection

Output Parameter:

X_CS_O.ReturnFlags	Return value 0 Terminate processing using error message in X_CS_I.ErrorMessage 1 Ignore error and continue processing

Example:

```
CDC_ExitSenderError:
SAY X_CS_I.ErrorMessage

X_CS_O.ReturnFlags = 0
RETURN '0, No Error'
```

3.5 CDC_ExitConnectSenderRetry

The exit procedure CDC_ExitConnectSenderRetry will be called BEFORE another connection attempt of the receiver script to the remote sender script is started.

Parameter Stem:

Input:	X_CS_I
Output:	X_CS_O

Input Parameter:

X_CS_I.LocalAddress	Local port for connection to a remote sender

Output Parameter:

X_CS_O.ReturnFlags	Return value 0 Continue processing 1 No more connection attempts, terminate processing

Example:

```
CDC_ExitConnectSenderRetry:
SAY 'Retry Connect to Sender:' X_CS_I.LocalAddress 'at' TIME('L')

X_CS_O.ReturnFlags = 0
RETURN '0, No Error'
```

3.6 CDC_ExitConnectSenderTimeout

The exit procedure CDC_ExitConnectSenderTimeout will be called if no connection could be started from the receiver script to a remote sender script.

Parameter Stem:

Input:	X_CS_I
Output:	X_CS_O

Input Parameter:

X_CS_I.LocalAddress	Local port for connection to remote sender

Output Parameter:

X_CS_O.ReturnFlags	Return value 0 Cancel processing with timeout 1 Start another connection attempt

Example:

```
CDC_ExitConnectSenderTimeout:
SAY 'Retry Connect to Sender:' X_CS_I.LocalAddress 'at' TIME('L')

X_CS_O.ReturnFlags = 0
RETURN '0, No Error'
```

3.7 CDC_ExitConnectSenderOK

The exit procedure CDC_ExitConnectSenderOK will be called if the connection of the receiver script to a remote sender script could be started successfully.

Parameter Stem:
Input: X_CS_I
Output: X_CS_O

Input Parameters:
X_CS_I.LocalAddress Local port for the connection with a remote sender
X_CS_I.RemoteAddress Address of the remote sender script

Output Parameter:
X_CS_O.ReturnFlags Return value
0 Continue processing
1 Terminate connection, cancel processing

Example:
```
CDC_ExitConnectSenderOK:
SAY 'Connected to Sender:' X_CS_I.RemoteAddress 'on local port'
X_CS_I.LocalAddress 'at' TIME('L')

X_CS_O.ReturnFlags = 0
RETURN '0, No Error'
```

3.8 CDC_ExitDisconnectSenderOK

The exit procedure CDC_ExitDisconnectSenderOK will be called if the connection of the receiver script to a remote sender script has been successfully disconnected.

Parameter Stem:
Input: X_CS_I
Output: X_CS_O

Input Parameters:
X_CS_I.LocalAddress Local port for the connection to the remote sender
X_CS_I.RemoteAddress Address of the remote sender script

Output Parameter:
X_CS_O.ReturnFlags Return value
0 Continue processing

Example:
```
CDC_ExitDisconnectSenderOK:
SAY 'Disconnected from Sender:' X_CS_I.RemoteAddress 'at' TIME('L')

X_CS_O.ReturnFlags = 0
RETURN '0, No Error'
```

3.9 CDC_ExitReceiverError

The exit procedure CDC_ExitReceiverError will be called if a connection attempt from the receiver script to a remote sender script terminates with an error.

Parameter Stem:
Input: X_CS_I

Output: X_CS_O

Input Parameter:
X_CS_I.ErrorMessage Error message for connection attempt

Output Parameter:
X_CS_O.ReturnFlags Return value
0 Cancel connection with error message in X_CS_I.ErrorMessage
1 Ignore error, continue processing

Example:
```
CDC_ExitReceiverError:
SAY X_CS_I.ErrorMessage

X_CS_O.ReturnFlags = 0
RETURN '0, No Error'
```

3.10 CDC_ExitConnectReceiverRetry

The exit procedure CDC_ExitConnectReceiverRetry will be called before another connection attempt is started from the sender script to a remote receiver script.

Parameter Stem:
Input: X_CS_I
Output: X_CS_O

Input Parameter:
X_CS_I.RemoteAddress Address of the remote receiver script

Output Parameter:
X_CS_O.ReturnFlags Return value
0 Continue processing
1 No more connection attempts, cancel processing

Example:
```
CDC_ExitConnectReceiverRetry:
SAY 'Retry Connect to Receiver:' X_CS_I.RemoteAddress 'at' TIME('L')

X_CS_O.ReturnFlags = 0
RETURN '0, No Error'
```

3.11 CDC_ExitConnectReceiverTimeout

The exit procedure CDC_ExitConnectReceiverTimeout will be called if no connection could be started from the sender script to a remote receiver script.

Parameter Stem:
Input: X_CS_I
Output: X_CS_O

Input Parameter:
X_CS_I.RemoteAddress Address of the remote receiver script

Output Parameter:

X_CS_O.ReturnFlags	Return value	
	0	Cancel processing with timeout
	1	Start another connection attempt

Example:

```
CDC_ExitConnectReceiverTimeout:
SAY 'Timeout while connecting to Receiver:' X_CS_I.RemoteAddress 'at' TIME('L')

X_CS_O.ReturnFlags = 0
RETURN '0, No Error'
```

3.12 CDC_ExitConnectReceiverOK

The exit procedure CDC_ExitConnectReceiverOK will be called if a connection could be successfully started from a sender script to a remote receiver script.

Parameter Stem:

Input:	X_CS_I
Output:	X_CS_O

Input Parameter:

X_CS_I.RemoteAddress	Address of the remote sender script

Output Parameter:

X_CS_O.ReturnFlags	Return value	
	0	Continue processing
	1	Terminate connection and cancel processing

Example:

```
CDC_ExitConnectReceiverOK:
SAY 'Connected to Receiver: ' X_CS_I.RemoteAddress 'at' TIME('L')

X_CS_O.ReturnFlags = 0
RETURN '0, No Error'
```

3.13 CDC_ExitDisconnectReceiverOK

The exit procedure CDC_ExitDisconnectReceiverOK will be called if the connection between the sender script and the remote receiver script could be successfully disconnected.

Parameter Stem:

Input:	X_CS_I
Output:	X_CS_O

Input Parameter:

X_CS_I.RemoteAddress	Address of the remote sender script

Output Parameter:

X_CS_O.ReturnFlags	Return value	
	0	Continue processing

Example:
```
CDC_ExitDisconnectReceiverOK:
SAY 'Disconnected from Receiver: ' X_CS_I.RemoteAddress 'at' TIME('L')

X_CS_O.ReturnFlags = 0
RETURN '0, No Error'
```

3.14 CDC_ExitUserRecord

The exit procedure CDC_ExitUserRecord will be called if a data record is expected from the exit because of parameter Input_Source_Type = 'USER_EXIT'.

Parameter Stem:

Input:	X_GL_I
Output:	X_GL_O

Input Parameter:

None

Output Parameters:

X_GL_O.StructureFileName	Name of the related structure file	
X_GL_O.Record	Data of the data record	
X_GL_O.ReturnFlags	Return value	
	0	Process record
	1	No more records (EOF)

Example:
```
CDC_ExitUserRecord:

X_GL_O.StructureFileName = 'VSAM.ARTIKEL.FILE'
X_GL_O.Record = "xxxxx"

X_GL_O.ReturnFlags = 0
RETURN '0, No Error'
```

3.15 CDC_ExitS0LogRecordRead

The exit procedure CDC_ExitS0LogRecordRead will be called after a data record of stage 0 has been read. The exit can process and also modify the data.

Parameter Stem:

Input:	X_SX_I
Output:	X_SX_O

Input Parameters:

X_SX_I.Stage	0	Always stage 0
X_SX_I.DataSourceType	Type of data source	
	1	DB2 EXIT
	2	DB2 LOGREC
	3	IMS DRA EXIT
	4	IMS EXIT
	5	IMS LOGREC

6 CICS VSAM EXIT
7 ADABAS EXIT
8 ADABAS PLOG
10 IDMS EXIT
11 BATCH COMPARE
12 BULK TRANSFER
13 CICS LOG STREAM or VSAM_LOGSTREAM
14 USER SOURCE
15
16
17 DB2 LOGREC RAW
18 IMS LOGREC RAW
19 POOL LOGREC RAW
20 IDMS JOURNALREC RAW
21 IDMS_JOURNALREC
22
23
24
25
26
27
28
29 CICS_JOURNAL_RAW
30 CICS_JOURNAL
31
32
33 DATACOM_LOGREC
34 DATACOM_LOG_BLOCKED
35 DB2_LUW_LOGREC_BLOCKED
36 DB2_LUW_LOGREC
37 ORACLE_LOGREC_BLOCKED
38 ORACLE_LOGREC
39 MSSQL_LOGREC_BLOCKED
40 MSSQL_LOGREC

X_SX_I.DataType — Type of data record
1 File
2 Db2
3 IMS
4 VSAM
5 Adabas
6 Datacom
7 IDMS
8 SQL (DRDA, ODBC, EXASOL, PostgreSQL Oracle)
9
10 Db2 LUW for Linux, UNIX, and Windows
11 Oracle for Linux, UNIX, and Windows
12 MS SQL Server

X_SX_I.Raw_Data — Data in specific format of the data source

Output Parameters:

X_SX_O.Raw_Data — Data in specific data source format
X_SX_O.LogRec — Change data of data record

X_SX_O.ReturnFlags Return value
0 Process record
1 Skip record

Example:

```
CDC_ExitS0LogRecordRead:

X_SX_O.ReturnFlags = 0
RETURN '0, No Error'
```

3.16 CDC_ExitS1LogRecordRead

The exit procedure CDC_ExitS1LogRecordRead will be called if a stage 1 record has been read. The exit can process and also modify the data.

Parameter Stem:
Input: X_SX_I
Output: X_SX_O

Input Parameters:

X_SX_I.Stage	1	Always stage 1
X_SX_I.DataSourceType	Type of data source	
	1	DB2 EXIT
	2	DB2 LOGREC
	3	IMS DRA EXIT
	4	IMS EXIT
	5	IMS LOGREC
	6	CICS VSAM EXIT
	7	ADABAS EXIT
	8	ADABAS PLOG
	10	IDMS EXIT
	11	BATCH COMPARE
	12	BULK TRANSFER
	13	CICS LOG STREAM or VSAM_LOGSTREAM
	14	USER SOURCE
	15	
	16	
	17	DB2 LOGREC RAW
	18	IMS LOGREC RAW
	19	POOL LOGREC RAW
	20	IDMS JOURNALREC RAW
	21	IDMS_JOURNALREC
	22	
	23	
	24	
	25	
	26	
	27	
	28	
	29	CICS_JOURNAL_RAW
	30	CICS_JOURNAL
	31	
	32	
	33	DATACOM_LOGREC

34 DATACOM_LOG_BLOCKED
35 DB2_LUW_LOGREC_BLOCKED
36 DB2_LUW_LOGREC
37 ORACLE_LOGREC_BLOCKED
38 ORACLE_LOGREC
39 MSSQL_LOGREC_BLOCKED
40 MSSQL_LOGREC

X_SX_I.DataType — Type of data record
1 File
2 Db2
3 IMS
4 VSAM
5 Adabas
6 Datacom
7 IDMS
8 SQL (DRDA, ODBC, Oracle)
9
10 Db2 LUW for Linux, UNIX, and Windows
11 Oracle for Linux, UNIX, and Windows
12 MS SQL Server

X_SX_I.Timestamp — Timestamp of data record if the data source supports this
X_SX_I.UniqueId — Data source-specific unique ID of the data record if the data source supports it.
X_SX_I.Function_Type — Function type of the data record:
1 Start of a transaction
2 Commit of a transaction
3 Rollback of a transaction
4 End a transaction with Commit
5 End a transaction with Rollback
6 End a transaction
7 Undo/redo Logrec
X_SX_I.Operation_Type — Operation type of data record undo/redo:
1 INSERT
2 UPDATE
3 DELETE
4 TRUNCATE
For adjunction and combination replications the original operation to the source object is given.
X_SX_I.LUW_ID — Name of Logical Unit of Work (LUW)
X_SX_I.Table_Name — Name of Logrec object transformed to a target table name. The structure information will be read using this name. Only for undo/redo data records
X_SX_I.BI_Data — Before image data for undo/redo data records
X_SX_I.AI_Data — After image data for undo/redo data records

Data source-specific fields:

DLI:
X_SX_I.DLI_CKL — DLI length of the concatenated key
X_SX_I.DLI_DB — DLI DB name
X_SX_I.DLI_SEGMENT — DLI segment name

ADABAS:

X_SX_I.ADA_DBNR	Adabas database number
X_SX_I.ADA_FINR	Adabas file number

DATACOM:

X_SX_I.DATACOM_COMMAND	Datacom command
X_SX_I.DATACOM_DBID	Datacom database ID
X_SX_I.DATACOM_FILE	Datacom filename
X_SX_I.DATACOM_ELEMENTS	Datacom element list
X_SX_I.DATACOM_RUN_UNIT	Datacom run unit
X_SX_I.DATACOM_TSN	Datacom transaction number
X_SX_I.DATACOM_JOB_NAME	Datacom job name
X_SX_I.DATACOM_USER_ID	Datacom user ID
X_SX_I.DATACOM_MONITOR_ID	Datacom monitor ID

CICS/VSAM:

X_SX_I. VSAM_TYPE	VSAM file type: 0: KSDS 1: ESDS 2: RRDS
X_SX_I.VSAM_RECORDFORMAT	VSAM record format: 0: Fixed 1: Variable
X_SX_I.VSAM_KEYPOS	Relative key position in data record
X_SX_I.VSAM_KEYLEN	Key length in data record
X_SX_I.VSAM_RBA_RRN	RBA or RRN of data record
X_SX_I.CICS_USERID	CICS user ID
X_SX_I.CICS_TRANSACTION	CICS transaction ID
X_SX_I.CICS_FILE	8-digit file ID of the CICS system
X_SX_I.CICS_ID	8-digit VTAM ID of the CICS system

Output Parameters:

X_SX_O.BI_Data	Before image data for undo/redo data records if these have been changed
X_SX_O.AI_Data	After image data for undo/redo data records if these have been changed
X_SX_O.Table_Name	Structure information will be read using this new name. Only for undo/redo data records
X_SX_O.ReturnFlags	Return value 0 Process record 1 Skip record

Example:

```
CDC_ExitS1LogRecordRead:
SAY 'LUW='X_SX_I.LUW_ID
SAY  X_SX_I.UniqueId
SAY  X_SX_I.Table_Name
SAY  X_SX_I.Timestamp
SAY  X_SX_I.Function_Type
SAY  X_SX_I.Operation_Type

IF  X_SX_I.Table_Name = 'ARTICLE' THEN
      X_SX_O.Table_Name = 'ARTICLE1'
```

```
X_SX_O.ReturnFlags = 0
RETURN '0, No Error'
```

3.17 CDC_ExitS2LogRecordRead

The exit procedure CDC_ExitS2LogRecordRead will be called after a stage 2 record has been read. The exit can process and also modify the data.

Parameter Stem:

Input:	X_SX_I
Output:	X_SX_O

Input Parameters:

X_SX_I.Stage	2	Always stage 2
X_SX_I.DataSourceType	Type of data source	
	1	DB2 EXIT
	2	DB2 LOGREC
	3	IMS DRA EXIT
	4	IMS EXIT
	5	IMS LOGREC
	6	CICS VSAM EXIT
	7	ADABAS EXIT
	8	ADABAS PLOG
	10	IDMS EXIT
	11	BATCH COMPARE
	12	BULK TRANSFER
	13	CICS LOG STREAM or VSAM_LOGSTREAM
	14	USER SOURCE
	15	
	16	
	17	DB2 LOGREC RAW
	18	IMS LOGREC RAW
	19	POOL LOGREC RAW
	20	IDMS JOURNALREC RAW
	21	IDMS_JOURNALREC
	22	
	23	
	24	
	25	
	26	
	27	
	28	
	29	CICS_JOURNAL_RAW
	30	CICS_JOURNAL
	31	
	32	
	33	DATACOM_LOGREC
	34	DATACOM_LOG_BLOCKED
	35	DB2_LUW_LOGREC_BLOCKED
	36	DB2_LUW_LOGREC
	37	ORACLE_LOGREC_BLOCKED
	38	ORACLE_LOGREC
	39	MSSQL_LOGREC_BLOCKED

40 MSSQL_LOGREC

X_SX_I.DataType — Type of data record
1 File
2 Db2
3 IMS
4 VSAM
5 Adabas
6 Datacom
7 IDMS
8 SQL (DRDA, ODBC, ORACLE)
9
10 Db2 LUW for Linux, UNIX or Windows
11 Oracle for Linux, UNIX or Windows
12 MS SQL Server

X_SX_I.Timestamp — Timestamp of data record if the data source supports this

X_SX_I.UniqueId — Data source-specific, unique ID of the data record if the data source supports it

X_SX_I.Function_Type — Function type of data record:
1 Start of a transaction
2 Commit of a transaction
3 Rollback of a transaction
4 End a transaction with Commit
5 End a transaction with Rollback
6 End a transaction
7 Undo/redo Logrec

X_SX_I.Operation_Type — Operation type of data record undo/redo:
1 INSERT
2 UPDATE
3 DELETE
4 TRUNCATE
For adjunction and combination replications the original operation to the source object is given.

X_SX_I.LUW_ID — Name of Logical Unit of Work (LUW)

X_SX_I.Table_Name — Name of Logrec object transformed to a target table name. The structure information will be read using this name. Only for undo/redo data records

X_SX_I.BI_Data — Before image data for undo/redo data records

X_SX_I.AI_Data — After image data for undo/redo data records

X_SX_I.Field_Count — Number of data fields in the before and after image data records. Only for undo/redo data records.

The following fields have an index from 1 to X_SX_I.Field_Count:

X_SX_I.Field_Name.suffix — Field name

X_SX_I.Field_Type.suffix — SQL data type

X_SX_I.Field_Length.suffix — SQL length

X_SX_I.Field_Scale.suffix — SQL scale

X_SX_I.Field_KeyNr.suffix — 1 to n for key fields, otherwise 0

X_SX_I.Field_CCSID.suffix — CCSID of the field

X_SX_I.Field_Table.suffix — Field table number

X_SX_I.Field_Level.suffix — Field table level

X_SX_I.Field_Index.suffix.n — Field table occurrence

X_SX_I.Field_BI_Null.suffix	Field null indicator in before image
X_SX_I.Field_BI_Value.suffix	Field value in before image
X_SX_I.Field_AI_Null.suffix	Field null indicator in after image
X_SX_I.Field_AI_Value.suffix	Field value in after image

Output Parameters:

The following fields have an index from 1 to X_SX_I.Field_Count:

X_SX_O.Field_CCSID.suffix	New CCSID for the field
X_SX_O.Field_BI_Null.suffix	New field null indicator in before image
X_SX_O.Field_BI_Value.suffix	New field value in before image
X_SX_O.Field_AI_Null.suffix	New field null indicator in after image
X_SX_O.Field_AI_Value.suffix	New field value in after image

X_SX_O.Operation_Type	Changed operation type
X_SX_O.ReturnFlags	Return value 0 Process record 1 Skip record 4 The processing shall proceed with the changed operation type in X_SX_O.Operation_Type

Example:

```
CDC_ExitS2LogRecordRead:

SAY 'LUW='X_SX_I.LUW_ID 'Table='X_SX_I.Table_Name

If X_SX_I.Function_Type = 7 THEN
   DO I = 1 TO X_SX_I.Field_Count
      SAY X_SX_I.Field_Name.I X_SX_I.Field_Type.I X_SX_I.Field_Length.I,
          X_SX_I.Field_Scale.I X_SX_I.Field_KeyNr.I X_SX_I.Field_BI_Null.I,
          X_SX_I.Field_BI_Value.I X_SX_I.Field_AI_Null.I,
          X_SX_I.Field_AI_Value.I
      END

X_SX_O.ReturnFlags = 0
RETURN '0, No Error'
```

3.18 CDC_ExitS3LogRecordRead

The exit procedure CDC_ExitS3LogRecordRead will be called after a stage 3 record has been read. The exit can process and modify the data.

Parameter Stem:

Input:	X_SX_I
Output:	X_SX_O

Input Parameters:

X_SX_I.Stage	3 Always stage 3
X_SX_I.CallFlag	Type of call 0 First call 1 Next call. The exit can now create a data record based on the data from the first call and pass this record for processing.

X_SX_I.DataSourceType — Type of data source

1	Db2 EXIT
2	Db2 LOGREC
3	IMS DRA EXIT
4	IMS EXIT
5	IMS LOGREC
6	CICS VSAM EXIT
7	ADABAS EXIT
8	ADABAS PLOG
10	IDMS EXIT
11	BATCH COMPARE
12	BULK TRANSFER
13	CICS LOG STREAM or VSAM_LOGSTREAM
14	USER SOURCE
15	
16	
17	DB2 LOGREC RAW
18	IMS LOGREC RAW
19	POOL LOGREC RAW
20	IDMS JOURNALREC RAW
21	IDMS_JOURNALREC
22	
23	
24	
25	
26	
27	
28	
29	CICS_JOURNAL_RAW
30	CICS_JOURNAL
31	
32	
33	DATACOM_LOGREC
34	DATACOM_LOG_BLOCKED
35	DB2_LUW_LOGREC_BLOCKED
36	DB2_LUW_LOGREC
37	ORACLE_LOGREC_BLOCKED
38	ORACLE_LOGREC
39	MSSQL_LOGREC_BLOCKED
40	MSSQL_LOGREC

X_SX_I.DataType — Type of data record

1	File
2	Db2
3	IMS
4	VSAM
5	Adabas
6	Datacom
7	IDMS
8	SQL (DRDA, ODBC, Oracle)
9	
10	Db2 LUW for Linux, UNIX, or Windows
11	Oracle for Linux, UNIX, or Windows
12	MS SQL Server

X_SX_I.Timestamp
Timestamp of data record if the data source supports it.

X_SX_I.UniqueId
Data-specific unique ID of the data record if the data source supports it.

X_SX_I.Function_Type
Function type of data record:
1 Start of a transaction
2 Commit of a transaction
3 Rollback of a transaction
4 End a transaction with Commit
5 End a transaction with Rollback
6 End a transaction
7 Undo/redo Logrec

X_SX_I.Operation_Type
Operation type of data record undo/redo:
1 INSERT
2 UPDATE
3 DELETE
4 TRUNCATE

X_SX_I.LUW_ID
Name of Logical Unit of Work (LUW)

X_SX_I.Table_Name
Name of Logrec object transformed to a target table name. The structure information will be read using this name. Only for undo/redo data records.

X_SX_I.SQL_Comment
An optional comment with additional data source-specific information

X_SX_I.SQL_DML
The SQL DML statement

X_SX_I._Parm_Count
Number of parameter values for the SQL DML statement. Only for undo/redo data records.

The following fields have an index from 1 to X_SX_I.Parm_Count:

X_SX_I.Parm_Name.suffix
Name of parameter

X_SX_I.Parm_Type.suffix
SQL type of parameter

X_SX_I.Parm_Length.suffix
SQL length of parameter

X_SX_I.Parm_Scale.suffix
SQL scale of parameter

X_SX_I.Parm_KeyNr.suffix
if > 0: Sequence number of parameter in the unique key

X_SX_I.Parm_Null.suffix
1 = value is NULL

X_SX_I.Parm_Value.suffix
Parameter value

Output Parameter:

X_SX_O.SQL_Comment
New optional comment

X_SX_O.SQL_DML
New SQL DML statement

X_SX_O.Parm_Count
New number of parameter values for SQL DML statement. Only for undo/redo data records.

The following fields have an index from 1 to X_SX_O.Parm_Count:

X_SX_O.Parm_Name.suffix
New name of parameter

X_SX_O.Parm_Type.suffix
New SQL type of parameter

X_SX_O.Parm_Length.suffix
New SQL length of parameter

X_SX_O.Parm_Scale.suffix
New SQL scale of parameter

X_SX_O.Parm_KeyNr.suffix
New sequence number of parameter in unique key

X_SX_O.Parm_Null.suffix
New null indicator, 1 = value is NULL

X_SX_O.Parm_Value.suffix
New parameter value

X_SX_O.ReturnFlags
Return value
0 Process record
1 Skip record

2 Process record and call the exit a second time with X_SX_I.CallFlag = 1. This will enable the exit to create an additional virtual data record based on the data from the first call.

Example:

```
CDC_ExitS3LogRecordRead:
SAY 'CallFlag:'X_SX_I.CallFlag
SAY 'Comment: 'X_SX_I.SQL_Comment
SAY 'DML:     'X_SX_I.SQL_DML

IF MyCount = 'MYCOUNT' THEN MyCount = 1

IF X_SX_I.CallFlag = 0 THEN DO
      X_SX_O.SQL_DML = 'INSERT INTO TEST.CDC (TX1,TXLV1) VALUES(?,?)'
      X_SX_O.Parm_Count    = 2
      X_SX_O.Parm_Name.1   = 'Number'
      X_SX_O.Parm_Type.1   = 457
      X_SX_O.Parm_Length.1 = 8
      X_SX_O.Parm_Scale.1  = 0
      X_SX_O.Parm_Null.1   = 0
      X_SX_O.Parm_Value.1  = MyCount
      X_SX_O.Parm_Name.2   = 'SQL'
      X_SX_O.Parm_Type.2   = 457
      X_SX_O.Parm_Length.2 = 300
      X_SX_O.Parm_Scale.2  = 0
      X_SX_O.Parm_Null.2   = 0
      X_SX_O.Parm_Value.2  = X_SX_I.SQL_Comment
      X_SX_O.ReturnFlags   = 2
END
IF X_SX_I.CallFlag = 1 THEN DO
      X_SX_O.SQL_DML = 'INSERT INTO BEAT.CDC (TX1,TXLV1) VALUES(?,?)'
      X_SX_O.Parm_Count    = 2
      X_SX_O.Parm_Count    = 2
      X_SX_O.Parm_Name.1   = 'Number'
      X_SX_O.Parm_Type.1   = 457
      X_SX_O.Parm_Length.1 = 8
      X_SX_O.Parm_Scale.1  = 0
      X_SX_O.Parm_Null.1   = 0
      X_SX_O.Parm_Value.1  = MyCount
      X_SX_O.Parm_Name.2   = 'SQL'
      X_SX_O.Parm_Type.2   = 457
      X_SX_O.Parm_Length.2 = 300
      X_SX_O.Parm_Scale.2  = 0
      X_SX_O.Parm_Null.2   = 0
      X_SX_O.Parm_Value.2  = X_SX_I.SQL_DML
      X_SX_O.ReturnFlags   = 0
END
MyCount = MyCount + 1
RETURN '0,No Error'
```

3.19 CDC_ExitInputField

The exit procedure CDC_ExitInputField will be called if field FLAG of data field in the structure definition has a value of 1 and/or 2.

Parameter Stem:

Input:	X_FL_I
Output:	X_FL_O

Input Parameters:

X_FL_I.CallType	Type of call 3 Call to convert from data source format to input SQL format 4 Call to convert from input SQL format to output SQL format
X_FL_I.Name	SQL name of the field
X_FL_I.InType	SQL type number of input field
X_FL_I.InLength	Binary length of input field including null indicator, data, etc.
X_FL_I.InPrecision	Number of digits
X_FL_I.InScale	Number of decimal digits
X_FL_I.InCCSID	Input CCSID
X_FL_I.InFieldData	Input data
X_FL_I. Field_Is_Null	Null indicator for the input field 1 Field is NULL 0 Field is not NULL

Output Parameters:

X_FL_O.ReturnFlags	Return value: 0 Do not apply changed field data 1 Apply changed field data
X_FL_O.OutType	Output SQL data type
X_FL_O.OutCCSID	Output CCSID
X_FL_O.OutScale	Output number of digits
X_FL_O.OutPrecision	Output number of decimal places
X_FL_O.OutFieldData	Output value
X_FL_O. Field_is_Null	Null indicator for the output field 1 Field is NULL 0 Field is not NULL

Example:

```
CDC_ExitInputField:

X_FL_O.ReturnFlags = 0
SELECT
  WHEN X_FL_I.ExitNumber = 2452 THEN DO
     X_FL_O.OutFieldData = LEFT(X_FL_I.InFieldData, LENGTH(X_FL_I.InFieldData) / 2)
     X_FL_O.OutFieldData = OVERLAY('d9'x , X_FL_O.OutFieldData)
     X_FL_O.ReturnFlags = 1
     RETURN 0
  END

    OTHERWISE RETURN '12,Invalid Input Type' X_FL_I.InType ' for Field' X_FL_I.Name
END

RETUEN '0,OK'
```

3.20 CDC_ExitLUW_Manager

The exit procedure CDC_ExitLUW_Manager will be called if:

- A new LUW starts
- An LUW has been completed and can be processed
  An LUW can be deleted after processing by the LUW Manager
- LUWs that are not completed at script termination are registered with the Agent

Parameter Stem:

Input:	X_LUW_I
Output:	X_LUW_O

Input Parameters:

X_LUW_I.LUW_Id	ID (RecoveryToken) of the Logical Unit of Work in text format
X_LUW_I.Processing	LUW type of processing: 1 Start of a new LUW 2 Execute an LUW that has been completed with COMMIT on the source system 3 Execute an LUW that has been completed with ROLLBACK on the source system 4 Execute an LUW that received neither a COMMIT nor a ROLLBACK return message 5 Delete an LUW after processing 6 A data record referencing an LUW that is not registered should be processed. Only the parameters up to and with X_LUW_I.StartTime are set. 7 Set Parallel Apply SlotID
X_LUW_I.StartTime	Start timestamp of LUW from source system
X_LUW_I.DataType	Type of data record 1 File 2 Db2 3 IMS 4 VSAM 5 Adabas 6 Datacom 7 IDMS 8 SQL (DRDA, ODBC, Oracle) 9 10 Db2 LUW for Linux, UNIX, or Windows 11 Oracle for Linux, UNIX, or Windows 12 MS SQL Server
X_LUW_I.DataSourceType	Type of data source 1 DB2_EXIT 2 DB2_LOGREC 3 IMS_DRA_EXIT 4 IMS_EXIT 5 IMS_LOGREC 6 CICS_VSAM_EXIT 7 ADABAS_EXIT 8 ADABAS_PLOG 10 IDMS_EXIT 11 BATCH_COMPARE

12	BULK_TRANSFER
13	CICS_LOGSTREAM or VSAM_LOGSTREAM
14	USER_SOURCE
15	
16	
17	DB2_LOGREC_RAW
18	IMS_LOGREC_RAW
19	POOL_LOGREC_RAW
20	IDMS_JOURNALREC_RAW
21	IDMS_JOURNALREC
22	
23	
24	
25	
26	
27	
28	
29	CICS_JOURNAL_RAW
30	CICS_JOURNAL
31	
32	
33	DATACOM_LOGREC
34	DATACOM_LOG_BLOCKED
35	DB2_LUW_LOGREC_BLOCKED
36	DB2_LUW_LOGREC
37	ORACLE_LOGREC_BLOCKED
38	ORACLE_LOGREC
39	MSSQL_LOGREC_BLOCKED
40	MSSQL_LOGREC

The following parameters are only set for X_LUW_I.Processing > 1:

X_LUW_I.NrLogRecs	Number of LogRecs in this LUW
X_LUW_I.NrLogBytes	Number of bytes of all LogRecs in this LUW
X_LUW_I.NrUndoRedoLogRecs	Number of undo/redo LogRecs in this LUW
X_LUW_I.EndTime	End timestamp of LUW from source system

Data source-specific fields:

DB2_EXIT, DB2_LOGREC:

X_LUW_I.DB2_AuthorizationId
X_LUW_I.DB2_ResourceName
X_LUW_I.DB2_ConnectionType
X_LUW_I.DB2_ConnectionId
X_LUW_I.DB2_CorrelationId

IMSEXIT,
IMS_DRA_EXIIT,

CICS_VSAM_EXIT:

X_LUW_I.CICS_ID
X_LUW_I.CICS_UserID
X_LUW_I.CICS_Transaction
X_LUW_I.CICS_Task

IDMS_EXIT ,IDMS_JOURNALREC:

X_LUW_I.IDMS_ID
X_LUW_I.IDMS_Area

X_LUW_I.IDMS_Record
X_LUW_I.IDMS_Program
X_LUW_I.IDMS_Task1
X_LUW_I.IDMS_Task2
X_LUW_I.IDMS_RUID
X_LUW_I.IDMS_CVNO

ADABAS EXIT, ADABAS PLOG

X_LUW_I.ADABAS_Jobname
X_LUW_I.ADABAS_UserID

DATACOM_LOGREC:

X_LUW_I.DATACOM_Run_Unit
X_LUW_I.DATACOM_Tsn
X_LUW_I.DATACOM_Job_Name
X_LUW_I.DATACOM_User_Id
X_LUW_I.DATACOM_Monitor_Id

ORACLE_LOGREC:

X_LUW_I.ORA_UserId
X_LUW_I.ORA_AuditSessionId
X_LUW_I.ORA_SCN

MSSQL_LOGREC:

X_LUW_I.MSSQL_UId
X_LUW_I.MSSQL_Server_Uid

POSTGRESQL_LOGREC:

X_LUW_I.POSTGRES_XID
X_LUW_I.POSTGRES_ORIGIN_ID

Note:

- If X_LUW_I.NrUndoRedoLogRecs = 0, no data has been changed and the LUW Manager flags this with the autoskip flag. This means that the LUW will not be passed to the target system.
- If X_LUW_I.Processing = 3 (the LUW was terminated with ROLLBACK in the source system), the LUW Manager flags this with the autoskip flag. This means that the LUW will not be passed to the target system.

Output Parameters:

X_LUW_O.ReturnFlags — Return value

0 Process LUW
1 Skip LUW
2 Reset autoskip flag and process LUW

For X_LUW_I.Processing = 6:

0 Issue error message and terminate processing
3 Pass this LUW
4 Implicitly start a new LUW with the LUW reference of the data record

For X_LUW_I.Processing = 7:

5 The Return Message contains the Parallel Apply SlotID to use.
Example SlotID=9 for the REXX-Exit:
RETURN '0,9'

Example SlotID=9 for the HP-Exit:
sprintf(ErrorMsg, "%d", 9);

X_LUW_O. BackUpdate Return value
For X_LUW_I.Processing = 1:
1 INSERT commands will be translated into UPDATE commands. Use this option when the corresponding data record already exists in the target system and the INSERT command would be rejected with a DUPLICATE condition. With this option an UPDATE will be issued and the data record that already exists in the target system will be modified. A possible application could be the synchronization of the ISN field of an Adabas table. Refer to Adabas: General Considerations on page 220.

X_LUW_O. IDMSBackUpdate
Return value
For X_LUW_I.Processing = 1:
1 INSERT commands will be translated into UPDATE commands. Use this option when the corresponding data record already exists in the target system and the INSERT command would be rejected with a DUPLICATE condition. With this option an UPDATE will be issued and the data record that already exists in the target system will be modified. Also refer to IDMS: Updates Received from a Target DBMS on page 224.

Example:

```
CDC_ExitLUW_Manager:

IF X_LUW_I.Processing = 6  THEN DO
      SAY 'No LUW!!'
      X_LUW_O.ReturnFlags = 4
      RETURN '0,No Error'
END

IF X_LUW_I.NrUndoRedoLogRecs > 0 THEN
      X_LUW_O.ReturnFlags = 2
ELSE
      X_LUW_O.ReturnFlags = 0

SAY X_LUW_I.LUW_Id X_LUW_I.Processing X_LUW_I.StartTime X_LUW_I.NrLogRecs
X_LUW_I.NrLogBytes
SAY X_LUW_I.NrUndoRedoLogRecs X_LUW_I.EndTime

IF X_LUW_I.Processing = 4  THEN X_LUW_O.ReturnFlags = 0
IF X_LUW_O.ReturnFlags = 1 THEN SAY 'Skipping 'X_LUW_I.LUW_Id

RETURN '0,No Error'
```

3.21 CDC_ExitDBControl

The exit procedure CDC_ExitDBControl will be called if special processing in the source database system takes place or has taken place. For example:

- A database utility loads bulk data into a table without logging functions.
- The data source name is no longer the same as the one in the repository.

Parameter Stem:

Input:	X_DBC_I
Output:	X_DBC_O

Input Parameters:

X_DBC_I.CallType — Type of call:

1 ADABAS ADALOD UPDATE function
2 ADABAS ADADBS REFRESH function
3
4
5 Db2 z/OS CREATE TABLE function
6 Db2 z/OS DROP TABLE function
8 Bulk output file name
9 Db2 LUW CREATE TABLE function
10 Db2 LUW DROP TABLE function
11 ORACLE CREATE TABLE function
12 ORACLE DROP TABLE function
13 MSSQL CREATE TABLE function
14 MSSQL DROP TABLE function
20 Db2 z/OS new row in SYSIBM.SYSCOPY
21 Db2 z/OS dictionary update
22 Db2 z/OS ALTER TABLE...
23 Db2 z/OS CREATE/DROP INDEX...
24 VSAM LogStream LS_NOTE()
25 SMF VSAM switch
26 ADABAS LUW PLOG switch
27 ADABAS LOB UPDATE unlinked
28 ADABAS LOB UPDATE partial data
30 Change access key for repository
32 Change output table name
40 ADABAS ADASAV RESTORE FILE functions
41 ADABAS ADASAV RESTORE PLOG functions
42 ADABAS ADASAV RESTORE DELTA functions
43 ADABAS ADAORD STORE FILE functions
44 ADABAS ADARES REGENERATE FILE functions
45 ADABAS ADARES REGENERATE ALL functions
46 ADABAS ADARES BACKOUT FILE functions
47 ADABAS ADARES BACKOUT ALL functions
48 ADABAS ADARES REPAIR functions
49 ADABAS AOSDBS CHANGE FIELD LENGTH functions
50 ADABAS AOSDBS DELETE FILE functions
51 ADABAS AOSDBS RENUMBER FILE functions
52 ADABAS AOSDBS DEFINE NEW FIELD functions
53 ADABAS AOSDBS DEL/UNDEL FIELD functions
54 ADABAS ADAULD functions
55 ADABAS ADASAV functions
60 Oracle CREATE RESTORE POINT functions

61 Oracle TRUNCATE TABLE event

The following parameters are only valid for X_DBC_I.CallType = 1,2, and 40-53:

X_DBC_I.ADABAS_DBNR	Adabas DBNR of the corresponding file
X_DBC_I.ADABAS_FINR	Adabas file number of the corresponding file
X_DBC_I.ADABAS_JOB	Job name of Adabas ADALOD Job
X_DBC_I.ADABAS_TS	Timestamp
X_DBC_I.ADABAS_PLOG	PLOG nr
X_DBC_I.ADABAS_BLOCK	PLOG block
X_DBC_I.ADABAS_OFFSET	PLOG block offset
X_DBC_I.ADABAS_C5	Adabas PLOG C5 user data. The data will be written into PLOG by the job of the ADALOD utility.
X_DBC_I.ADABAS_INFO	Additional information as text for X_DBC_I.CallType = 49,51,52, and 53.

The following parameters are only set for X_DBC_I.CallType = 5,6:

X_DBC_I.DB2_TS	Timestamp of CREATE/DROP
X_DBC_I.DB2_CREATOR	Creator name of CREATE/DROP
X_DBC_I.DB2_TABLE	Table name of CREATE/DROP
X_DBC_I.DB2_DBID	DBID of CREATE/DROP
X_DBC_I.DB2_OBID	OBID of CREATE/DROP

The following parameters is only set for X_DBC_I.CallType = 8:

X_DBC_I.DataTargetName	Defined target name

The following parameters are only set for X_DBC_I.CallType = 9,10:

X_DBC_I.D2L_TS	Timestamp of CREATE/DROP
X_DBC_I.D2L_CREATOR	Creator name of CREATE/DROP
X_DBC_I.D2L_TABLE	Table name of CREATE/DROP
X_DBC_I.D2L_TID	TID of CREATE/DROP
X_DBC_I.D2L_FID	FID of CREATE/DROP

The following parameters are only set for X_DBC_I.CallType = 11,12,61:

X_DBC_I.ORA_TS	Timestamp of CREATE/DROP/TRUNCATE
X_DBC_I.ORA _CREATOR	Schema name
X_DBC_I.ORA_TABLE	Table name
X_DBC_I.ORA_OBJ	Object ID

The following parameters are only set for X_DBC_I.CallType = 20:

X_DBC_I.DB2_TS	Timestamp
X_DBC_I.DB2_DBNAME	Database name
X_DBC_I.DB2_TSNAME	Tablespace name
X_DBC_I.DB2_ICTYPE	ICTYPE

The following parameters are only set for X_DBC_I.CallType = 21:

X_DBC_I.DB2_DIC_SIZE	Dictionary size
X_DBC_I.DB2_DIC_DBID	DBID
X_DBC_I.DB2_DIC_PSID	PSID
X_DBC_I.DB2_DIC_PARTITION	Partition
X_DBC_I.DB2_DIC_PAGESIZE	Page size

The following parameters are only set for X_DBC_I.CallType = 22:

X_DBC_I.DB2_TS	Timestamp
X_DBC_I.DB2_AT_STAGE	Processing stage 0 or 2 if LUW COMMITed

For X_DBC_I.DB2_AT_TYPE=
'MODIFY RECOVERY' | 'LOAD DATA' | 'LOAD REPLACE DATA' | 'RECOVER':

X_DBC_I.DB2_AT_DBID	DBID
X_DBC_I.DB2_AT_OBID	OBID[3]
X_DBC_I.DB2_AT_PSID	PSID

The following parameter is only set for X_DBC_I.CallType = 22, if a PSID is available:

X_DBC_I.DB2_AT_FULLTABLENAME	Tablename of the loaded table in the form SUBSYSTEM.CREATOR.NAME

For X_DBC_I.DB2_AT_TYPE='ADD COLUMN'

X_DBC_I.DB2_AT_TYPE	'ADD COLUMN'
X_DBC_I.DB2_AT_VERSION	Version
X_DBC_I.DB2_AT_DBID	DBID
X_DBC_I.DB2_AT_OBID	OBID
	and further if contained in LogRec:
X_DBC_I.DB2_AT_COLNR	Column number
X_DBC_I.DB2_AT_NULL	Column nullable
X_DBC_I.DB2_AT_DEFAULT	Column default
X_DBC_I.DB2_AT_SBCS	SBCS
X_DBC_I.DB2_AT_DBCS	DBCS
X_DBC_I.DB2_AT_MBCS	MBCS
X_DBC_I.DB2_AT_COLTYPE	Column type
X_DBC_I.DB2_AT_SRCTYPE	Column source type
X_DBC_I.DB2_AT_LEN	Column length
X_DBC_I.DB2_AT_PREC	Column precision
X_DBC_I.DB2_AT_SCALE	Column scale
X_DBC_I.DB2_AT_COLNAME	Column name
X_DBC_I.DB2_AT_TABLENAME	Table name

The following parameter is only set for X_DBC_I.CallType = 23:
X_DBC_I.DB2_TS TimeStamp
X_DBC_I.DB2_AI_STAGE Processing-Stage 1 or 2, if LUW COMMITed
X_DBC_I.DB2_AI_TYPE „CREATE“, „ALTER“ or „DROP“ Index
X_DBC_I.DB2_AI_RBA_LRSN
X_DBC_I.DB2_AI_INDEXNAME Creator and Index-Name
X_DBC_I.DB2_AI_TABLENAME Table-Creator and Table-Name
X_DBC_I.DB2_AI_DBID
X_DBC_I.DB2_AI_OBID
X_DBC_I.DB2_AI_ISOBID

X_DBC_I.DB2_AI_COLCOUNT	Number of Fields in Index
X_DBC_I.DB2_AI_UNIQUERULE	UNIQUERULE aus SYSIBM.SYSINDEXES
X_DBC_I.DB2_AI_CLUSTERING	CLUSTERING aus SYSIBM.SYSINDEXES
X_DBC_I.DB2_AI_CLUSTERED	CLUSTERED aus SYSIBM.SYSINDEXES

For the TYPE "ALTER" the following parameters are additionally set with the Before-Values:

X_DBC_I.DB2_AI_BI_COLCOUNT	Number of Fields in Index
X_DBC_I.DB2_AI_BI_UNIQUERULE	UNIQUERULE from SYSIBM.SYSINDEXES
X_DBC_I.DB2_AI_BI_CLUSTERING	CLUSTERING from SYSIBM.SYSINDEXES
X_DBC_I.DB2_AI_BI_CLUSTERED	CLUSTERED from SYSIBM.SYSINDEXES

[3] if PS-ID is available only

The dB2 table SYSIBM.SYSINDEXES must have the CDC flag active to activate the X_DBC_I.CallType = 23 (DATA CAPTURE CHANGES)!
INDEX operations are only processed if there is a valid repository definition for the valid repository definition exists for the corresponding table.

The following parameter is only set for X_DBC_I.CallType = 24:

X_DBC_I.LS_NOTE	Content of the 2. field of the LS_NOTE() function With the REXX-Funktion LS_NOTE(), customer-specific entries can be made in the respective journal log stream e.g. after a VSAM REPRO in order to start a certain processing in the target DB at the correct time.

The following parameters are only set for X_DBC_I.CallType = 25:

X_DBC_I.SMF_OLD_ACTIVE	Previous active VSAM file
X_DBC_I.SMF_NEW_ACTIVE	New active VSAM file

The following parameters are only set for X_DBC_I.CallType = 26:

X_DBC_I.PLOG_SESSION	PLOG session number
X_DBC_I.PLOG_EXTENT	PLOG extent number

The following parameters are only set for X_DBC_I.CallType = 30:

X_DBC_I.REPO_TABLETYPE	Table type for input table in the repository
X_DBC_I.REPO_TABLEID	Table name for input table in the repository
X_DBC_I.REPO_TABLETIMESTAMP	Timestamp for input table in the repository

After the processing of type 30 the following parameters can be used to provide changed values for the repository access.

X_DBC_O.REPO_TABLETYPE	Table type for input table in the repository
X_DBC_O.REPO_TABLEID	Table name for input table in the repository
X_DBC_O.REPO_TABLETIMESTAMP	Timestamp for input table in the repository

The following parameters are only set for X_DBC_I.CallType = 32:

X_DBC_I.REPO_TABLENAME	Table name of the output table

After the processing of type 32 the following parameters can be used to provide changed values for the output table names.

X_DBC_O.REPO_TABLENAME	New table name of the output table

The following parameters are only set for X_DBC_I.CallType = 60:

X_DBC_I.ORA_RESTORE_POINT	Commandtext of the CREATE RP...
X_DBC_I.ORA_RP_SCN	SCN of the CREATE RESTORE POINT
X_DBC_I.ORA_RP_RL_SCN	Resetlogs-SCN of the CREATE RP
X_DBC_I.ORA_DB_RL_SCN	Resetlogs-SCN current

Notes to Adabas:

- In the Job Stream of the Adabas Utilities, a C5 user entry into the PLOG can be made with an auxiliary script. This entry defines the important parameters of the following Adabas Utility through keywords, e.g. the name of the input file for ADALOD. The data of this input file can be written into the respective target database as SQL Loader data at the desired time with another data script.
  At this point in the job, this input file can already be used with a data script for backing up or forwarding to another target system.
  The C5 entry has to contain the following keywords:
  JOBNAME= Name of the Job
  DBID= Adabas DBID
  FILE= Adabas File Number
  With this information, the subsequent corresponding checkpoint entry of the ADALOD Utility can be found in the PLOG.
  C5 entries can also be written for the other Adabas utilities – FILE=0 can also be captured for these. With this, only DBID and JOBNAME are compared, e.g for ADARES BACKOUT ALL which can impact various files.
- The ADALOD utility writes a checkpoint into PLOG that defines the logical point of time for the bulk insert. At the right point in time this exit can be called with all information needed using this C5 entry and the ADALOD entry.

Notes to ADADBS REFRESH:

- This function can be automatically built from the corresponding PLOG entries of the utility. The exit will only be called in this case to suppress the DELETE in the target system. Using X_DBC_O.ReturnFlags = 2 the ADADBS REFRESH will not be processed further.

Output Parameters:

X_DBC_O.ReturnFlags Return value

0	No special processing
1	Start a script using start parameters
2	ADADBS REFRESH: No DELETE will be performed in the target system
3	Change access key for repository: this call shall not be cached
4	Change access key for repository: The exit is deactivated for further processing
5	Clear the internal cache of metadata information after call for „DB2 z/OS ALTER TABLE“
6	ADABAS LOB: Resume processing and skip UPDATE
7	Just like 6 without calling the exit
8	Oracle CREATE RESTORE POINT: Terminate processing.
9	Oracle CREATE RESTORE POINT: Resume processing at the current Resetlogs-SCN.

X_DBC_O.ScriptName Name of the script to be started. The corresponding script will be started and the calling script waits for the successful completion of this script before it continues processing.

X_DBC_O.ScriptParms Start parameters for the script to be started

X_DBC_O.DataSourceName	New target name
X_DBC_O.ADASAV_PATH	The optional specification of a target directory for calltype=55

Example for the creation of the C5 entry in the job stream using the tcSCRIPT Adabas interface:

```
...
//*
//*   tcVISION PLOG Hook
//*   We write an Adabas User CP-Record
//*
//STEP1    EXEC PGM=TCSCRIPT,PARM='-E SYSSCRT',REGION=0M
//STEPLIB  DD DISP=SHR,DSN=TCVISION.LOADLIB
//STDENV   DD DISP=SHR,DSN=TCVISION.MACLIB(STDENVJ)
//SYSSCRT  DD *
ADA.Command = 'C5'
ADA.RB =        'tcVISION ADALOD INFO'
ADA.RB = ADA.RB 'DATE='DATE(S)
ADA.RB = ADA.RB 'TIME='TIME(N)
ADA.RB = ADA.RB 'JOBNAME='JOBNAME()
ADA.RB = ADA.RB 'JOBID='JOBID()
ADA.RB = ADA.RB 'FUNCTION=UPDATE'
ADA.RB = ADA.RB 'DBID=1'
ADA.RB = ADA.RB 'FILE=1'
ADA.RB = ADA.RB 'DDEBAND=ADABAS.COMP01'

RC = CALLADA('','ADA', 'DQ1')
SAY 'ADABAS RC('RC') ADABAS RC('DQ1.ReturnCode')'
IF RC = 0 THEN RETURN 0
RETURN 12
...
```

Example for the exit:

```
CDC_ExitDBControl:

IF X_DBC_I.CallType = 1 THEN DO
      say X_DBC_I.CallType
      say X_DBC_I.ADABAS_DBNR
      say X_DBC_I.ADABAS_FINR
      say X_DBC_I.ADABAS_JOB
      say X_DBC_I.ADABAS_C5

      X_DBC_O.ReturnFlags = 1
      X_DBC_O.ScriptName  = 'MyScript'
      X_DBC_O.ScriptParms = 'Input_Source_Name=adalod.input.file'
END

RETURN '0,No Error'

CDC_ExitDBControl:
IF X_DBC_I.CallType = 3 THEN SAY 'Create at' X_DBC_I.D2V_TS
IF X_DBC_I.CallType = 4 THEN SAY 'Drop   at' X_DBC_I.D2V_TS
SAY TRIM(X_DBC_I.D2V_CREATOR)'.'TRIM(X_DBC_I.D2V_TABLE)
say X_DBC_I.D2V_DBSPACE', 'X_DBC_I.D2V_TABID

X_DBC_O.ReturnFlags = 0
```

```
RETURN '0,No Error'
```

Example for the exit

```
CDC_ExitDBControl:
IF  X_DBC_I.CallType = 20 THEN DO
      SAY 'DB2 z/OS SYSCOPY entry:'
      SAY 'Database:  ' X_DBC_I.DB2_DBNAME
      SAY 'Tablespace:' X_DBC_I.DB2_TSNAME
      SAY 'Timestamp: ' X_DBC_I.DB2_TS
      SELECT
         WHEN X_DBC_I.DB2_ICTYPE = 'X' THEN Operation = 'REORG LOG(YES)'
         WHEN X_DBC_I.DB2_ICTYPE = 'R' THEN Operation = 'LOAD REPLACE LOG(YES)'
         WHEN X_DBC_I.DB2_ICTYPE = 'S' THEN Operation = 'LOAD REPLACE LOG(NO)'
         WHEN X_DBC_I.DB2_ICTYPE = 'W' THEN Operation = 'REORG LOG(NO)'
         WHEN X_DBC_I.DB2_ICTYPE = 'Y' THEN Operation = 'LOAD LOG(NO)'
         WHEN X_DBC_I.DB2_ICTYPE = 'Z' THEN Operation = 'LOAD LOG(YES)'
         WHEN X_DBC_I.DB2_ICTYPE = 'A' THEN Operation = 'ALTER'
         WHEN X_DBC_I.DB2_ICTYPE = 'B' THEN Operation = 'REBUILD INDEX'
         WHEN X_DBC_I.DB2_ICTYPE = 'D' THEN Operation = 'CHECK DATA LOG(NO)'
         WHEN X_DBC_I.DB2_ICTYPE = 'F' THEN Operation = 'COPY FULL YES'
         WHEN X_DBC_I.DB2_ICTYPE = 'I' THEN Operation = 'COPY FULL NO'
         WHEN X_DBC_I.DB2_ICTYPE = 'P' THEN Operation = 'RECOVER TOCOPY'
         WHEN X_DBC_I.DB2_ICTYPE = 'Q' THEN Operation = 'QUIESCE'
         WHEN X_DBC_I.DB2_ICTYPE = 'V' THEN Operation = 'REPAIR VERSIONS'
         WHEN X_DBC_I.DB2_ICTYPE = 'T' THEN Operation = 'TERM UTILITY'
         WHEN X_DBC_I.DB2_ICTYPE = 'C' THEN Operation = 'CREATE'
         WHEN X_DBC_I.DB2_ICTYPE = 'M' THEN Operation = 'MODIFY RECOVERY'
         OTHERWISE Operation = X_DBC_I.DB2_ICTYPE
      END
      SAY 'Operation: ' X_DBC_I.DB2_ICTYPE '=' Operation
END

IF  X_DBC_I.CallType = 21 THEN DO
      SAY 'DB2 z/OS Dictionary update:'
      SAY 'DBID:      ' X_DBC_I.DB2_DIC_DBID
      SAY 'PSID:      ' X_DBC_I.DB2_DIC_PSID
      SAY 'Dict Size: ' X_DBC_I.DB2_DIC_SIZE
      SAY 'Partition: ' X_DBC_I.DB2_DIC_PARTITION
      SAY 'PageSize:  ' X_DBC_I.DB2_DIC_PAGESIZE
END

IF  X_DBC_I.CallType = 22 THEN DO
      SAY 'DB2 z/OS ALTER TABLE:'
      SAY 'Type:       ' X_DBC_I.DB2_AT_TYPE
      SAY 'Version:    ' X_DBC_I.DB2_AT_VERSION
      SAY 'DBID:       ' X_DBC_I.DB2_AT_DBID
      SAY 'OBID:       ' X_DBC_I.DB2_AT_OBID
      SAY 'COLNR:      ' X_DBC_I.DB2_AT_COLNR
      SAY 'NULL:       ' X_DBC_I.DB2_AT_NULL
      SAY 'DEFAULT:    ' X_DBC_I.DB2_AT_DEFAULT
      SAY 'DEFAULTVAL.:' X_DBC_I.DB2_AT_DEFAULTVALUE
      SAY 'SBCS:       ' X_DBC_I.DB2_AT_SBCS
      SAY 'DBCS:       ' X_DBC_I.DB2_AT_DBCS
      SAY 'MBCS:       ' X_DBC_I.DB2_AT_MBCS
      SAY 'COLTYPE:    ' X_DBC_I.DB2_AT_COLTYPE
      SAY 'SRCTYPE:    ' X_DBC_I.DB2_AT_SRCTYPE
      SAY 'LEN:        ' X_DBC_I.DB2_AT_LEN
      SAY 'PREC:       ' X_DBC_I.DB2_AT_PREC
```

```
      SAY 'SCALE:       ' X_DBC_I.DB2_AT_SCALE
      SAY 'COLNAME:     ' X_DBC_I.DB2_AT_COLNAME
END

X_DBC_O.ReturnFlags = 0

RETURN '0,No Error'
```

3.22 CDC_ExitProcessControl

Exit procedure CDC_ExitProcessControl will be called at specific points in processing to allow users to control processing.

Parameter Stem:

Input:	X_PRC_I
Output:	X_PRC_O

Input Parameters:

X_PRC_I.CallType — Type of call:

1 All available staging files or all data of a tcVISION Pipe has been processed. The next step in processing is waiting for more data.

The following parameters are only valid for X_PRC_I.CallType = 1:

X_PRC_I.LastProcessTimestamp
Source timestamp (UTC) of the last processed data record.

X_PRC_I.TimestampLastProcess
Timestamp (UTC) when that data record was processed.

Output Parameters:

X_PRC_O.ReturnFlags — Return value

0 No special action

1 Terminate the script processing with return code 4 and message TCS0279I

X_PRC_O.RecallInterval — A time interval in seconds. If a value greater than 0 is specified, the exit is called again after this interval has expired, even if there is no new data in the pipe to be read.
Possible values are 0 – 86400.
The value 0 deactivates the interval. This optional parameter is only used as input for pipes.

Example for the exit:

```
CDC_ExitProcessControl:

say "CallType:            " X_PRC_I.CallType
say "LastProcessTimestamp:" X_PRC_I.LastProcessTimestamp
say "TimestampLastProcess:" X_PRC_I.TimestampLastProcess

X_PRC_O.ReturnFlags = 0
if right(X_PRC_I.LastProcessTimestamp, 15) > "19.00.00.000000" then
   X_PRC_O.ReturnFlags = 1

RETURN '0,No Error'
```

3.23 CDC_ExitOutputTargetConnectError

Exit procedure CDC_ExitOutputTargetConnectError will be called if an error has occurred during a connection attempt to an SQL data source.

Parameter Stem:

Input:	X_OTCE_I
Output:	X_OTCE_O

Input Parameter:

X_OTCE_I.Target_Type — Type of data source:

1	ODBC
2	Db2 – DRDA Protocol
3	Oracle – OCI Interface
4	Db2 – direct call on the mainframe
6	MS SQL Server
7	DB2 – CLI Interface
8	MS SQL Server BCP Interface
9	Exasol
10	PostgreSQL
11	Teradata
12	MySQL
13	JDBC
14	MariaDB
15	Oracle – OCILIB Interface

X_OTCE_I.Connect_String — Connection parameter

X_OTCE_I.Connect_Error — Error text

Output Parameter:

X_OTCE_O.ReturnFlags — Return value

0	No further processing.
1	New connection attempt.

X_OTCE_O.Connect_String — Changed connection string for new connection attempt (optional).

Example for the exit:

```
CDC_ExitOutputTargetConnectError:

if pos("ISAM ERROR", upper(X_OTCE_I.Connect_Error)) > 0 then
do
      say "Some temporary connect error encountered. Waiting for 3 seconds"
      rc = sleep(3)
      X_OTCE_O.ReturnFlags = 1
      say "Connect retry"
end
else
do
      X_OTCE_O.ReturnFlags = 0
end

RETURN '0,No Error'
```

3.24 CDC_ExitLoaderControl

The exit procedure CDC_ExitLoaderControl is called before a line is written into the loader control file.

Parameter Stem:

Input:	X_LDRC_I
Output:	X_LDRC_O

Input Parameters:

X_LDRC_I.Loader_Type — Loader type for which this line is created:

ORACLE	Oracle
DB2/LUW	Db2 – Linux, UNIX, Windows
DB2/ZOS	Db2 – z/OS
MSSQL	Microsoft SQL Server

X_LDRC_I.Line_Type — Category of the generated line

1	General information
2	Start of a table
3	Field entry
4	End of a table
5	Delete of a table
6	Database control
9	End of control file

X_LDRC_I.Control_Line — The generated line

X_LDRC_I.Loader_TargetName — Name of output target from repository

X_LDRC_I.Loader_TargetConnection — Connection string of output target from repository

Output Parameters:

X_LDRC_O.ReturnFlags — Return value

0	Write this line
1	Do not write this line, no further processing

X_LDRC_O.Control_Line — Modified line

X_LDRC_O.Before_Line.n — Lines (1-n) that should be inserted in front of the generated line (optional)

X_LDRC_O.After_Line.n — Lines (1-n) that should be inserted following the generated line (optional)

Exit example:

```
CDC_ExitLoaderControl:
say X_LDRC_I.Line_Type X_LDRC_I.Control_Line
if X_LDRC_I.Line_Type = 2 then
do
      if word(X_LDRC_I.Control_Line, 1) = "INDDN" then
      do
            X_LDRC_O.After_Line.1 = "DISCARDDN SYSDISC"
      end
end
```

```
X_LDRC_O.ReturnFlags = 0

return '0, No Error'
```

3.25 CDC_ExitInputFileControl

The exit procedure CDC_ExitInputFileControl is called if an end-of-file condition of the input file has been detected during processing. The exit can return a new file name for further processing.

Parameter Stem:

Output:	X_IFC_O

Input Parameter:

none

Output Parameter:

X_IFC_O.ReturnFlags	Return value
	0 No further processing
	1 Parameter Input_Source_Name contains a new file name that should be processed
	n The exit should be called again after n seconds

Example:

```
CDC_ExitInputFileControl:

DirName = 'C:\data\'

/* Perform a DOS DIR command and extract filenames */
ADDRESS SYSTEM 'CMD /C DIR /b "'DirName'*.bin"' WITH INPUT NORMAL OUTPUT STEM DirData.
if rc <> 0 then
do
   /* wait for 10 seconds */
   X_IFC_O.ReturnFlags = 10
   return "0, No files"
end

X_IFC_O.ReturnFlags = 0
do i = 1 TO DirData.0
    if index(DirData.i, '.bin') > 0 then
    do
         /* Build fully qualified file name */
        Input_Source_Name = DirName||DirData.i
        X_IFC_O.ReturnFlags = 1
        say "NEW FILE="Input_Source_Name
        leave
    end
end

return "0, No error"
```

3.26 CDC_ExitSQLError

The exit procedure CDC_ExitSQLError is called if an error has occurred during the execution of an SQL statement. A decision can be made in the exit whether the error can be ignored.

Parameter Stem:
Input: X_SQE_I
Output: X_SQE_O

Input Parameters:

X_SQE_I.ReturnCode	The tcSCRIPT error return code
X_SQE_I.MessageText	The error message related to the return code
X_SQE_I.Timestamp	Timestamp of source change
X_SQE_I.SQL_DML	The SQL DML statement
X_SQE_I.Parm_Count	Number of parameter values for the SQL DMLstatement. It is 0 (zero) for array operations.

The following fields have an index from 1 to X_SQE.Parm_Count:

X_SQE_I.Parm_Name.suffix	Name of parameter
X_SQE_I.Parm_Type.suffix	SQL type of parameter
X_SQE_I.Parm_Length.suffix	SQL length of parameter
X_SQE_I.Parm_Scale.suffix	SQL scale of parameter
X_SQE_I.Parm_KeyNr.suffix	If > 0: Represents the parameter sequence number in the unique key
X_SQE_I.Parm_Null.suffix	1 value is NULL
X_SQE_I.Parm_Value.suffix	Parameter value

Target-specific Input Parameters:

MongoDB:

X_SQE_I.MongoDBDocument	The JSON document of the operation
X_SQE_I.MongoDBQuery	The JSON for the condition of the operation
X_SQE_I.MongoDBOptions	The JSON for the options, only for subdocuments

Adabas:

X_SQE_I.AdabasDatabaseNumber	Adabas database number
X_SQE_I.AdabasFileNumber	Adabas file number
X_SQE_I.AdabasISN	The ISN of the used CB (command blocks)
X_SQE_I.AdabasCommandCode	The command code of the used CB
X_SQE_I.AdabasResponseCode	The response code of the used CB
X_SQE_I.AdabasCommandOption1	Content of Command Option 1 of the used CB
X_SQE_I.AdabasCommandOption2	Content of Command Option 2 of the used CB
X_SQE_I.AdabasAdditions1	Content of Additions 1 of the used CB
X_SQE_I.AdabasAdditions2	Content of Additions 2 of the used CB

DL/I:

X_SQE_I.DLIDatabaseName	DLI database name
X_SQE_I.DLISegmentName	Segment name
X_SQE_I.DLILevelFeedback	PCB level feedback
X_SQE_I.DLIStatusCode	PCB status code
X_SQE_I.DLIProcessingOptions	Processing options
X_SQE_I.DLISensitiveSegments	Sensitive segments
X_SQE_I.DLIKeyFeedbackArea	KeyFeedbackArea of the dataset

Datacom:

X_SQE_I.DATACOMcommand	Used Datacom command
X_SQE_I.DATACOMReturnCode	Return code
X_SQE_I.DATACOMReasonCode	Reason code

X_SQE_I.DATACOMDatabaseID	Database ID
X_SQE_I.DATACOMFileName	Filename
X_SQE_I.DATACOMKeyName	Key name
X_SQE_I.DATACOMElementList	Element list

Output Parameter:

X_SQE_O.ReturnFlags		Return value
	0	Do not ignore the error.
	1	Ignore the error, continue processing

Example:

```
CDC_ExitSQLError:
say "***"
say "An SQL error has occurred"
say "X_SQE_I.ReturnCode " d2x(X_SQE_I.ReturnCode)
say "X_SQE_I.MessageText" X_SQE_I.MessageText
say "X_SQE_I.SQL_DML    " X_SQE_I.SQL_DML
say "X_SQE_I.Parm_Count " X_SQE_I.Parm_Count

do i = 1 to X_SQE_I.Parm_Count
       say i X_SQE_I.Parm_Name.i
end
X_SQE_O.ReturnFlags = 0

/* Just ignore DB2 constraint violations */
if index(X_SQE_I.MessageText, "-803") > 0 then
do
       X_SQE_O.ReturnFlags = 1
       say "Error ignored"
end

return "0, No error"
```

3.27 CDC_ExitReplicationControl

The exit procedure CDC_ExitReplicationControl is called during the replication processing of stage 3 when corresponding input and output table linkages are processed. The exit can evaluate and change field data and also change the processing type of operation (INSERT/UPDATE/DELETE).

Parameter Stem:

Input:	X_RPC_I
Output:	X_RPC_O

Input Parameters:

X_RPC_I.DataSourceType	Type of data source (refer to CDC_ExitS2LogRecordRead on page 109)
X_RPC_I.DataType	Type of data record (refer to CDC_ExitS2LogRecordRead on page 109)
X_RPC_I.Timestamp	Timestamp of data record if provided by the data source
X_RPC_I.UniqueId	Data source-specific unique ID of the data record if provided by the data source
X_RPC_I.Operation_Type	Operation type of data source (refer to CDC_ExitS2LogRecordRead on page 109)
X_RPC_I.InputType	Input data source type
X_RPC_I.InputTable	Name of input data source

X_RPC_I.OutputTarget	Name of output target
X_RPC_I.OutputTable	Name of output table
X_RP_I.Field_Count	Number of data fields

The following fields have an index from 1 to X_RP_I.Field_Count:

X_RPC_I.Field_Name.suffix	Field name
X_RPC_I.Field_Type.suffix	SQL data type
X_RPC_I.Field_Length.suffix	SQL length
X_RPC_I.Field_Scale.suffix	SQL scale
X_RPC_I.Field_CCSID.suffix	CCSID of field
X_RPC_I.Field_KeyNr.suffix	1 to n for key fields, otherwise 0
X_RPC_I.Field_Table.suffix	Field table number
X_RPC_I.Field_Level.suffix	Field table depth
X_RPC_I.Field_Index.suffix.n	Field table occurrence
X_RPC_I.Field_Null_BI.suffix	Field null indicator before image
X_RPC_I.Field_Value_BI.suffix	Field value before image
X_RPC_I.Field_Null_BI_Raw.suffix	Field null indicator input before image
X_RPC_I.Field_Value_BI_Raw.suffix	Field value input before image
X_RPC_I.Field_Null_AI.suffix	Field null indicator after image
X_RPC_I.Field_Value_AI.suffix	Field value after image
X_RPC_I.Field_Null_AI_Raw.suffix	Field null indicator input after image
X_RPC_I.Field_Value_AI_Raw.suffix	Field value input after image

Output Parameters:

X_RPC_O.Operation_Type	New operation type of data record:(refer to CDC_ExitS2LogRecordRead on page 109). Setting this to 0 discards the operation.

The following fields have an index from 1 to X_RPC_I.Field_Count. Only the fields whose contents should be changed must be filled:

X_RPC_O.Field_Null_AI.suffix	New null indicator in after image
X_RPC_O.Field_Value_AI.suffix	New field value in after image

Example:

```
OPR_INSERT = 1
OPR_UPDATE = 2
OPR_DELETE = 3

/* change any DELETE to UPDATE and change first field value (KEY)*/
IF X_RPC_I.Operation_Type = OPR_DELETE THEN
DO
  X_RPC_O.Operation_Type = OPR_UPDATE
  X_RPC_O.Field_Value_AI.1 = X_RPC_I.Field_Value_BI.1 + 16000000
  X_RPC_O.Field_Null_AI.1  = 0
  say "X_RPC_I.Field_Value_BI.1" X_RPC_I.Field_Value_BI.1
                        "-->  X_RPC_O.Field_Value_AI.1" X_RPC_O.Field_Value_AI.1
END
ELSE
  X_RPC_O.Operation_Type = X_RPC_I.Operation_Type

RETURN '0,OK'
```

3.28 CDC_ExitReplicationProcess

The exit procedure CDC_ExitReplicationProcess is invoked during the replication processing of stage 3 when the corresponding field is processed. The exit can evaluate the data, modify the data, and make the data contents available to other output fields.

Parameter Stem:

Input:	X_RP_I
Output:	X_RP_O

Input Parameters:

X_RP_I.Field_Name	Field name that has been defined for the replication processing
X_RP_I.Field_Number	Internal field number of the field that has been defined for the replication processing
X_RP_I.Field_Level	Field table level for output field table processing
X_RP_I.Field_Index	Field table index for output field table processing
X_RP_I.DataSourceType	Type of data source (refer to 3.17 CDC_ExitS2LogRecordRead)
X_RP_I.DataType	Type of data record (refer to 3.17 CDC_ExitS2LogRecordRead)
X_RP_I.Timestamp	Timestamp of data record if provided by the data source
X_RP_I.UniqueId	Data source-specific and unique ID of the data record if provided by the data source
X_RP_I.Operation_Type	Operation type of data record:(refer to 3.17 CDC_ExitS2LogRecordRead)
X_RP_I.InputType	Type of input data source
X_RP_I.InputTable	Name of input data source
X_RP_I.OutputTarget	Name of output target
X_RP_I.OutputTable	Name of output table
X_RP_Image_Data	Image data for undo/redo data records
X_RP_Image_Type	Type of image data (before/after)
X_RP_I.Field_Count	Number of data fields

The following fields have an index from 1 to X_RP_I.Field_Count:

X_RP_I.Field_Name.suffix	Field name
X_RP_I.Field_Type.suffix	SQL data type
X_RP_I.Field_Length.suffix	SQL length
X_RP_I.Field_Scale.suffix	SQL scale
X_RP_I.Field_CCSID.suffix	CCSID of field
X_RP_I.Field_KeyNr.suffix	1 to n for key fields, otherwise 0
X_RP_I.Field_Table.suffix	Field table number
X_RP_I.Field_Level.suffix	Field table depth
X_RP_I.Field_Index.suffix.n	Field table occurrence
X_RP_I.Field_Null.suffix	Field null indicator
X_RP_I.Field_Value.suffix	Field value

Output Parameter:

X_RP_O.Field_Count	Number of values returned. The first value is used for the actual processing. Additional values can be retrieved using the replication processing of type 4.

The following fields have an index from 1 to X_RP_O.Field_Count:

X_RP_O.Field_Null.suffix	Null indicator

X_RP_O.Field_Value.suffix Value

Example:

```
X_RP_O.Field_Count = 1
X_RP_O.Field_Null.1 = 0
X_RP_O.Field_Value.1 =
      X_RP_I.Field_Value.4""right("00000000"X_RP_I.Field_Value.5,8)
return '0, ok'
```

3.29 CDC_ExitDDLEvent

Exit procedure CDC_ExitDDLEvent is called if a DDL event is encountered. Some examples of DDL events:

- Adding or dropping a column to/from a table
- Changing the column data type
- Creating or dropping a primary key, etc.

With this exit the user can react to DDL events occurring in the source database and flexibly respond to any of them accordingly, for example automatically propagating changes to the target database, ignoring specific DDL events, informing a responsible person via email, etc.

Note: For executing the DDLEvent exit, DDL processing should be activated in the script settings (see parameter DDL_Flags).

Parameter Stem:
Input: X_DDL_I
Output: X_DDL_O

Input Parameters:

X_DDL_I.DDLFile Full name of the DDL protocol file. String value.

Event level variables

X_DDL_I.EventType Type of the DDL event. String value. Possible values:

For DB2/MVS:

'CREATE_TABLE'
'DROP_TABLE'
'RENAME_TABLE'

'SET_DATATYPE'
'ADD_COLUMN'
'DROP_COLUMN'
'RENAME_COLUMN'
'SET_DEFAULT'
'DROP_DEFAULT'

'ADD_UNIQUE_CONSTRAINT'
'DROP_UNIQUE_CONSTRAINT'

'ADD_CHECK_CONSTRAINT'
'DROP_CHECK_CONSTRAINT'
'ADD_PRIMARY_KEY'
'DROP_PRIMARY_KEY'
'ADD_FOREIGN_KEY'
'DROP_FOREIGN_KEY'
'CREATE_INDEX'
'DROP_INDEX'
'ALTER_INDEX_ADD_COLUMN'

For Adabas:

'DROP_TABLE'
'RENAME_TABLE'

'SET_DATATYPE'
'ADD_COLUMN'
'ACTIVATE_FIELD'
'DEACTIVATE_FIELD'

For Oracle:

'CREATE_TABLE'
'DROP_TABLE'
'RENAME_TABLE'

'ADD_COLUMN'
'MODIFY_COLUMN'
'DROP_COLUMN'
'RENAME_COLUMN'

'ADD_UNIQUE_CONSTRAINT'
'ADD_PRIMARY_KEY'
'DROP_PRIMARY_KEY'
'ADD_CHECK_CONSTRAINT'
'ADD_FOREIGN_KEY'
'DROP_CONSTRAINT'
'CREATE_INDEX'
'DROP_INDEX'

X_DDL_I.EventSource — Type of the source. String value.

X_DDL_I.EventTimestamp — Timestamp of the event. String value.

X_DDL_I.ParentEvent — Type of the parent DDL event; is defined for CREATE_TABLE only. String value. See below for explanation.

For Oracle there is an additional set of event-level string variables:

X_DDL_I.EventInfo.SysEvent — Type of the DDL change (like 'ALTER', 'DROP' etc.)
X_DDL_I.EventInfo.Dict_Obj_Owner — Owner of the DB object affected.
X_DDL_I.EventInfo.Login_User — DB user who issued the DDL command.
X_DDL_I.EventInfo.SQL_Text — Full text of the SQL command.

Multiple events and ParentEvent variable

Some DDL statements can, in fact, contain multiple DDL events and can be seen as a group of DDL statements just written in a shortened form and executed in a single transaction. In such cases the exit procedure is called for each of the events, one after another. If it is a CREATE TABLE statement, the exit variable X_DDL_I.ParentEvent is set.
Typical example:

```
create table t1(id int, primary key(id))
```

This statement defines two DDL events: CREATE_TABLE and ADD_PRIMARY_KEY. The exit procedure will be called twice, for CREATE_TABLE and then for ADD_PRIMARY_KEY. The second event (ADD_PRIMARY_KEY) has its X_DDL_I.ParentEvent variable set to 'CREATE_TABLE'.

Another example (Oracle):

```
alter table t1 add c1 int modify c2 number add c3 int constraint t1_pk primary key
```

Four events will be identified by tcVISION:
- ADD_COLUMN (column c1)
- MODIFY_COLUMN (column c2)
- ADD_COLUMN (column c3)
- ADD_PRIMARY_KEY (key column c3)

The exit will be called for each of them. The X_DDL_I.ParentEvent remains empty.

Note1: the current implementation defines ParentEvent variable only if CURRENT_TABLE event occurs.

Note2: all the events from the same DDL statement have the same time stamp.

Object List Variables

The following exit variables are defined irrespective of the event type, source, etc. They can be used for logging or for similar purposes for which the event type or object type does not matter.

X_DDL_I.Objects.Count — Number of the DDL objects affected by the event. Integer.

For the variables below <obindex> can take values between 1 and X_DDL_I.Objects.Count.

X_DDL_I.Objects.<obindex>.Type
: Type of the DDL object. String.

X_DDL_I.Objects.<obindex>.BIData_Exists
: Integer value that is 0 if Before Image data exists for the object. If not, the value is 1.

X_DDL_I.Objects.<obindex>.AIData_Exists
: Integer value that is 0 if After Image data exists for the object. If not, the value is 1.

X_DDL_I.Objects.<obindex>.BIData.Fields.Count
: Number of the fields in the object's BI-Data field list. Integer.

X_DDL_I.Objects.<obindex>.AIData.Fields.Count
: Number of the fields in the object's AI-Data field list. Integer.

For the variables below <findex> can take values between 1 and X_DDL_I.Objects.<obindex>.BIData.Fields.Count or X_DDL_I.Objects.<obindex>.AIData.Fields.Count respectively.

X_DDL_I.Objects.<obindex>.BIData.Fields.<findex>.Field_Name
: Name of the field from BI-Data field list. String.

X_DDL_I.Objects.<obindex>.AIData.Fields.<findex>.Field_Name
: Name of the field from AI-Data field list. String.

X_DDL_I.Objects.<obindex>.BIData.Fields.<findex>.Field_Value
Value of the field from BI-Data field list.

X_DDL_I.Objects.<obindex>.AIData.Fields.<findex>.Field_Value
Value of the field from AI-Data field list.

Example:

```
say "EventType:" || X_DDL_I.EventType
say "Data source:" || X_DDL_I. EventSource
say "Event timestamp:" || X_DDL_I.EventTimestamp

say 'Number of affected objects:' X_DDL_I.Objects.Count

do I = 1 to X_DDL_I.Objects.Count
  say "Object"I":"
  say "Type: " || X_DDL_I.Objects.I.Type
  say "Fields:"
  if X_DDL_I.Objects.I.BIData_Exists = 1 then do
    say "BI Fields.Count: " X_DDL_I.Objects.I.BIData.Fields.Count
    do J = 1 to X_DDL_I.Objects.I.BIData.Fields.Count
      say X_DDL_I.Objects.I.BIData.Fields.J.Field_Name || "="
                  X_DDL_I.Objects.I.BIData.Fields.J.Field_Value
    end
  end
  else do
    say "BI Data is not defined"
  end
end
```

Named Object Variables

The definition of the following exit variables depends on the event, object type, and source.
For the list of events see X_DDL_I.EventType variable above.

See below for the list of objects created in each case and list of variables available for each object.

DDL objects could be:

Table
Column
CheckConstr
UniqueConstr
PK
FK
Index
KeyColumn
Constraint (Oracle specific)

Object-Specific Variables for Db2/MVS (X_DDL_I.EventSource = 'DB2_MVS')

In case of CREATE_TABLE event are defined:
- **Table** object, AI Data only
- List of **Column** objects, AI Data only

In case of DROP_TABLE event:
- **Table** object, BI Data only

In case of RENAME_TABLE event:
- **Table** object, BI, and AI Data

In case of ADD_COLUMN event:
- **Table** object, BI, and AI Data
- List of **Column** objects, AI Data only

In case of DROP_COLUMN event:
- **Table** object, BI, and AI Data
- List of **Column** objects, BI Data only (*)

In case of RENAME_COLUMN event:
- **Table** object, BI, and AI Data
- List of **Column** objects, BI, and AI Data (*)

In case of SET_DEFAULT event:
- **Table** object, BI, and AI Data
- List of **Column** objects, BI, and AI Data (*)

In case of DROP_DEFAULT event:
- **Table** object, BI, and AI Data
- List of **Column** objects, BI, and AI Data (*)

In case of SET_DATATYPE event:
- **Table** object, BI, and AI Data
- List of **Column** objects, BI, and AI Data (*)

In case of ADD_CHECK_CONSTRAINT event:
- **Table** object, BI, and AI Data
- **CheckConstr** object, AI Data only

In case of DROP_CHECK_CONSTRAINT event:
- **Table** object, BI, and AI Data
- **CheckConstr** object, BI Data only

In case of ADD_UNIQUE_CONSTRAINT event:
- **Table** object, BI, and AI Data
- **UniqueConstr** object, AI Data only
- List of **KeyColumn** objects, AI Data only

In case of DROP_UNIQUE_CONSTRAINT event:
- **Table** object, BI, and AI Data
- **UniqueConstr** object, BI Data only
- List of **KeyColumn** objects, BI Data only

In case of ADD_PRIMARY_KEY event:
- **Table** object, BI, and AI Data
- **PK** object, AI Data only
- List of **KeyColumn** objects, AI Data only

In case of DROP_PRIMARY_KEY event:
- **Table** object, BI, and AI Data
- **PK** object, BI Data only
- List of **KeyColumn** objects, BI Data only

In case of ADD_FOREIGN_KEY event:
- **Table** object, BI, and AI Data
- **FK** object, AI Data only
- List of **KeyColumn** objects, AI Data only

In case of DROP_FOREIGN_KEY event:
- **Table** object, BI, and AI Data
- **FK** object, BI Data only

- List of **KeyColumn** objects, BI Data only

In case of CREATE_INDEX event:
- **Table** object, BI, and AI Data
- **Index** object, AI Data only
- List of **KeyColumn** objects, AI Data only

In case of DROP_INDEX event:
- **Table** object, BI, and AI Data
- **Index** object, BI Data only

In case of ALTER_INDEX_ADD_COLUMN event:
- **Table** object, BI, and AI Data
- **Index** object, BI, and AI Data
- List of **KeyColumn** objects, AI Data only (*)

(*): In these cases the list of the column objects has only 1 element (see also **Counter variables** below)

In most cases objects correspond to one of the Db2 catalog tables (SYSIBM.SYSTABLES, SYSIBM.SYSCOLUMNS etc.), the variables representing the columns of these tables.

- Variables available for the **Table** object:

<prefix>.Name
<prefix>.Type
<prefix>.Creator
<prefix>.DBID
<prefix>.OBID
<prefix>.Colcount (*)
<prefix>.Version (*)
<prefix>.PSID (*)(**)

whereas <prefix> can take values X_DDL_I.Table.BIData or X_DDL_I.Table.AIData.

(*): This variable is **not** defined in case of CREATE_INDEX, DROP_INDEX or ALTER_INDEX_ADD_COLUMN event.
(**): This variable is **only** defined in case of CREATE_TABLE event.

For more info about possible values, their meaning, etc. please see Db2 documentation on SYSTABLES and SYSTABLESPACE catalog tables.

Example:

```
if X_DDL_I.EventSource = 'DB2_MVS' then do
  if X_DDL_I.EventType = 'CREATE_TABLE' then do
    say "Table " || X_DDL_I.Table.AIData.Name || " was created"
  end
end
```

- Variables available for the **Column** object:

<prefix>.Name
<prefix>.Colno
<prefix>.Typename
<prefix>.Datatypeid
<prefix>.Length
<prefix>.Scale
<prefix>.Length2
<prefix>.CCSID
<prefix>.Nulls
<prefix>.Default

<prefix>.Defaultvalue
<prefix>.Hidden

whereas <prefix> can take values
X_DDL_I.Columns.<index>.BIData or X_DDL_I.Columns.<index>.AIData.

For more info about possible values, their meaning, etc. see Db2 documentation on SYSCOLUMNS catalog table.

Example:

```
if X_DDL_I.EventSource = 'DB2_MVS' then do
  if X_DDL_I.EventType = 'RENAME_COLUMN' then do
    say "Old column name: " ||  X_DDL_I.Columns.1.BIData.Name
    say "New column name: " ||  X_DDL_I.Columns.1.AIData.Name
  end
end
```

- Variables available for the **CheckConstr** object:

<prefix>.Checkname
<prefix>.TBName
<prefix>.TBOwner
<prefix>.CheckCondition

whereas <prefix> can take values
X_DDL_I.CheckConstr.BIData or X_DDL_I.CheckConstr.AIData.

For more info about possible values, their meaning, etc. see Db2 documentation on SYSCHECKS catalog table.

Example:

```
if X_DDL_I.EventSource = 'DB2_MVS' then do
  if X_DDL_I.EventType = 'ADD_CHECK_CONSTRAINT' then do
    say "Check constraint created"
    say "... name: " ||  X_DDL_I.CheckConstr.AIData.Checkname
    say "... condition: " ||  X_DDL_I.CheckConstr.AIData.CheckCondition
  end
end
```

- Variables available for the **UniqueConstr** object and **PK** object:

<prefix>.Constname
<prefix>.Type
<prefix>.TBName
<prefix>.TBCreator

whereas <prefix> can take values
X_DDL_I.UniqueConstr.BIData or X_DDL_I.UniqueConstr.AIData for a UniqueConstr object
and
X_DDL_I.PK.BIData or X_DDL_I.PK.AIData for a PK object.

For more info about possible values, their meaning, etc. see Db2 documentation on SYSTABCONST catalog table.

Example:

```
if X_DDL_I.EventSource = 'DB2_MVS' then do
```

```
  if X_DDL_I.EventType = 'DROP_UNIQUE_CONSTRAINT' then do
    say "Unique constraint dropped:" X_DDL_I.UniqueConstr.BIData.Constname
  end
  else
  if X_DDL_I.EventType = 'DROP_PRIMARY_KEY' then do
    say "Primary key constraint dropped:" X_DDL_I.PK.BIData.Constname
  end
end
```

- Variables available for the **FK** object:

<prefix>.Relname
<prefix>.Creator
<prefix>.TBName
<prefix>.RefTBName
<prefix>.RefTBCreator
<prefix>.DeleteRule
<prefix>.Enforced
<prefix>.IXName
<prefix>.IXOwner

whereas <prefix> can take values
X_DDL_I.FK.BIData or X_DDL_I.FK.AIData.

For more info about possible values, their meaning, etc. see Db2 documentation on SYSRELS catalog table.

Example:

```
if X_DDL_I.EventSource = 'DB2_MVS' then do
  if X_DDL_I.EventType = 'ADD_FOREIGN_KEY' then do

    say "New FK constraint was created"
    say "Primary table: " ||  X_DDL_I.FK.AIData.RefTBName
    say "Secondary table: " ||  X_DDL_I.FK.AIData.TBName

    if X_DDL_I.FK.AIData.IXName = "" then do
      say "The parent key is the primary table's PK"
    end
    else do
      say "The parent index is " ||  X_DDL_I.FK.AIData.IXName
    end
  end
end
```

- Variables available for the **Index** object:

<prefix>.Name
<prefix>.TBName
<prefix>.TBCreator
<prefix>.UniqueRule
<prefix>.ColCount

whereas <prefix> can take values
X_DDL_I.Index.BIData or X_DDL_I.Index.AIData.

For more info about possible values, their meaning, etc. see Db2 documentation on SYSINDEXES catalog table.

Example:

```
if X_DDL_I.EventSource = 'DB2_MVS' then do
  if X_DDL_I.EventType = 'CREATE_INDEX' then do
    say "Index " || X_DDL_I.Index.AIData.Name || " was created"
  end
end
```

- Variables available for the **KeyColumn** object:

<prefix>.Colname
<prefix>.Colseq
<prefix>.Ordering (*)

whereas <prefix> can take values
X_DDL_I.KeyColumns.<index>.BIData or X_DDL_I.KeyColumns.<index>.AIData.

(*): This variable is **only** defined in case of CREATE_INDEX, DROP_INDEX or ALTER_INDEX_ADD_COLUMN event.

The KeyColumn object represents a column that is part of a UniqueConstr, PK, FK or Index object.
For more info about possible values, their meaning, etc. see Db2 documentation on SYSKEYCOLUSE, SYSFOREIGNKEYS and SYSKEYS catalog tables.

Example:

```
if X_DDL_I.EventSource = 'DB2_MVS' then do
  if X_DDL_I.EventType = 'ADD_PRIMARY_KEY' then do
    say "PK consists of" X_DDL_I.Keycolumns.Count "columns"
    say "They are:"
      do I = 1 to X_DDL_I.Keycolumns.Count
        say X_DDL_I.Keycolumns.I.AIData.Colname
      end
  end
end
```

Counter Variables

Depending on the event type the following counter variables could be defined:

X_DDL_Coulmns.Count	Number of columns in the list of the Column objects. It is defined for CREATE_TABLE, ADD_COLUMN (*), DROP_COLUMN (*), RENAME_COLUMN (*), SET_DATATYPE (*), SET_DEFAULT (*), DROP_DEFAULT (*) events. (*): In these cases the column counter is 1.
X_DDL_KeyCoulmns.Count	Number of key columns in the list of the KeyColumn objects. It is defined for ADD_UNIQUE_CONSTRAINT, DROP_UNIQUE_CONSTRAINT, ADD_PRIMARY_KEY, DROP_PRIMARY_KEY, ADD_FOREIGN_KEY, DROP_FOREIGN_KEY, CREATE_INDEX events.

Example:

```
if X_DDL_I.EventSource = 'DB2_MVS' then do
  if X_DDL_I.EventType = 'ADD_PRIMARY_KEY' then do
```

```
    sql = "ALTER TABLE" X_DDL_I.Table.AIData.Name "ADD CONSTRAINT"
X_DDL_I.PK.AIData.Constname "PRIMARY KEY("

    do I = 1 to X_DDL_I.Keycolumns.Count
      sql = sql || X_DDL_I.Keycolumns.I.AIData.Colname
      if I \= X_DDL_I.Keycolumns.Count then do
        sql = sql || ", "
      end
    end

    sql = sql || ")"

    say "DDL SQL:" || sql
  end
end
```

Object-Specific Variables for Adabas (X_DDL_I.EventSource = 'ADABAS')

- Variables available for the **Table** object:

<prefix>.DBID
<prefix>.FileNum

whereas <prefix> can take values X_DDL_I.Table.BIData or X_DDL_I.Table.AIData.

Example:

```
if X_DDL_I.EventSource = 'ADABAS' then do
  if X_DDL_I.EventType = 'RENAME_TABLE' then do
      sql = "ADADBS RENUMBER FILES=" || X_DDL_I.Table.BIData.FileNum || "," ||
X_DDL_I.Table.AIData.FileNum
  end
end
```

- Variables available for the **Column** object:

<prefix>.Name
<prefix>.Level
<prefix>.Length
<prefix>.Format
<prefix>.Option1
<prefix>.Option2
<prefix>.DTEMask (*)
<prefix>.DTEMask1 (*)

whereas <prefix> can take values
X_DDL_I.Columns.<index>.BIData or X_DDL_I.Columns.<index>.AIData.

(*): This variable is only defined in case DT option was specified (date-time edit mask) for example: ADADBS FNDEF='1,D1,8,U,DT=E(DATE)'

Object-Specific Variables for Oracle (X_DDL_I.EventSource = 'ORACLE')

For all Oracle events the following variables are always defined:

X_DDL_I.EventInfo.SysEvent	Type of the DDL action. Possible values are "CREATE", "ALTER", "DROP"
X_DDL_I.EventInfo.Dict_Obj_Owner	Owner (schema) of the object affected
X_DDL_I.EventInfo.Login_User	Oracle user who executed the DDL command
X_DDL_I.EventInfo.SQL_Text	Full text of the DDL command

In case of CREATE_TABLE event are defined:
- **Table** object, AI Data only

In case of DROP_TABLE event:
- **Table** object, BI Data only

In case of RENAME_TABLE event:
- **Table** object, BI, and AI Data

In case of ADD_COLUMN event:
- **Table** object, BI, and AI Data
- List of **Column** objects, AI Data only (*)

In case of MODIFY_COLUMN event:
- **Table** object, BI, and AI Data
- List of **Column** objects, AI Data only (*)

In case of DROP_COLUMN event:
- **Table** object, BI, and AI Data
- List of **Column** objects, BI Data only (*)

In case of RENAME_COLUMN event:
- **Table** object, BI, and AI Data
- List of **Column** objects, BI, and AI Data (*)

In case of ADD_UNIQUE_CONSTRAINT event:
- **Table** object, BI, and AI Data
- **UniqueConstr** object, AI Data only
- List of **KeyColumn** objects, AI Data only

In case of ADD_PRIMARY_KEY event:
- **Table** object, BI, and AI Data
- **PK** object, AI Data only
- List of **KeyColumn** objects, AI Data only

In case of DROP_PRIMARY_KEY event:
- **Table** object, BI, and AI Data
- **PK** object, BI Data only

In case of ADD_CHECK_CONSTRAINT event:
- **Table** object, BI, and AI Data
- **CheckConstr** object, AI Data only

In case of ADD_FOREIGN_KEY event:
- **Table** object, BI, and AI Data
- **FK** object, AI Data only
- List of **KeyColumn** objects, AI Data only

In case of DROP_CONSTRAINT KEY event:
- **Table** object, BI, and AI Data
- **Constraint** object, BI Data only

In case of CREATE_INDEX event:
- **Table** object, BI, and AI Data
- **Index** object, AI Data only
- List of **KeyColumn** objects, AI Data only

In case of DROP_INDEX event:
- **Table** object, BI, and AI Data
- **Index** object, BI Data only

(*): In these cases the list of the column objects has only 1 element (see also **Counter variables** below)

- Variables available for the **Table** object:

<prefix>.Name
<prefix>.Schema
<prefix>.OBID

whereas <prefix> can take values X_DDL_I.Table.BIData or X_DDL_I.Table.AIData.

Example:

```
if X_DDL_I.EventSource = 'ORACLE' then do
  if X_DDL_I.EventType = 'MODIFY_COLUMN' then do
    say "Table " || X_DDL_I.Table.AIData.Schema || """."""" ||
        X_DDL_I.Table.AIData.Name || " was altered"
  end
end
```

- Variables available for the **Column** object:

<prefix>.Name
<prefix>.Typename
<prefix>.Length
<prefix>.Scale
<prefix>.Semantics (*)
<prefix>.Nulls (**)
<prefix>.Default (**)
<prefix>.Defaultvalue (**)

whereas <prefix> can take values
X_DDL_I.Columns.<index>.BIData or X_DDL_I.Columns.<index>.AIData

(*): for CHAR, VARCHAR, VARCHAR2 data types
(**): if specified in the DDL statement

Example:

```
if X_DDL_I.EventSource = 'ORACLE' then do
  if X_DDL_I.EventType = 'ADD_COLUMN' then do
    say "New column added"
    say "... name: " ||  X_DDL_I.Columns.1.AIData.Name
    say "... type name: " ||  X_DDL_I.Columns.1.AIData.Typename
  end
end
```

- Variables available for the **CheckConstr** object:

<prefix>.Constname (*)
<prefix>.TBName
<prefix>.TBSchema
<prefix>.CheckCondition

whereas <prefix> can take values
X_DDL_I.Columns.<index>.BIData or X_DDL_I.Columns.<index>.AIData

(*): if specified in the DDL statement

Example:

```
if X_DDL_I.EventSource = 'ORACLE' then do
  if X_DDL_I.EventType = 'ADD_CHECK_CONSTRAINT' then do
    say "Check constraint created"
    say "... condition: " ||  X_DDL_I.CheckConstr.AIData.CheckCondition
  end
end
```

- Variables available for the **UniqueConstr** object, **PK** object and **Constraint** object:

<prefix>.Constname (*)
<prefix>.Type (**)
<prefix>.TBName
<prefix>.TBSchema

whereas <prefix> can take values
X_DDL_I.UniqueConstr.BIData or X_DDL_I.UniqueConstr.AIData for a UniqueConstr object
and
X_DDL_I.PK.BIData or X_DDL_I.PK.AIData for a PK object.

(*): if specified in the DDL statement
(**): can take values 'U' (unique constraint), 'P' (primary key) or 'A' (any constraint, in case of DROP_CONSTRAINT event)

Example:

```
if X_DDL_I.EventSource = 'ORACLE' then do
  if X_DDL_I.EventType = 'ADD_UNIQUE_CONSTRAINT' then do
    say "Unique constraint added"

    if (X_DDL_I.UniqueConstr.AIData.Constname \=
"X_DDL_I.UNIQUECONSTR.AIDATA.CONSTNAME")) then do
      say "... constraint name: " X_DDL_I.UniqueConstr.AIData.Constname
    end

else
  if X_DDL_I.EventType = 'ADD_PRIMARY_KEY' then do
    say "Primary key added"

    if (X_DDL_I.PK.AIData.Constname \= "X_DDL_I.PK.AIDATA.CONSTNAME")the do
      sql = sql "...  PK name:" X_DDL_I.PK.AIData.Constname
    end
  end
end
```

- Variables available for the **FK** object:

<prefix>.Constname (*)
<prefix>.TBName

<prefix>.TBSchema
<prefix>.RefTBName
<prefix>.DeleteRule (**)

whereas <prefix> can take values
X_DDL_I.FK.BIData or X_DDL_I.FK.AIData.

(*): if specified in the DDL statement
(**): can take values 'C' (CASCADE) or 'N' (SET NULL)

Example:

```
if X_DDL_I.EventSource = 'ORACLE' then do
  if X_DDL_I.EventType = 'ADD_FOREIGN_KEY' then do

    say "New FK constraint was created"
    say "... primary table: " ||  X_DDL_I.FK.AIData.RefTBName
    say "... secondary table: " ||  X_DDL_I.FK.AIData.TBName

    if X_DDL_I.FK.AIData.DeleteRule = 'C' then do
      say "... delete rule: ON DELETE CASCADE"
    end
    else
    if X_DDL_I.FK.AIData.DeleteRule = 'N' then do
      say "... delete rule: ON DELETE SET NULL"
    end
  end
end
```

- Variables available for the **Index** object:

<prefix>.Name
<prefix>.Schema
<prefix>.OBID
<prefix>.UniqueRule (*) (**)

whereas <prefix> can take values
X_DDL_I.Index.BIData or X_DDL_I.Index.AIData.

(*): is set to 'U' (UNIQUE) for a unique index
(**): this variable is only set in case of CREATE_INDEX event

Example:

```
if X_DDL_I.EventSource = 'ORACLE' then do
  if X_DDL_I.EventType = 'CREATE_INDEX' then do
    say "Index " || X_DDL_I.Index.AIData.Name || " was created"
  end
end
```

- Variables available for the **KeyColumn** object:

<prefix>.Colname

whereas <prefix> can take values X_DDL_I.KeyColumns.<index>.AIData.

The KeyColumn object represents a column that is part of a UniqueConstr, PK, FK or Index object.

Example:

```
if X_DDL_I.EventSource = 'ORACLE' then do
  if X_DDL_I.EventType = 'ADD_PRIMARY_KEY' then do
    say "PK consists of" X_DDL_I.Keycolumns.Count "columns"
    say "They are:"
      do I = 1 to X_DDL_I.Keycolumns.Count
        say X_DDL_I.Keycolumns.I.AIData.Colname
      end
  end
end
```

Counter Variables

Depending on the event type the following counter variables could be defined:

X_DDL_Coulmns.Count	Number of columns in the list of the Column objects (currently the events handled can only contain a single column and thus, column counter is always 1)
X_DDL_KeyCoulmns.Count	Number of key columns in the list of the KeyColumn objects. It is defined for ADD_UNIQUE_CONSTRAINT, ADD_PRIMARY_KEY, ADD_FOREIGN_KEY events.

Output Parameters:

X_DDL_O.ReturnFlags

Return value:

0	Continue processing
1	Continue processing, but ignore this particular event

Example (no further processing in tcScript for "ADD_COLUMN" event):

```
if X_DDL_I.EventType = 'ADD_COLUMN' then do
      X_DDL_O.ReturnFlags = 1
      RETURN '0, Ignore'
end
```

4 High-Performance Exit Module

This chapter covers the high-performance exit load modules. With predefined exit points these load modules can process parts of the script exits described in chapter 3, Exit Procedures for Data Scripts.
High-performance exits are very useful for large data volumes because the exits run as compiled code, hence resulting in very efficient execution times.

4.1 Implementation of the HP Exits

4.1.1 Mainframe Systems

HP exits are load modules written in the Assembler language. They are loaded before the first script invocation.
Optional entry points can be specified in the exit for initialization purposes (e.g. storage allocation, open of files, read of script variables, etc.) or termination (deallocation of storage, close of files, rewrite of script variables, etc.).
During the script processing the defined entry points in the exit will be called and the corresponding parameters will be supplied.

Each call of an exit function receives a pointer to an array of script functions like the opening of files and the reading and writing of script variables.

COPY file TCSEXIT contains all necessary definitions to develop an Assembler HP exit.

4.1.2 UNIX Systems, Windows

HP exits are libraries written in the C language (DLL, so, a) that are dynamically loaded at runtime.
Optional entry points can be specified in the exit for initialization purposes (e.g. storage allocation, open of files, read of script variables, etc.) or termination (deallocation of storage, close of files, rewrite of script variables, etc.).
The defined entry points in the exit are called during the script processing and the corresponding parameters are supplied.

Each call of an exit function receives a pointer to an array of script functions like the opening of files and the reading and writing of script variables.

For the implementation of user-defined exits a sample package can be requested from the local tcVISION support which contains C code, repository structures, scripts, and sample data.

4.2 HP Field Exit

The system-wide load module with the fixed name of 'TCSFEXIT' corresponds to exit:

CDC_ExitInputField.

114

The exit can be used for the user-specific processing of data fields. It communicates with the calling system via parameter structures.

If defined, entry point 'FE_Initialize' is called at the beginning of the processing to allocate storage (as an example). If defined at the end of the processing, entry point 'FE_Terminate' is called (e.g. to deallocate storage).

Entry point 'FE_Exit_Before' is called whenever data from a data source must be converted to an SQL data type.

For the available example, input table 'DEFAULT.DB2/LUW.SAMPLE.DB2INST1.EMPLOYEE' and output table 'DEFAULT.DEMO_EXIT.DB2INST1.EMPLOYEE' are used. The tables are part of repository backup 'tcVISION-Repository-PartBackup-Demo-Fld-Exit.tcVRCF'. In addition, file 'fld_input.s0' is required which contains the sample data. The corresponding script is 'DEMO_FLD_EXIT.TSF'. The paths used in the script for the S0 and output file must be adapted to meet the individual testing environment.

The structures and functions explained in the following text are extracts of the C sample files and are not complete.

4.2.1 Explanations to the Structures

The structures can be found in the header file 'tcsexit.h'.

USERDATASTRUCT

```
typedef struct __userdatastruct {
  FILE*              fLog;
  // counters
  unsigned long long ullRunTime;
  unsigned int       nLookups;
  unsigned int       nInserts;
  unsigned int       nUpdates;
  unsigned int       nDeletes;
  unsigned int       nTruncates;
  unsigned int       nErrors;
  unsigned int       nStarttransactions;
  unsigned int       nCommits;
  unsigned int       nRollbacks;
  char*              pszBuffer;
} USERDATASTRUCT, *PUSERDATASTRUCT;
```

This structure contains variables that can be available from the first call of the exit using 'FE_Initialize' and through all calls from 'FE_Exit_Before' until the exit ends with 'FE_Terminate'. All important events that occur during the exit execution can be logged. The structure is used for user data and can be modified to meet the individual requirements.

FE_FLD_DESC

```
typedef struct {
  short          Version;                    // Input        Version of this structure
  short          StructureLength;            // Input        Length of this structure
  short          InFieldType;                // Input        Input Field Type
  short          InFieldCCSID;               // Input        Input CCSID
  short          InFieldScale;               // Input        Input Field Field Scale
  int            InFieldLength;              // Input        Length of field input data in bytes
  int            InFieldOffset;              // Input        Input Field offset in input record
  int            OutFieldLength;             // Input/Output Output Field Length in Bytes or Digits
  short          OutFieldType;               // Input/Output Output SQL Field Type
  short          OutFieldCCSID;              // Input/Output Output CCSID
  short          OutFieldScale;              // Input/Output Output Field Scale
  short          TableLevel;                 // Input        Table level depth
  unsigned short TableIndex[MAX_TABLE_DEPTH]; // Input        Field indexes in table structures
  short          UserFieldExitNumber;        // Input        User Field exit Number
  short          FieldNameLength;            // Input        Length of Field Name
```

```
  char           *FieldName;                // Input        Pointer to Fieldname
  void           *InFieldData;              // Input        Pointer to Input Data
  int            InFieldDataLength;         // Input        Length of Input data
  void           *InRecord;                 // Input        Pointer to original input record
  int            InRecordLength;            // Input        Length of original Input record
  void           *OutFieldData;             // Input/Output Pointer to Output Data
  int            OutFieldDataLength;        // Input/Output Length of Output Data
  unsigned short FieldFlag;                 // Input/Output Output processing flags
    #define FIELD_IS_NULL    0x0001         // Field Value is NULL (may be set on entry and changed on exit)
} FE_FLD_DESC, *PFE_FLD_DESC;
```

This structure is passed to function 'FE_Exit_Before'. It contains all information relevant to the field that is processed by the exit.

4.2.2 Explanations to the Functions

The functions can be found in C-file 'fe_exit.cc', the helper (assisting) functions can be found in C++-file 'tcsutils.cc'.

The functions of the field exit are called by the script if an exit module with a name of 'tcsfexit.dll' containing the functions described in this chapter can be found.

FE_Initialize

This optional function is called once at the start of the replication processing for which a field exit has been defined. In this example, storage is allocated for structure USERDATASTRUCT and assigned to variable 'UserData', hence this storage area is available to all further functions of the field exit. In addition a log file that can be used for trace entries is opened. In case of an error, variable 'ErrorMsg' is filled with an error message and the function is terminated with a return value not equal to 0 (zero). The error message can be up to 1000 characters in length.

```
TCS_EXPORT(int) FE_Initialize(char *ErrorMsg, void **UserData, C_SCRIPT_FUNCTIONS *FunctionTable)
{
  if (NULL == UserData)
  {
    snprintf(ErrorMsg, 50, "Call without UserData set.");
    return (8);
  };

  PUSERDATASTRUCT pUser = (PUSERDATASTRUCT) malloc(sizeof(USERDATASTRUCT));

  if (pUser == NULL)
  {
    snprintf(ErrorMsg, 50, "Memory allocation error.");
    return (8);
  };

  memset(pUser, 0, sizeof(USERDATASTRUCT));
  *UserData = pUser;

#ifdef VERBOSE
  pUser->fLog = fopen("C:\\Temp\\fld_exit.log", "w");
  if (NULL == pUser->fLog)
  {
    snprintf(ErrorMsg, 50, "Unable to open logfile.");
    return (8);
  };
#else
  pUser->fLog = NULL;
#endif

  return (0);
}
```

FE_Exit_Before

This function is called for every field of the data record which has the exit flag set in the repository that indicates an HP exit and has a value defined for user exit (variable

'UserFieldExitNumber' of structure FE_FLD_DESC). Different program sections can process the data of the field depending on the exit number. If the function is called with an unknown exit number, variable 'ErrorMsg' is filled with a message and the function is terminated with a return value of not equal 0 (zero). The error message can be up to 1000 characters in length.

The exit programmer must understand the data structure. In the following examples the programmer must be aware that exit number 1 is related to a CHARACTER type field, exit number 2 is related to a field of type SHORT, etc. It also has to be known how a specific format is stored in the database (BIG-ENDIAN, LITTLE-ENDIAN, format of data types DATE, TIME, TIMESTAMP, DECIMAL, etc.).

In this example, the string 'MANAGER ' is replaced by the string 'CEO' in all fields.

```
TCS_EXPORT(int) FE_Exit_Before(char *ErrorMsg, void **UserData, C_SCRIPT_FUNCTIONS *FunctionTable, PFE_FLD_DESC pfe_desc)
{
  if (pfe_desc && pfe_desc->InFieldLength != 0)
  {
    PUSERDATASTRUCT pUser = (PUSERDATASTRUCT) *UserData;

    switch (pfe_desc->UserFieldExitNumber)
    {
      case 1:
      {
        // changing character data
        if (!strcmp((char*)pfe_desc->InFieldData, "MANAGER " ) )
        {
          memcpy(pfe_desc->OutFieldData, "CEO     ", 8);
          pfe_desc->OutFieldDataLength = 8;

          if (pUser)
            pUser->nUpdates++;
        }
        break;
      }

      . . .

      default:
      {
        sprintf(ErrorMsg, "Unknown exit number %d in FE_Exit_Before", pfe_desc->UserFieldExitNumber);
        return(99);
      }
    }
  }

  return(0);
}
```

It is also possible to change numeric values in the field data. To retrieve the actual field value, the numeric format of the underlying database must be known (in the example a 2-byte SHORT INTEGER in BIG-ENDIAN format). It is possible to write the number as a character string. This requires the field type to be appropriately defined (SQL_CHAR_452). The conversion of the value in the actual defined target format is performed by tcVISION. The same applies to 4-byte integer and 8-byte big integer. The only difference is that more bytes must be reserved to retrieve the data.

```
      case 2:
      {
        // changing integer data
        int nEdLevel = 0;
        nEdLevel = ((char*)pfe_desc->InFieldData)[0] * 256 + ((char*)pfe_desc->InFieldData)[1];
        if (nEdLevel < 100)
          nEdLevel += 100;

        pfe_desc->OutFieldType = SQL_CHAR_452;
        pfe_desc->OutFieldDataLength = sprintf((char*)pfe_desc->OutFieldData, "%d", nEdLevel);

        if (pUser)
          pUser->nUpdates++;
        break;
      }
```

The numeric format of the underlying database must also be known to retrieve the actual value of a field type of DECIMAL (here in packed format). The assisting function 'print_packed_decimal' of the sample exit is used. After that, the number can be transformed into a variable of type DOUBLE using standard C-functions. It is possible to write the number as a character string. This requires the field type to be appropriately defined (SQL_CHAR_452). The conversion of the value in the actual defined target format is performed by tcVISION.

```
      case 3:
      {
        // changing decimal data
        unsigned char szBonus[20] = {0};
        print_packed_decimal((unsigned char*)pfe_desc->InFieldData, pfe_desc->InFieldLength, pfe_desc->InFieldScale,
szBonus);
        double dbBonus = atof((char*)szBonus);
        dbBonus *= 2;

        pfe_desc->OutFieldType = SQL_CHAR_452;
        pfe_desc->OutFieldDataLength = sprintf((char*)pfe_desc->OutFieldData, "%f", dbBonus);

        if (pUser)
          pUser->nUpdates++;
        break;
      }
```

To retrieve the current value of a field type of DATE (same applies to TIME and TIMESTAMP), the format of the underlying database is important (here we use the standard character format). The values of year, month, and day are evaluated and adapted. The writing of the new field value is performed as character string in the usual format.

Data type	Format	Length
DATE	YYYY-MM-DD	10
TIME	hh.mm.ss	8
TIMESTAMP	YYYY-MM-DD-hh.mm.ss.ffffff	26

```
      case 4:
      {
        // changing date data
        char szHireDate[20] = {0};
        strncpy(szHireDate, (char*)pfe_desc->InFieldData, pfe_desc->InFieldLength);
        int nYear, nMonth, nDay = 0;
        sscanf(szHireDate, "%04d-%02d-%02d", &nYear, &nMonth, &nDay);
        if (nYear < 1980)
        {
          nYear  = 1980;
          nMonth = 01;
          nDay   = 01;
        }
        sprintf(szHireDate, "%04d-%02d-%02d", nYear, nMonth, nDay);

        strncpy((char*)pfe_desc->OutFieldData, szHireDate, pfe_desc->InFieldLength);
        pfe_desc->OutFieldDataLength = pfe_desc->InFieldLength;

        if (pUser)
          pUser->nUpdates++;
        break;
      }
```

FE_Terminate

This optional function is called once at the end of the replication processing for which a Field Exit has been specified. In the example all actions performed in function 'FE_Exit_Before' are written into a log file and the file is closed. The number of actions performed is passed into the script function (passed as parameter) 'Exit_SetVariableInteger' and inserted into the script variable 'PM_O.EXITUPDATES'. Finally, the storage of the allocated structure USERDATASTRUCT is deallocated. In case of an error, the passed variable 'ErrorMsg' is filled with a message and the

function is terminated with a return value of not equal to 0 (zero). An error message can be used with a length of up to 1000 characters.

```
TCS_EXPORT(int) FE_Terminate(char *ErrorMsg, void **UserData, C_SCRIPT_FUNCTIONS *FunctionTable)
{
  if (NULL == UserData)
  {
    snprintf(ErrorMsg, 50, "Call without UserData set.");
    return (8);
  }
  else
  {
    PUSERDATASTRUCT pUser = (PUSERDATASTRUCT) *UserData;

    if (pUser->fLog)
    {
      write_message(pUser->fLog, "Field Exit terminated. Field data changed in exit: %d.\n", pUser->nUpdates);

      (*FunctionTable->Exit_SetVariableInteger)((char*) "PM_O.EXITUPDATES", pUser->nUpdates);

      fclose(pUser->fLog);
      pUser->fLog = NULL;
    };

    free(pUser);
  };
  return (0);
}
```

4.3 HP Exit

The system-wide load module with the name specified with parameter HP_Exit_Name corresponds to exits:

	Entry points in load module:
CDC_ExitUserRecord	HP_ExitGetLogRecord
CDC_ExitS1LogRecordRead	HP_ExitS1LogRecordRead
CDC_ExitS2LogRecordRead	HP_ExitS2LogRecordRead
CDC_ExitS3LogRecordRead	HP_ExitS3LogRecordRead

The exit is used for user-specific processing of data records.
If defined, the entry point 'HP_Initialize' is called at the beginning of the processing. Possible use is the allocation of storage.
If defined, the entry point 'HP_Terminate' is called at the end of the processing. Possible use is the deallocation of storage.
The module communicates with the calling script via parameter structures.

The HP exits will only be called if

- an HP exit load module has been defined with the parameter HP_Exit_Name and the module could be successfully loaded.
- the corresponding entry point has been defined in the load module.

If an HP exit has been defined in this way, a possibly existing script exit will be ignored according to chapter *3, Exit Procedures for Data Scripts*.

For the available example input tables 'DEFAULT.FILE.FROMEXIT', 'DEFAULT.VSAM.ARTIKEL' and 'DEFAULT.VSAM.ARTICLE', and output tables 'DEFAULT.DEMO_EXIT.DEMO.ARTIKEL' and 'DEFAULT.DEMO_EXIT.DEMO.ARTICLE' are used. The tables are part of the repository backup 'tcVISION-Repository-PartBackup-Demo-HP-Exit.tcVRCF'. In addition, file 'hp_input.s0' is required which contains the sample data. The processing scripts for the example are 'DEMO_HP_EXIT.TSF' and 'DEMO_HP2_EXIT.TSF' (for function HP_ExitGetLogRecord). The paths

used in the script for the S0 and output file must be adapted to meet your individual testing environment.

The structures and functions explained in the following text are extracts of the C sample file and are not meant to be complete.

4.3.1 Explanations to the Structures

The structures can be found in the header file 'tcsexit.h'.

USERDATASTRUCT

```
typedef struct __userdatastruct {
  FILE*              fLog;
  // counters
  unsigned long long ullRunTime;
  unsigned int       nLookups;
  unsigned int       nInserts;
  unsigned int       nUpdates;
  unsigned int       nDeletes;
  unsigned int       nTruncates;
  unsigned int       nErrors;
  unsigned int       nStarttransactions;
  unsigned int       nCommits;
  unsigned int       nRollbacks;
  char*              pszBuffer;
} USERDATASTRUCT, *PUSERDATASTRUCT;
```

This structure contains variables that can be available from the first call of the exit using 'HP_Initialize' and through all calls from 'HP_ExitSxLogRecordRead' and 'HP_ExitGetLogRecord' until the exit ends with 'HP_Terminate'. All important events that may occur during the exit execution can be logged. The structure is used for user data and can be modified to meet your individual requirements.

HP_GLR_DESC

```
typedef struct {
  short Version;               // Input      Version of this structure
  short StructureLength;       // Input      Length of this structure
  short InputFileNameLength;   // Input      Length of Input File Name
  short MaxKeyLength;          // Output     Max Length of key field
  short KeyLength;             // Output     Actual Length of key field
  short Filler1;
  int   MaxRecordLength;       // Output     Max Length of a Data Record
  int   RecordLength;          // Output     Actual Length of Data Record
  int   ReturnFlags;           // Output     Return Flags
  char* RecordAddress;         // Output     Pointer to Record Data
  char* KeyAddress;            // Output     Pointer to Key Data
  char* InputFileName;         // Input      Pointer to Input File Name
  char  StructureFileName[65]; // Output     Structure File Name
} HP_GLR_DESC, *PHP_GLR_DESC;
```

This structure is passed to function 'HP_ExitGetLogRecord'. It contains only little information. The intention is to gather the data for the new record.

HP_SX1_DESC

```
typedef struct {
  short Version;           // Input          Version of this structure
  short StructureLength;   // Input          Length of this structure
  short DataSourceType;    // Input          Type of datasource
  short DataType;          // Input          Type of recordset
  int   ReturnFlags;       // Output         Return Flags
  char  TimeStamp[26];     // Input          Timestamp
  char  UniqueId[13];      // Input          Unique id of recordset
  char  RecoveryToken[25]; // Input          Id of unit of recovery
  char  TableId[64];       // Input          Table Id of unit of recovery
  int   FunctionType;      // Input          Function
  int   OperationType;     // Input          Operation
```

```
  int   BI_Length;         // Input/Output    Length of BI Data
  void  *BI_Data;          // Input/Output    Pointer to BI data
  int   AI_Length;         // Input/Output    Length of AI Data
  void  *AI_Data;          // Input/Output    Pointer to AI data
  short DLI_CKL;           // Input           DLI length of the concatenated key
  char  DLI_DB[8];         // Input           DLI DB-name
  char  DLI_Segment[8];    // Input           DLI segment-name
  int   Out_Count;         // Output          Exit output count
  char  TimeStampRaw[8];   // Input           Timestamp in STCK format
  char  Userid[8];         // Input           DLI user ID
  char  SystemId[8];       // Input           DLI system ID
} HP_SX1_DESC, *PHP_SX1_DESC;
```

This structure is passed to function 'HP_ExitS1LogRecordRead'. It contains all information relevant to the data record that should be processed by the exit.

HP_SX2_DESC

```
typedef struct {
  short       Version;           // Input          Version of this structure
  short       StructureLength;   // Input          Length of this structure
  short       DataSourceType;    // Input          Type of datasource
  short       DataType;          // Input          Type of recordset
  int         ReturnFlags;       // Output         Return Flags
  char        TimeStamp[26];     // Input          Timestamp
  char        UniqueId[13];      // Input          Unique id of recordset
  char        RecoveryToken[25]; // Input          Id of unit of recovery
  char        TableId[64];       // Input          Table Id input side
  char        TableName[128];    // Input          Table name output side
  int         FunctionType;      // Input          Function
  int         OperationType;     // Input          Operation
  short       Fields_Count;      // Input          Field count
  short       Fields_Count_new;  // Output         New Field count after user Field/Data modification
  PHP_SX2_FLD Fields;            // Input/Output   Field list
} HP_SX2_DESC, *PHP_SX2_DESC;
```

This structure is passed to function 'HP_ExitS2LogRecordRead'. It contains all information relevant to the data record that should be processed by the exit.

HP_SX2_FLD

```
typedef struct SX2_FLD {
  struct SX2_FLD *next;               // Input          pointer to next field
  short          FieldType;           // Input/Output   SQL data type
  short          FieldScale;          // Input/Output   SQL scale
  int            FieldLength;         // Input/Output   SQL length
  short          FieldType_CSD;       // Input          SQL data type of input field
  short          FieldFlags;          // Input          Flags of field
#define FIELD_NOT_AVAILABLE   0x0001
  short          FieldCCSID;          // Input/Output   CCSID of field
  short          FieldKeySequenceNr;  // Input          1 to n for key fields, 0 otherwise
  short          Field_BI_Null;       // Input/Output   Field BI Null indicator
  short          Field_AI_Null;       // Input/Output   Field AI Null indicator
  int            Field_BI_DataLength; // Input/Output   Field BI value length
  void           *Field_BI_Data;      // Input/Output   Field BI value
  int            Field_AI_DataLength; // Input/Output   Field AI value length
  void           *Field_AI_Data;      // Input/Output   Field AI value
  short          FieldNameLength;     // Input          Field name length
  char           FieldName[0];        // Input          Field name
} HP_SX2_FLD, *PHP_SX2_FLD;
```

This structure is used by structure HP_SX2_DESC to describe the fields and make their contents available.

HP_SX3_DESC

```
typedef struct {
  short       Version;           // Input          Version of this structure
  short       StructureLength;   // Input          Length of this structure
  short       DataSourceType;    // Input          Type of datasource
  short       DataType;          // Input          Type of recordset
  int         CallFlag;          // Input          Call flag
  int         ReturnFlags;       // Output         Return Flags
  char        TimeStamp[26];     // Input          Timestamp
```

```
  char        UniqueId[13];      // Input           Unique id of recordset
  char        RecoveryToken[25]; // Input           Id of unit of recovery
  char        TargetName[128];   // Input           Output target name
  char        TableName[128];    // Input           Output table name
  int         FunctionType;      // Input           Function
  int         OperationType;     // Input           Operation
  char        *pComment;         // Input/Output    Comment
  int         CommentLength;     // Input/Output    Comment Length
  int         MaxCommentLength;  // Input           Maximum comment Length
  char        *pSQLDML;          // Input/Output    SQL DML
  int         SQLDMLLength;      // Input/Output    SQL DML Length
  int         MaxSQLDMLLength;   // Input           Maximum SQL DML Length
  short       SQLDMLCCSID;       // Input           SQL DML codepage id
  short       reserved;
  short       Parms_Count;       // Input           Parameter count
  short       Parms_Count_new;   // Output          New parameter count
  PHP_SX3_PRM Parms;             // Input/Output    Parameter list

} HP_SX3_DESC, *PHP_SX3_DESC;
```

This structure is passed to function 'HP_ExitS3LogRecordRead'. It contains all relevant information for this data record, the generated SQL command, and a list of all parameters for the SQL command if this command uses bound parameters.

HP_SX3_PRM

```
typedef struct SX3_PRM {
  struct SX3_PRM *next;                    // Input          pointer to next parms
  char           FieldName[128];           // Input/Output   Field name
  short          FieldType;                // Input/Output   SQL data type
  short          FieldScale;               // Input/Output   SQL scale
  short          FieldCCSID;               // Input/Output   CCSID of field
  short          FieldParamSequenceNr;     // Input/Output   sequence number of field
  short          FieldWhereParamSequenceNr; // Input/Output  sequence number of field in where clause
  short          Field_Null;               // Input/Output   Field Null indicator
  int            FieldLength;              // Input/Output   SQL length
  int            Field_DataLength;         // Input/Output   Field value length
  void           *Field_Data;              // Input/Output   Field value
} HP_SX3_PRM, *PHP_SX3_PRM;
```

This structure is used by structure HP_SX3_DESC to describe the fields and make their contents available.

HP_SEGTEST_DESC

```
typedef struct
{
  short          Version;                  // Input          Version of this structure
  short          StructureLength;          // Input          Length of this structure
  short          DataSourceType;           // Input          Type of data source
  short          DataType;                 // Input          Type of data
  int            ReturnFlags;              // Output         Return Flags
  char           TimeStamp[26];            // Input          Timestamp
  char           TableId[64];              // Input          Table Id input side
  char           TableName[128];           // Input          Table Id output side
  int            FunctionType;             // Input          Function
  int            OperationType;            // Input          Operation
  int            SegmentRecordsWritten;    // Input          Number of records written to the current segment
  long long      SegmentBytesWritten;      // Input          Number of bytes written to the current segment

} HP_SEGTEST_DESC, * PHP_SEGTEST_DESC;
```

This structure is passed to function 'HP_ExitSegmentationTest'. It contains all information relevant to this data record.

HP_LMNG_PRM

```
typedef struct {
  short Version;                          // Input       Version of this structure
  short StructureLength;                  // Input       Length of this structure
  short DataSourceType;                   // Input       Length of Input File Name
  short DataType;                         // Input       Length of Input File Name
  int   ReturnFlags;                      // Output      Return Flags
  int   Back_Update;                      // Output      back update Flag
  char  RecoveryToken[25];                // Input       Id of unit of recovery
  int   ActualProcessing;                 // Input       call time of this exit
#define  LMNG_DESC_LUW_CREATE             1
#define  LMNG_DESC_LUW_EXECUTE_COMMIT     2
#define  LMNG_DESC_LUW_EXECUTE_ROLLBACK   3
#define  LMNG_DESC_LUW_EXECUTE_UNKOWN     4
#define  LMNG_DESC_LUW_DELETE             5
#define  LMNG_DESC_LUW_ADD_IMPLICITE      6
  unsigned char StartTimeStamp[8];         // Input        start of the LUW
  unsigned char EndTimeStamp[8];           // Input        end of the LUW
  int      LUW_LogRecs;                    // Input
  int      LUW_LogBytes;                   // Input
  int      LUW_UndoRedoLogRecs;            // Input
  char     UserFilenameToken[64];          // Input/Output
  union                                    // Input
    {
        struct {
                  char            DB2_CorrelationId[12];
                  char            DB2_AuthorizationId[8];
                  char            DB2_ResourceName[8];
                  char            DB2_ConnectionType[8];
                  char            DB2_ConnectionId[8];
        } DB2M_DATA;
        struct {
                  char            DB2_AuthorizationId[8];
        } DB2L_DATA;
        struct {
                  char            DB2_AuthorizationId[8];
        } DB2V_DATA;
        struct {
                  char            IDMS_id[8];
                  char            IDMS_area[18];
                  char            IDMS_record[18];
                  char            IDMS_program[8];
                  char            IDMS_task1[4];
                  int             IDMS_task2;
                  int             IDMS_RUID;
                  unsigned short  IDMS_CVNO;
        } IDMS_DATA;
        struct {
                  char            ADABAS_Jobname[8];
                  char            ADABAS_Userid[8];
        } ADABAS_DATA;
        struct {
                  char            DLI_system[8];
                  char            DLI_command[4];
                  char            DLI_AuthorizationId[8];
                  char            DLI_userid[8];
                  char            DLI_db[8];
                  char            DLI_psb[8];
        } DLI_DATA;
        struct {
                  char            CICS_id[8];
                  char            CICS_userid[8];
                  char            CICS_transaction[4];
                  int             CICS_task;
        } CICS_VSAM_DATA;
        struct {
                  unsigned int    DATACOM_Run_Unit;
                  unsigned int    DATACOM_Tsn;
                  char            DATACOM_Job_Name[8];
                  char            DATACOM_User_Id[8];
                  char            DATACOM_Monitor_Id[12];
        } DATACOM_DATA;
        struct {
                  char            ORA_RBA[25];
                  char            ORA_SCNc[12];
                  int             ORA_UserId;
                  int             ORA_AuditSessionId;
                  char            ORA_Username[33];
                  short           ORA_Thread;
```

```
        } ORACLE_DATA;
        struct {
                  char            MSSQL_DatabaseName[128];
                  short           MSSQL_uid;
                  short           MSSQL_spid;
        } MSSQL_DATA;
    };

} HP_LMNG_DESC, *PHP_LMNG_DESC;
```

This structure is passed to function 'HP_ExitLUWManager'. The function contains all information relevant to this transaction (LUW).

4.3.2 Explanation to the Functions

The functions can be found in C-file 'hp_exit.cc', the helper (assisting) functions can be found in C++-file 'tcsutils.cc'.

The functions of the HP Exit are called by the script if an exit module with the name specified in the script can be found and contains the functions described in this chapter.

HP_Initialize

This optional function is called once at the start of the replication processing for which a HP Exit has been defined. In this example, storage is allocated for structure USERDATASTRUCT and assigned to variable 'UserData', hence this storage area is available to all further functions of the HP Exit. In addition, a log file is opened that can be used for trace entries. In case of an error, variable 'ErrorMsg' is filled with an error message and the function is terminated with a return value not equal to 0 (zero). The error message can be up to 1000 characters in length.

```
TCS_EXPORT(int) HP_Initialize(char *ErrorMsg, void **UserData, C_SCRIPT_FUNCTIONS *FunctionTable)
{
  if (NULL == UserData)
  {
    snprintf(ErrorMsg, 50, "Call without UserData set.");
    return (8);
  };

  PUSERDATASTRUCT pUser = (PUSERDATASTRUCT) malloc(sizeof(USERDATASTRUCT));

  if (pUser == NULL)
  {
    snprintf(ErrorMsg, 50, "Memory allocation error.");
    return (8);
  };

  memset(pUser, 0, sizeof(USERDATASTRUCT));
  pUser->pszBuffer = NULL;

  *UserData = pUser;

#ifdef VERBOSEHP_
  pUser->fLog = fopen("C:\\Temp\\hp_exit.log", "w");
  if (NULL == pUser->fLog)
  {
    snprintf(ErrorMsg, 50, "Unable to open logfile.");
    return (8);
  };
#else
  pUser->fLog = NULL;
#endif

  return (0);
}
```

HP_ExitGetLogRecord

This function is called if an exit module has been defined for the script with a data type of 'USEREXIT' (only available for data sources 'BULK_TRANSFER' and 'BATCH_COMPARE').
With this exit it is possible to create data records that are further processed according to the script definition (apply to a database, creation of a DML file, etc.).

The exit programmer must understand the data structure. In the following example the data type of individual fields must be known. The way a specific format is stored in the database (BIG-ENDIAN, LITTLE-ENDIAN, format of data types DATE, TIME, TIMESTAMP, DECIMAL, ZONED, etc.) must also be known.

In the example, three data records will be created and processed according to the replication path starting with input table 'DEFAULT.FILE.FROMEXIT', whereas 'DEFAULT' defines the script group, 'FILE' is the specification for a USEREXIT, and 'FROMEXIT' is the name of the selected input table.

```
struct s_article
{
  int   nNumber;
  char  szName[40];
  float fPrice;
  int   nSupplier;
  int   nCatalog;
  char  szUnit[25];
  int   nStock;
};

s_article articles[] =
{
  {201, "Vinegar",  23.45,  23, 67, "12 bottles",  321},
  {202, "Candy",     3.78,  24, 65, "3 rolls",    2365},
  {203, "Sausages", 31.99,  25, 68, "1 dozen",      24},
  {-1,  NULL,       -1,     -1, -1, NULL,           -1},
};
```

This is the definition of a structure and the sample data.

```
#define EXPECTED_GLR_PARAMETER_VERSION 1

TCS_EXPORT(int) HP_ExitGetLogRecord(char *ErrorMsg, void **UserData, C_SCRIPT_FUNCTIONS *FunctionTable, PHP_GLR_DESC
glr_desc)
{
  PUSERDATASTRUCT pUser;

  if (NULL == UserData)
  {
    snprintf(ErrorMsg, 50, "Call without UserData set.");
    return (8);
  }

  pUser = (PUSERDATASTRUCT) *UserData;

  if (NULL == pUser)
  {
    snprintf(ErrorMsg, 50, "Call without UserData initialized.");
    return (8);
  };

  if (glr_desc->Version != EXPECTED_GLR_PARAMETER_VERSION)
  {
    snprintf(ErrorMsg, 50, "Wrong version %d of parameter, %d expected.", sx3_desc->Version, EXPECTED_GLR_PARAMETER_VERSION);
    return (8);
  };

  static int  rec_count = 0;
  static char szData[200];

  if (!rec_count)
  {
    // do first initialization
    glr_desc->MaxKeyLength    =  11;
    glr_desc->MaxRecordLength = 114;
    strcpy ( glr_desc->StructureFileName, "FROMEXIT");
  }
```

```
  if (articles[rec_count].nNumber == -1)
  {
    glr_desc->ReturnFlags = 1; // EOF
  }
  else
  {
    memset(szData, ' ', sizeof(szData));
    sprintf ( szData,       "%011d", articles[rec_count].nNumber);
    memcpy  ( &szData[11],  articles[rec_count].szName, strlen(articles[rec_count].szName) );
    sprintf ( &szData[51],  "%010d", (int)(articles[rec_count].fPrice * 100) ); // ZONED(10, 2)
    sprintf ( &szData[61],  "%011d", articles[rec_count].nSupplier);
    sprintf ( &szData[72],  "%011d", articles[rec_count].nCatalog);
    memcpy  ( &szData[83],  articles[rec_count].szUnit, strlen(articles[rec_count].szUnit) );
    sprintf ( &szData[108], "%06d",  articles[rec_count].nStock);

    glr_desc->RecordAddress = szData;
    glr_desc->RecordLength  = 114;
    glr_desc->KeyAddress    = szData;
    glr_desc->KeyLength     = 11;

    glr_desc->ReturnFlags   = 0;

    rec_count++;
  }

  return (0);
}
```

In structure HP_GLR_DESC the function sets the variable for the maximum length of the data for the data record (here 114), the maximum length of the key (here 11) and the name of the input table (here 'FROMEXIT'). The static variable 'rec_count' saves the number of the currently created data record. As long as data is created, it is saved to a continuous data area and the corresponding variable is set. A value of 0 (zero) for variable 'ReturnFlags' indicates that the record should be processed, the exit function is then called again to create the next data record. A value of 1 for variable 'ReturnFlags' indicates that no more data is available and the exit function is no longer called.

HP_ExitS1LogRecordRead

This function is called for every data record that is processed. Different program sections of the exit can be executed depending on the function number and operation number. Usually this exit is used when processing different record types and a record field acts as an indicator which redefine of the data structure should be applied.

The exit programmer must understand the data structure. In the following example this means that the programmer knows that field number 1 of the replication is of data type ZONED(11). Also it is required to know how a specific format is stored in the database (BIG-ENDIAN, LITTLE-ENDIAN, format of data types DATE, TIME, TIMESTAMP, DECIMAL, ZONED, etc.).

In the example a data record with an even number (content of field 1, type ZONED(11) is processed without any change to table 'ARTIKEL' where a data record with an odd number is replicated to output table 'ARTICLE'.

```
#define EXPECTED_S1_PARAMETER_VERSION 1
TCS_EXPORT(int) HP_ExitS1LogRecordRead(char *ErrorMsg, void **UserData,
                                       C_SCRIPT_FUNCTIONS *FunctionTable, PHP_SX1_DESC sx1_desc)
{
  PUSERDATASTRUCT pUser;
  int nRC = 0;

  // just initialize
  sx1_desc->ReturnFlags = 0;

  if (NULL == UserData)
  {
    snprintf(ErrorMsg, 50, "Call without UserData set.");
    return (8);
  };
```

```
  pUser = (PUSERDATASTRUCT) *UserData;

  if (NULL == pUser)
  {
    snprintf(ErrorMsg, 50, "Call without UserData initialized.");
    return (8);
  };

  if (sx1_desc->Version != EXPECTED_S1_PARAMETER_VERSION)
  {
    snprintf(ErrorMsg, 50, "Wrong version %d of parameter, %d expected.", sx1_desc->Version, EXPECTED_S1_PARAMETER_VERSION);
    return (8);
  };

  if (sx1_desc->FunctionType == 7)
  {
    // UNDO-REDO record
    char szStatement[10];

    switch (sx1_desc->OperationType)
    {
      case 1:
        strcpy(szStatement, "INSERT");
        break;
      case 2:
        strcpy(szStatement, "UPDATE");
        break;
      case 3:
        strcpy(szStatement, "DELETE");
        break;
      case 4:
        strcpy(szStatement, "TRUNCATE");
        break;
      default:
        snprintf(ErrorMsg, 50, "unknown operation type (%d)", sx1_desc->OperationType);
        return (8);
    };

    if ( (sx1_desc->OperationType >= 1) && (sx1_desc->OperationType <= 3) )
    {
      // INSERT (1), UPDATE (2) or DELETE (3)
      if (sx1_desc->AI_Data && sx1_desc->AI_Length)
      {
        if ( ( (((unsigned char*)sx1_desc->AI_Data)[10] - 0xf0) % 2) == 1)
          strcpy(sx1_desc->TableId, "ARTICLE " );
      }

      if (sx1_desc->BI_Data && sx1_desc->BI_Length)
      {
        if ( ( (((unsigned char*)sx1_desc->BI_Data)[10] - 0xf0) % 2) == 1)
          strcpy(sx1_desc->TableId, "ARTICLE " );
      }
    };

#ifdef VERBOSE
    if (pUser->fLog)
      write_message(pUser->fLog, "Undo/Redo record (%s) at '% .26s' for '%s' ...\n",
                    szStatement, sx1_desc->TimeStamp, sx1_desc->TableId);
#endif
  }

  return 0;
}
```

HP_ExitS2LogRecordRead

This function is called for every data record that is processed. Different program sections of the exit can be executed depending on the function number and operation number.

The exit programmer must understand the structure of the underlying data. For the S2 exit the field data is already available in the format of the output table.

In this example, information about the processed fields are output and the contents of some fields of the data record are changed.

```
#define EXPECTED_S2_PARAMETER_VERSION 3
TCS_EXPORT(int) HP_ExitS2LogRecordRead(char *ErrorMsg, void **UserData,
                                       C_SCRIPT_FUNCTIONS *FunctionTable, PHP_SX2_DESC sx2_desc)
{
  PUSERDATASTRUCT pUser;
  int nRC = 0;

  // just initialize
  sx2_desc->ReturnFlags      = 0;
  sx2_desc->Fields_Count_new = sx2_desc->Fields_Count;

  if (NULL == UserData)
  {
    snprintf(ErrorMsg, 50, "Call without UserData set.");
    return (8);
  };

  pUser = (PUSERDATASTRUCT) *UserData;

  if (NULL == pUser)
  {
    snprintf(ErrorMsg, 50, "Call without UserData initialized.");
    return (8);
  };

  if (sx2_desc->Version != EXPECTED_S2_PARAMETER_VERSION)
  {
    snprintf(ErrorMsg, 50, "Wrong version %d of parameter, %d expected.", sx2_desc->Version, EXPECTED_S2_PARAMETER_VERSION);
    return (8);
  };

  if (sx2_desc->FunctionType == 7)
  {
    // UNDO-REDO record
    char szStatement[10];

    switch (sx2_desc->OperationType)
    {
      case 1:
        strcpy(szStatement, "INSERT");
        pUser->nInserts++;
        break;
      case 2:
        strcpy(szStatement, "UPDATE");
        pUser->nUpdates++;
        break;
      case 3:
        strcpy(szStatement, "DELETE");
        pUser->nDeletes++;
        break;
      case 4:
        strcpy(szStatement, "TRUNCATE");
        pUser->nTruncates++;
        break;
      default:
        snprintf(ErrorMsg, 50, "unknown operation type (%d)", sx2_desc->OperationType);
        return (8);
    };

#ifdef VERBOSE
    if (pUser->fLog)
      write_message(pUser->fLog, "Undo/Redo record (%s) at '% .26s' for '%s' with %d fields...\n",
                    szStatement, sx2_desc->TimeStamp, sx2_desc->TableId, sx2_desc->Fields_Count);
#endif

    if ( (sx2_desc->OperationType >= 1) && (sx2_desc->OperationType <= 3) )
    {
      // INSERT (1), UPDATE (2) or DELETE (3)

      int i;
      PHP_SX2_FLD pFld = sx2_desc->Fields;

      for (i = 0; i < sx2_desc->Fields_Count; i++)
      {
        short sFieldType;
        char  szFieldName[256];

        if (pFld == NULL)
        {
          snprintf(ErrorMsg, 50, "Non matching field count/fields.");
```

```
          return (8);
        };

#ifdef VERBOSE
        if (pUser->fLog)
        {
          write_message(pUser->fLog, "  FieldType:   %d (%d)\n", pFld->FieldType, pFld->FieldType_CSD);
          write_message(pUser->fLog, "  FieldScale:  %d\n",      pFld->FieldScale);
          write_message(pUser->fLog, "  FieldLength: %d\n",      pFld->FieldLength);
          write_message(pUser->fLog, "  KeySeqNr:    %d\n",      pFld->FieldKeySequenceNr);
          write_message(pUser->fLog, "  FieldCCSID:  %d\n",      pFld->FieldCCSID);
          write_message(pUser->fLog, "  FieldFlags:  %d\n",      pFld->FieldFlags);
          write_message(pUser->fLog, "  Field_BI_Data:\n");
          dump_hex_data(pUser->fLog, (unsigned char*) pFld->Field_BI_Data, pFld->Field_BI_DataLength, (char*)"    ");
          write_message(pUser->fLog, "  Field_AI_Data:\n");
          dump_hex_data(pUser->fLog, (unsigned char*) pFld->Field_AI_Data, pFld->Field_AI_DataLength, (char*)"    ");
        };
#endif
        if (pFld->FieldFlags & FIELD_NOT_AVAILABLE)
          break;

        memcpy(szFieldName, pFld->FieldName, min(pFld->FieldNameLength, sizeof(szFieldName) - 1));
        szFieldName[min(pFld->FieldNameLength, sizeof(szFieldName) - 1)] = '\0';

#ifdef VERBOSE
        if (pUser->fLog)
          write_message(pUser->fLog, "  FieldName:   %s\n\n", szFieldName);
#endif

        sFieldType = pFld->FieldType - (pFld->FieldType % 2);

        ...

        pFld = pFld->next;
      };
    };
  }

  return (0);
}
```

The field 'NUMBER' is defined as INTEGER for the output table and tcVISION stores integers in the BIG-ENDIAN format, hence the field value must be retrieved. Then the value is changed and written back into the data buffer (according to the BIG-ENDIAN format).

```
        // change the value of the NUMBER to Open Server NUMBERs
        if (!strcmp(szFieldName, "NUMBER"))
        {
          if (pFld->Field_AI_Data && pFld->Field_AI_DataLength)
          {
            int nNumber = ( ((unsigned char*)pFld->Field_AI_Data)[0] << 24)
                        + ( ((unsigned char*)pFld->Field_AI_Data)[1] << 16)
                        + ( ((unsigned char*)pFld->Field_AI_Data)[2] <<  8)
                        + ( ((unsigned char*)pFld->Field_AI_Data)[3]);

            nNumber += 1000;

            ((char*)pFld->Field_AI_Data)[0] = (nNumber & 0xff000000) >> 24;
            ((char*)pFld->Field_AI_Data)[1] = (nNumber & 0x00ff0000) >> 16;
            ((char*)pFld->Field_AI_Data)[2] = (nNumber & 0x0000ff00) >>  8;
            ((char*)pFld->Field_AI_Data)[3] = (nNumber & 0x000000ff);
          }

          if (pFld->Field_BI_Data && pFld->Field_BI_DataLength)
          {
            int nNumber = ( ((unsigned char*)pFld->Field_BI_Data)[0] << 24)
                        + ( ((unsigned char*)pFld->Field_BI_Data)[1] << 16)
                        + ( ((unsigned char*)pFld->Field_BI_Data)[2] <<  8)
                        + ( ((unsigned char*)pFld->Field_BI_Data)[3]);

            nNumber += 1000;

            ((char*)pFld->Field_BI_Data)[0] = (nNumber & 0xff000000) >> 24;
            ((char*)pFld->Field_BI_Data)[1] = (nNumber & 0x00ff0000) >> 16;
            ((char*)pFld->Field_BI_Data)[2] = (nNumber & 0x0000ff00) >>  8;
            ((char*)pFld->Field_BI_Data)[3] = (nNumber & 0x000000ff);
          }
        }
```

The field 'NAME' is of data type CHARACTER and can simply be overwritten.

```
  // translate the value of the NAME
  if (!strcmp(szFieldName, "NAME"))
  {
    if (pFld->Field_AI_Data && pFld->Field_AI_DataLength)
    {
      if (!strcmp((char*)pFld->Field_AI_Data, "Zucker                                  "))
        strcpy((char*)pFld->Field_AI_Data, "Sugar                                 ");
      else if (!strcmp((char*)pFld->Field_AI_Data, "Salz                                      "))
        strcpy((char*)pFld->Field_AI_Data, "Salt                                  ");
    }

    if (pFld->Field_BI_Data && pFld->Field_BI_DataLength)
    {
      if (!strcmp((char*)pFld->Field_BI_Data, "Zucker                                  "))
        strcpy((char*)pFld->Field_BI_Data, "Sugar                                 ");
      else if (!strcmp((char*)pFld->Field_BI_Data, "Salz                                      "))
        strcpy((char*)pFld->Field_BI_Data, "Salt                                  ");
    }
  }
```

The field 'PRICE' is of data type DECIMAL. tcVISION stores this in a packed format. Therefore, the field content is first converted into a string (assist function 'print_packed_decimal') and converted into a number using the standard C-function 'atof'. Then the number is changed and converted back into a string using standard C-function 'sprintf'. Finally, the assisting function 'DisplayTo_S390_Packed' converts back to the correct format. To do this, the length of the number (number of decimal digits) and the number of decimal places is required.

```
  // change the value of the PRICE
  if (!strcmp(szFieldName, "PRICE"))
  {
    if (pFld->Field_AI_Data && pFld->Field_AI_DataLength)
    {
      unsigned char szPrice[32] = {0};
      print_packed_decimal((unsigned char*)pFld->Field_AI_Data, pFld->FieldLength, pFld->FieldScale, szPrice);
      double dbPrice = atof((char*)szPrice);
      dbPrice *= 2;
      sprintf((char*)szPrice, "%f", dbPrice);
      DisplayTo_S390_Packed ((char*)szPrice, pFld->FieldLength, pFld->FieldScale,
                             (unsigned char*)pFld->Field_AI_Data, '.');
    }

    if (pFld->Field_BI_Data && pFld->Field_BI_DataLength)
    {
      unsigned char szPrice[32] = {0};
      print_packed_decimal((unsigned char*)pFld->Field_BI_Data, pFld->FieldLength, pFld->FieldScale, szPrice);
      double dbPrice = atof((char*)szPrice);
      dbPrice *= 2;
      sprintf((char*)szPrice, "%f", dbPrice);
      DisplayTo_S390_Packed ((char*)szPrice, pFld->FieldLength, pFld->FieldScale,
                             (unsigned char*)pFld->Field_BI_Data, '.');
    }
  }
```

HP_ExitS3LogRecordRead

This function is called for every processed data record. Different sections of the exit can be executed depending on the function number, operation number, and CallFlag.
The following example passes the first call for a data record (with variable 'CallFlag' = 0), but variable 'ReturnFlag' is set to 2 so that the exit for the same data record is called again (this time with variable 'CallFlag' = 1). This pass uses the sample exit to write the SQL command of the data record into a log file.

```
#define EXPECTED_S3_PARAMETER_VERSION 2

TCS_EXPORT(int) HP_ExitS3LogRecordRead(char *ErrorMsg, void **UserData,
                                       C_SCRIPT_FUNCTIONS *FunctionTable, PHP_SX3_DESC sx3_desc)
{
  PUSERDATASTRUCT pUser;

  sx3_desc->ReturnFlags = 0; // ok, thats all

  if (NULL == UserData)
```

```
{
  snprintf(ErrorMsg, 50, "Call without UserData set.");
  return (8);
}

pUser = (PUSERDATASTRUCT) *UserData;

if (NULL == pUser)
{
  snprintf(ErrorMsg, 50, "Call without UserData initialized.");
  return (8);
};

if (sx3_desc->Version != EXPECTED_S3_PARAMETER_VERSION)
{
  snprintf(ErrorMsg, 50, "Wrong version %d of parameter, %d expected.", sx3_desc->Version, EXPECTED_S3_PARAMETER_VERSION);
  return (8);
};

if (sx3_desc->FunctionType == 7)
{
  if (sx3_desc->CallFlag == 0)
  {
    // first call with original SQL - we write and pass
    write_message(pUser->fLog, "First call: '%s'.\nPass through and call again.\n", sx3_desc->pSQLDML);
    sx3_desc->ReturnFlags = 2;
  }
  else if (sx3_desc->CallFlag == 1)
  {
    // follow up call
    PHP_SX3_PRM s3fp = sx3_desc->Parms;

    // insert into protocol table
    if (pUser->pszBuffer == NULL)
      pUser->pszBuffer = (char*)malloc(sx3_desc->MaxSQLDMLLength);

    if (pUser->pszBuffer)
    {
      char szProtocol[256] = {0};

      strcpy(pUser->pszBuffer, sx3_desc->pSQLDML);

      strcpy(szProtocol, "insert into DEMO.PROTOCOL (TIMESTMP, DML) values (?, ?)");

      sx3_desc->Parms_Count_new = 2;

      if (!s3fp)
      {
        sx3_desc->Parms = (PHP_SX3_PRM)malloc(sizeof(HP_SX3_PRM));
        memset(sx3_desc->Parms, 0, sizeof(HP_SX3_PRM));
        s3fp = sx3_desc->Parms;
      }

      strcpy(s3fp->FieldName, "TIMESTMP");
      s3fp->FieldType                 = SQL_TIMESTP_392;
      s3fp->FieldLength               = 26;
      s3fp->FieldScale                = 0;
      s3fp->Field_Null                = 0;
      s3fp->Field_DataLength          = 26;
      s3fp->Field_Data                = sx3_desc->TimeStamp;
      s3fp->FieldParamSequenceNr      = 1;
      s3fp->FieldWhereParamSequenceNr = 0;

      if (!s3fp->next)
      {
        s3fp->next = (PHP_SX3_PRM)malloc(sizeof(HP_SX3_PRM));
        memset(s3fp->next, 0, sizeof(HP_SX3_PRM));
      }

      s3fp = s3fp->next;

      strcpy(s3fp->FieldName, "DML");
      s3fp->FieldType                 = SQL_CHAR_452;
      s3fp->FieldLength               = 400;
      s3fp->FieldScale                = 0;
      s3fp->Field_Null                = 0;
      s3fp->Field_DataLength          = min(400, strlen(pUser->pszBuffer));
      s3fp->Field_Data                = pUser->pszBuffer;
      s3fp->FieldParamSequenceNr      = 2;
      s3fp->FieldWhereParamSequenceNr = 0;
```

```
        strcpy(sx3_desc->pSQLDML, szProtocol);
        sx3_desc->SQLDMLLength = strlen(szProtocol);

        sx3_desc->ReturnFlags = 0;
      }
    }
  }

  return (0);
}
```

The insert statement for the log file is created, as well as the bound parameters for the timestamp and the original statement. The variable for the bound parameter 'Parms_Count_new' is set (here 2). Setting of variable 'ReturnFlags' to 0 (zero) takes over the data record and the exit is not called again for this data record.

HP_ExitSegmentationTest

This function is called for every data record processed during a BULK load with a file as output target. The function can be used to take a decision about the segmentation of output files based on logical relationships.

The exit programmer must understand the structure of the underlying data.
Value 0 for variable 'ReturnFlags' indicates that the data record is transferred into the current file, a value of 1 closes the current file, and the data record is written into the new file. Please note that the naming convention for the output file(s) is prepared by using the relevant tokens (e.g. <counter>) for multiple names.

```
#define EXPECTED_SGMT_PARAMETER_VERSION 1

TCS_EXPORT(int) HP_ExitSegmentationTest(char *ErrorMsg, void **UserData, C_SCRIPT_FUNCTIONS *FunctionTable, PHP_SEGTEST_DESC
sgt_desc)
{

  if (sgt_desc->Version != EXPECTED_SGMT_PARAMETER_VERSION)
  {
    snprintf(ErrorMsg, 50, "Wrong version %d of parameter, %d expected.", sgt_desc->Version,
EXPECTED_SGMT_PARAMETER_VERSION);
    return (8);
  };

  sgt_desc->ReturnFlags = 0;
  if (sgt_desc->FunctionType == 7)
  {
    static int ctr = 0;
    if (strcmp(sgt_desc->TableId, "DI21PART.PARTROOT") == 0 &&
        ++ctr >= 5000)
    {
      sgt_desc->ReturnFlags = 1;
      ctr = 0;
    }
  }

  return (0);
}
```

HP_ExitLUWManager

This function is called while transactions are processed. Different sections of the exit can be executed depending on the function number.

The following example examines the CorrelationId of a z/OS Db2 transaction and sets - depending on a selection list - the value for the user-specific file name token. Therefore, the processing of different CorrelationIds takes place in different paths.

```
#define EXPECTED_LMNG_PARAMETER_VERSION 1
```

```
TCS_EXPORT(int) HP_ExitLUWManager(char *ErrorMsg, void **UserData,
                                  C_SCRIPT_FUNCTIONS *FunctionTable, PHP_LMNG_DESC lmng_desc)
{
  PUSERDATASTRUCT pUser;

  lmng_desc->ReturnFlags = 0; // all ok

  if (NULL == UserData)
  {
    snprintf(ErrorMsg, 50, "Call without UserData set.");
    return (8);
  }


  pUser = (PUSERDATASTRUCT) *UserData;

  if (NULL == pUser)
  {
    snprintf(ErrorMsg, 50, "Call without UserData initialized.");
    return (8);
  };

  if (lmng_desc->DataSourceType == 2 &&                                    // is this a DB2_LOGREC transaction?
      lmng_desc->ActualProcessing == LMNG_DESC_LUW_EXECUTE_COMMIT &&       // about to execute and commit transaction?
      lmng_desc->LUW_UndoRedoLogRecs > 0)                                  // some data in this tranasaction
  {
    char * ptr;
    char DB2_CorrelationId[13];
    if (NULL == pUser->pszBuffer)
    {
      // get and prepare the selection list for the correlations ids
      char  *Buffer;
      int   BufLen;
      int   rc = (FunctionTable->Exit_GetVariableBuffer) ("CorrelationIdList", (void **) &Buffer, &BufLen);
      if (!BufLen)
        return 0;

      pUser->pszBuffer = (char*)malloc(BufLen+2);
      memcpy(pUser->pszBuffer, Buffer, BufLen);
      pUser->pszBuffer[BufLen] = 0;
      pUser->pszBuffer[BufLen+1] = 0;

      ptr = strtok(pUser->pszBuffer, ",");
      while (ptr)
        ptr = strtok(0, ",");
      }

      // prepare the given correlation id for proper string compare
      memset(DB2_CorrelationId, 0, sizeof(DB2_CorrelationId));
      strncpy(DB2_CorrelationId, lmng_desc->DB2M_DATA.DB2_CorrelationId,
          sizeof(lmng_desc->DB2M_DATA.DB2_CorrelationId));
      if (ptr = strchr(DB2_CorrelationId, ' '))
        *ptr = 0;

      // check out, if the correlation id is in our list
      ptr = pUser->pszBuffer;
      while (*ptr)
      {
        if (stricmp(ptr, DB2_CorrelationId) == 0)
        {
          strcpy(lmng_desc->UserFilenameToken, "Special");
          return(0);
        }
        ptr += strlen(ptr) +1;
      }

      strcpy(lmng_desc->UserFilenameToken, "Normal");
    }

    return(0);
}
```

HP_Terminate

This optional function is called once at the end of the replication processing for which an HP Exit has been specified. In the example all actions performed in function 'HP_ExitSxLogRecordRead' are written into a log file and the file is closed. The number of actions performed is passed to the script function (passed as parameter) 'Exit_SetVariableInteger' and inserted into the script variable 'PM_O.EXIT...'. Finally, the storage of the allocated structure USERDATASTRUCT is deallocated. In case of an error, the passed variable 'ErrorMsg' is filled with a message and the function is terminated with a return value of not equal to 0 (zero). An error message can be used with a length of up to 1000 characters.

```
TCS_EXPORT(int) HP_Terminate(char *ErrorMsg, void **UserData, C_SCRIPT_FUNCTIONS *FunctionTable)
{
  if (NULL == UserData)
  {
    snprintf(ErrorMsg, 50, "Call without UserData set.");
    return (8);
  }
  else
  {
    PUSERDATASTRUCT pUser = (PUSERDATASTRUCT) *UserData;

#ifdef VERBOSE
    if (pUser->fLog)
    {
      write_message(pUser->fLog, "\nActions performed by exit:\n");
      write_message(pUser->fLog, "Inserts:           %d\n",   pUser->nInserts);
      write_message(pUser->fLog, "Updates:           %d\n",   pUser->nUpdates);
      write_message(pUser->fLog, "Deletes:           %d\n",   pUser->nDeletes);
      write_message(pUser->fLog, "Truncates:         %d\n",   pUser->nTruncates);
    }
#endif

    (*FunctionTable->Exit_SetVariableInteger)((char*) "PM_O.EXITINSERTS",    pUser->nInserts);
    (*FunctionTable->Exit_SetVariableInteger)((char*) "PM_O.EXITUPDATES",    pUser->nUpdates);
    (*FunctionTable->Exit_SetVariableInteger)((char*) "PM_O.EXITDELETES",    pUser->nDeletes);
    (*FunctionTable->Exit_SetVariableInteger)((char*) "PM_O.EXITTRUNCATES",  pUser->nTruncates);
    (*FunctionTable->Exit_SetVariableInteger)((char*) "PM_O.EXITERRORS",     pUser->nErrors);

    if (pUser->fLog)
    {
      fclose(pUser->fLog);
      pUser->fLog = NULL;
    };

    if (pUser->pszBuffer)
    {
      free(pUser->pszBuffer);
      pUser->pszBuffer = NULL;
    };

    free(pUser);
  };

  return (0);
}
```

4.4 HP Replication Process Exit

The load module with the name defined in the replication linkage of the repository corresponds to the exit:

CDC_ExitReplicationProcess	Entry point in load module: RP_Process

The exit is used for the user-specific processing of data record fields.

If the exit point 'RP_Initialize' has been defined, this point is called at the beginning of the processing and can - for example - be used to allocate storage.
If the exit point 'RP_Terminate' has been defined, this point is called at the end of the processing and can be used, for example, to deallocate storage.
The module communicates with the calling script via parameter structures.

RP-Exits are only called if

1. an RP-Exit load module has been specified in the replication linkage of the repository and the exit could be successfully loaded.
2. the corresponding entry point has been defined in the load module.

If an RP-Exit has been activated with this method, a specified script exit as described in chapter Exit Procedures for Data Scripts will be skipped.

For the available example the input table 'DEFAULT.DB2/LUW.SAMPLE.DB2INST1.EMPLO_RP' and the output table 'DEFAULT.DEMO_EXIT.DB2INST1.EMPLO_RP' are used. The tables are part of the repository backup 'tcVISION-Repository-PartBackup-Demo-RP-Exit.tcVRCF'. In addition, the file 'rp_input.s0' is required which contains the sample data. The processing scripts for the example are 'DEMO_RP_EXIT.TSF'. The paths used in the script for the S0 and output file must be adapted to meet your individual testing environment.

The structures and functions explained in the following text are extracts of the C sample file and are not meant to be complete.

4.4.1 Explanation to the Structures

The described structures can be found in the header file 'tcsexit.h'.

USERDATASTRUCT

```
typedef struct __userdatastruct {
  FILE*              fLog;
  // counters
  unsigned long long ullRunTime;
  unsigned int       nLookups;
  unsigned int       nInserts;
  unsigned int       nUpdates;
  unsigned int       nDeletes;
  unsigned int       nTruncates;
  unsigned int       nErrors;
  unsigned int       nStarttransactions;
  unsigned int       nCommits;
  unsigned int       nRollbacks;
  char*              pszBuffer;
} USERDATASTRUCT, *PUSERDATASTRUCT;
```

This structure contains a collection of variables that can be available from the first call of the exit using 'RP_Initialize' and through all calls from 'RP_Process' until the exit ends with 'RP_Terminate'. All important events that may occur during the exit execution can be logged. The structure is used for user data and can be modified to meet your individual requirements.

RP_DESC

```
typedef struct SRP_DESC {
  short         Version;           // Input          Version of this structure
  short         StructureLength;   // Input          Length of this structure
  short         CallNumber;        // Input          From replication process definition
  short         DataSourceType;    // Input          Type of data source
  short         DataType;          // Input          Type of recordset
  short         FunctionType;      // Input          Function
  short         OperationType;     // Input          Operation
  char          TimeStamp[26];     // Input          Timestamp
  char          UniqueId[13];      // Input          Unique id of recordset
```

```
  char          RecoveryToken[25]; // Input          Id of unit of recovery
  char          InputType[50];     // Input          Type of input object
  char          InputTable[128];   // Input          Name of input object
  char          OutputTarget[128]; // Input          Name of output target
  char          OutputTable[128];  // Input          Name of output table
  unsigned char ImageType;         // Input          Type of image data (B=before/A=after)
  void          *pImageData;       // Input          Pointer to input image data
  int           ImageLength;       // Input          Length of input image data
  void          *pNewData;         // Input/Output   Pointer to output data
  short         Fields_Count;      // Input          Field count
  PRP_FLD       ActField;          // Input          Pointer to current field
  PRP_FLD       Fields;            // Input          Pointer to field list

} RP_DESC, *PRP_DESC;
```

This structure is passed to the function 'RP_Process'. It contains all information relevant to the data record, the field that should be processed by the exit because of the definition of the replication linkage (processing), and a list of all fields of the data record.

RP_FLD

```
typedef struct SRP_FLD {
  struct SRP_FLD *pnext;       // Input          pointer to next field
  short  FieldType;            // Input/Output   SQL data type
  short  FieldScale;           // Input/Output   SQL scale
  int    FieldLength;          // Input/Output   SQL length
  short  FieldCCSID;           // Input/Output   CCSID of field
  short  FieldKeySequenceNr;   // Input          1 to n for key fields, 0 otherwise
  short  FieldTable;           // Input          Field table number
  short  FieldLevel;           // Input          Field table depth
  short  FieldIndex[5];        // Input          Field table occurrence
  short  Field_Null;           // Input/Output   Field Null indicator
  int    Field_DataLength;     // Input/Output   Field value length
  void   *Field_Data;          // Input/Output   Field value
  short  FieldNameLength;      // Input          Field name length
  char   FieldName[0];         // Input          Field name
} RP_FLD, *PRP_FLD;
```

This structure is used by the structure SRP_DESC to describe the fields and make their contents available.

4.4.2 Explanations to the Functions

The functions can be found in C-file 'rp_exit.cc', the helper (assisting) functions can be found in C-file 'tcsutils.cc'.

The functions of the RP exit are called by the script if an exit module with a name specified in the replication linkage (processing) has been found which contains the functions described in this chapter.

RP_Initialize

This optional function is called once at the start of the replication processing for which a Replication Process Exit exit has been defined. In this example, storage is allocated for the structure USERDATASTRUCT and assigned to variable 'UserData', hence this storage area is available to all further functions of the Replication Process Exit. In addition, a log file that can be used for trace entries is opened. In case of an error variable, 'ErrorMsg' is filled with an error message and the function is terminated with a return value not equal to 0 (zero). The error message can be up to 1000 characters in length.

```
TCS_EXPORT(int) RP_Initialize(char *ErrorMsg, void **UserData, C_SCRIPT_FUNCTIONS *FunctionTable)
{
  if (NULL == UserData)
  {
    snprintf(ErrorMsg, 50, "Call without UserData set.");
```

```
    return (8);
  };

  PUSERDATASTRUCT pUser = (PUSERDATASTRUCT) malloc(sizeof(USERDATASTRUCT));

  if (pUser == NULL)
  {
    snprintf(ErrorMsg, 50, "Memory allocation error.");
    return (8);
  };

  memset(pUser, 0, sizeof(USERDATASTRUCT));
  *UserData = pUser;

#ifdef VERBOSE
  pUser->fLog = fopen("C:\\Temp\\rp_exit.log", "w");
  if (NULL == pUser->fLog)
  {
    snprintf(ErrorMsg, 50, "Unable to open logfile.");
    return (8);
  };
#else
  pUser->fLog = NULL;
#endif

  return (0);
}
```

RP_Process

This function is called for every field of the data record whose replication linkage (processing) contains the exit name and a function number (variable 'CallNumber' of structure RP_DESC. Different sections of the exit that process data to create or maintain a field can be executed depending on the function number . If the function is called with an unknown function number, the variable 'ErrorMsg' is filled with an error message and the function is terminated with a return value not equal to 0 (zero). The error message can be up to 1000 characters in length.

With this exit it is possible to create data records that are further processed according to the script definition (apply to a database, creation of a DML file, etc.).

The exit programmer must understand the data structure. In the following example the data type of individual fields must be known. For the RP exit the field data is already in the format of the output table. If values should be read (i.e. INTEGER or DECIMAL values), the same rules as for the function HP_ExitS2LogRecordRead of the HP exit apply.

In the example, a text value is copied into a field of data type CHARACTER.

```
TCS_EXPORT(int) RP_Process(char *ErrorMsg, void **UserData, C_SCRIPT_FUNCTIONS *FunctionTable, PRP_DESC prp_desc)
{
  static int ctr1, ctr2, ctr3, ctr4, ctr5, ctr6, ctr7, ctr8, ctr9 = 0;
  PUSERDATASTRUCT pUser = (PUSERDATASTRUCT) *UserData;

  switch (prp_desc->CallNumber)
  {
    case 2:
    {
      // creating character data
      prp_desc->ActField->FieldType        = SQL_CHAR_452;
      prp_desc->ActField->FieldLength      = 50;
      prp_desc->ActField->Field_Null       = 0;
      prp_desc->ActField->Field_Data       = prp_desc->pNewData;
      prp_desc->ActField->Field_DataLength = sprintf( (char*)prp_desc->pNewData,
                                                      "Processing No. %d done for call number %d",
                                                      ++ctr1, prp_desc->CallNumber);

      if (pUser)
        pUser->nUpdates++;

      break;
    }

    . . .
```

```
    default:
      sprintf(ErrorMsg, "Unknown call number %d in RP_Process", prp_desc->CallNumber);
      return(99);
  }

  return(0);
}
```

It is also possible to copy numbers into the field data. The number must be copied as a character string and the field type must be adapted accordingly (SQL_CHAR_452). The actual conversion of the character string value into the format specified in the repository (here SQL_SMALLINT_500 for a 2-byte small integer) is performed by tcVISION.

```
    case 5:
    {
      // creating short integer data
      short sValue = (short)(++ctr2 * prp_desc->CallNumber);

      prp_desc->ActField->FieldType        = SQL_CHAR_452;
      prp_desc->ActField->FieldLength      = 6;
      prp_desc->ActField->Field_Null       = 0;
      prp_desc->ActField->Field_Data       = prp_desc->pNewData;
      prp_desc->ActField->Field_DataLength = sprintf((char*)prp_desc->pNewData, "%05d", sValue);

      if (pUser)
        pUser->nUpdates++;

      break;
    }
```

The same applies to a 4-byte integer (SQL_INTEGER_496)

```
      // creating integer data
      . . .
      prp_desc->ActField->FieldType        = SQL_CHAR_452;
      prp_desc->ActField->FieldLength      = 11;
      prp_desc->ActField->Field_Null       = 0;
      prp_desc->ActField->Field_Data       = prp_desc->pNewData;
      prp_desc->ActField->Field_DataLength = sprintf((char*)prp_desc->pNewData, "%010d", nValue);
      . . .
```

and an 8-byte big integer (SQL_BIGINTEGER_492).

```
      // creating big integer data
      . . .
      prp_desc->ActField->FieldType        = SQL_CHAR_452;
      prp_desc->ActField->FieldLength      = 20;
      prp_desc->ActField->Field_Null       = 0;
      prp_desc->ActField->Field_Data       = prp_desc->pNewData;
      prp_desc->ActField->Field_DataLength = sprintf((char*)prp_desc->pNewData, "%020lld", lValue);
      . . .
```

Floating point numbers can also be used as field data. The same applies: The number is copied as a character string and the field type must be specified accordingly (SQL_CHAR_452). The actual conversion of the character string value into the format specified in the repository (here SQL_FLOAT_480) is performed by tcVISION.

```
    case 8:
    {
      // creating float data
      float fValue = (float)12.34 * (++ctr5 * prp_desc->CallNumber);

      prp_desc->ActField->FieldType        = SQL_CHAR_452;
      prp_desc->ActField->FieldLength      = 25;
      prp_desc->ActField->Field_Null       = 0;
      prp_desc->ActField->Field_Data       = prp_desc->pNewData;
      prp_desc->ActField->Field_DataLength = sprintf((char*)prp_desc->pNewData, "%f", fValue);

      if (pUser)
        pUser->nUpdates++;

      break;
    }
```

The creation of decimal numbers follows the same procedure. The decimal number is copied as a character string and the field type must be specified accordingly (SQL_CHAR_452). The actual conversion of the character string value into the format specified in the repository (here SQL_DECIMAL_484) is performed by tcVISION.

```
case 9:
{
  // creating decimal data
  double dValue = 23.45 * (++ctr6 * prp_desc->CallNumber);

  prp_desc->ActField->FieldType        = SQL_CHAR_452;
  prp_desc->ActField->FieldLength      = prp_desc->ActField->FieldLength + 2;
  prp_desc->ActField->Field_Null       = 0;
  prp_desc->ActField->Field_Data       = prp_desc->pNewData;
  prp_desc->ActField->Field_DataLength = sprintf((char*)prp_desc->pNewData, "%f", dValue);

  if (pUser)
    pUser->nUpdates++;

  break;
}
```

Even data types like DATE, TIME, and TIMESTAMP are set as character strings. The character strings follow the standard conventions.

Data type	Format	Length
DATE	YYYY-MM-DD	10
TIME	hh.mm.ss	8
TIMESTAMP	YYYY-MM-DD-hh.mm.ss.ffffff	26

```
case 10:
{
  // creating time data
  prp_desc->ActField->FieldType   = SQL_TIME_388;
  prp_desc->ActField->FieldLength = 8;
  prp_desc->ActField->FieldScale  = 0;
  prp_desc->ActField->Field_Null  = 0;
  prp_desc->ActField->Field_Data  = prp_desc->pNewData;

  int nHour   = (ctr7 * 3) % 24;
  int nMinute = (ctr7 * 4) % 60;
  int nSecond = (ctr7 * 5) % 60;

  ctr7 += 1;

  prp_desc->ActField->Field_DataLength = sprintf((char*)prp_desc->pNewData, "%02d.%02d.%02d", nHour, nMinute, nSecond);

  if (pUser)
    pUser->nUpdates++;

  break;
}
```

It is generally possible to use the same coding for data types DATE and TIMESTAMP. The following example discusses type DATE, because the data must be retrieved first from another field. The example searches the passed field list until the field 'HIREDATE' has been found. This field is of type DATE, the field value is retrieved and stored in variables. These variables are then modified and the newly created date is returned as the new field value.

```
case 11:
{
  // creating date data
  prp_desc->ActField->FieldType   = SQL_DATE_384;
  prp_desc->ActField->FieldLength = 10;
  prp_desc->ActField->FieldScale  = 0;
  prp_desc->ActField->Field_Null  = 0;
  prp_desc->ActField->Field_Data  = prp_desc->pNewData;
```

```
      PRP_FLD pFld = prp_desc->Fields;
      char szFieldName[256];
      int nField = 0;

      int nYear  = 1900;
      int nMonth = 1;
      int nDay   = 1;

      for (nField = 0 ; nField < prp_desc->Fields_Count ; nField++)
      {
        memcpy(szFieldName, pFld->FieldName, min(pFld->FieldNameLength, sizeof(szFieldName) - 1));
        szFieldName[min(pFld->FieldNameLength, sizeof(szFieldName) - 1)] = '\0';

        if (!strcmp(szFieldName, "HIREDATE"))
        {
          sscanf((const char*)pFld->Field_Data, "%04d-%02d-%02d", &nYear, &nMonth, &nDay);
          nYear  += 10;
          nMonth += 5;
          nDay   += 10;
          if (nMonth > 12)
            nMonth = 12;
          if (nDay > 28)
            nDay = 28;
          break;
        }

        pFld = pFld->pnext;
      }

      prp_desc->ActField->Field_DataLength = sprintf((char*)prp_desc->pNewData, "%04d-%02d-%02d", nYear, nMonth, nDay);

      if (pUser)
        pUser->nUpdates++;

      break;
    }
```

RP_Describe

This function is called by the tcVISION Dashboard application if the replication exit has been selected for a replication linkage and the button 'obtain' is used. The exit function returns a text to the Dashboard application that displays a list showing the number and a brief explanation of the available exit functions. The length of the text can be up to 5000 characters. The text format is shown in the following example.

```
#define RP_EXIT_DESCRIPTION "2 = Character data;5 = Short integer data;6 = Integer data;7 = Big integer data;8 = Float data;9
= Decimal data;10 = Time data;11 = Date data;12 = Timestamp data"

TCS_EXPORT(int) RP_Describe(char *Description, int cbChars)
{
  if ( cbChars < (int)strlen ( RP_EXIT_DESCRIPTION))
    return (0);
  else
    strcpy ( Description,  RP_EXIT_DESCRIPTION);

  return ( ( int) strlen ( RP_EXIT_DESCRIPTION));
}
```

The text defined and returned in the example above corresponds to the available exit functions of the previously discussed function RP_Process.

RP_Terminate

This optional function is called once at the end of the replication processing for which a Replication Process Exit has been specified. In the example, all actions performed in function 'RP_Process' are written into a log file and the file is closed. The number of actions performed is passed into the script function (passed as parameter) 'Exit_SetVariableInteger' and inserted into the script variable 'PM_O.EXITUPDATES'. Finally, the storage of the allocated structure USERDATASTRUCT is deallocated. In case of an error, the passed variable 'ErrorMsg' is filled with a message and the function is terminated with a return value of not equal to 0 (zero). An error message can be used with a length of up to 1000 characters.

```
TCS_EXPORT(int) RP_Terminate(char *ErrorMsg, void **UserData, C_SCRIPT_FUNCTIONS *FunctionTable)
{
  if (NULL == UserData)
  {
    snprintf(ErrorMsg, 50, "Call without UserData set.");
    return (8);
  }
  else
  {
    PUSERDATASTRUCT pUser = (PUSERDATASTRUCT) *UserData;

    if (pUser->fLog)
    {
      write_message(pUser->fLog, "RP Exit terminated. Field data created in exit: %d.\n", pUser->nUpdates);

      (*FunctionTable->Exit_SetVariableInteger)((char*) "PM_O.EXITUPDATES", pUser->nUpdates);

      fclose(pUser->fLog);
      pUser->fLog = NULL;
    };

    free(pUser);
  };

  return (0);
}
```

5 Parameters for Control Scripts

This chapter covers parameters and functions of control scripts. These are scripts with Function = 'CONTROL'.

5.1 Function

The parameter Function defines the script as control script.
The parameter value is a character string with the value 'CONTROL'.

Example: `Function = 'CONTROL'`

This parameter is mandatory and must be specified.

6 Exit Procedures for Control Scripts

This chapter covers the predefined exit procedures of control scripts.

Exit calls are activated by inserting the procedures with the predefined procedure name underneath the function call

```
CDC_RC = CALL_CDC()
```

Input parameters are available to all exit procedures that can be evaluated for processing and output parameters to control further processing.
The parameter names themselves consist of a stem and a name separated by a period (.).

Example: `X_CS_I.ErrorMessage`

These procedures must be terminated at their logical end with a RETURN instruction.
Using the RETURN instruction the procedure returns information about possible error situations in the internal processing to the calling function. The RETURN parameter consists of a return code and an optional error text that is separated by a comma.

RETURN Code	0	No error in the procedure, continue processing
	<>0	Error, termination of processing. A possible error text will become part of the error message of the control script.

Examples:

```
RETURN '12, Error in CDC_ExitControl: xxxxx'
RETURN '0'
```

The following chapter describes the predefined exit procedures and their corresponding input and output parameters.

6.1 CDC_ExitControl

The exit procedure CDC_ExitControl is always called to perform the processing steps as defined by the user.

Parameter Stem:

Input:	none
Output:	none

Input Parameter:

none

Output parameter:

none

6.1.1 Start of an IMS Batch Script in a z/OS Batch Region

```
/*-------------------------------------------------------------
  z/OS Batch Job Submit, JCL is in STEM Job.
-------------------------------------------------------------*/
CDC_ExitControl:
```

```
/* Read in the JCL for our job */
File = 'TCVISION.TCSCRIPT.CNTLX(JOBIMS)'
Call Stream  File, 'C', 'OPEN READ'
IF Result <> 'READY:' THEN DO
      SAY result
      call Stream File, 'D'
      SAY result
      RETURN '12, Unable to access File('File')' Result
END

i = 0
DO FOREVER
      InLine = LINEIN(File)
      if LENGTH(InLine) = 0 THEN LEAVE
      i      = i + 1
      Job.i  = InLine
END
Call Stream File, 'C', 'CLOSE'

/* Append the dynamic parameters */
i = i + 1
Job.i = '//STDPARM  DD *'
i = i + 1
Job.i = 'TCAVDI:SYSDSK:/SCRIPTS/IMS_Batch.tsf'
i = i + 1
Job.i = Input_Parms 'EXTFLG=1'
Job.0 = i

/* Submit job to internal reader */
File = 'SYSOUT=A;WRITER=INTRDR;LRECL=80'
Call Stream  File, 'C', 'OPEN WRITE REPLACE'
IF Result <> 'READY:' THEN DO
      SAY result
      call Stream File, 'D'
      SAY result
      RETURN '12, Unable to access Internal Reader' result
END

DO i=1 to Job.0
      Call Lineout File, LEFT(Job.i, 72)
END
Call Stream File, 'C', 'CLOSE'

RETURN '0, No Error'
```

6.1.2Start of an Oracle Loader Script

```
/*-------------------------------------------------------------
  Call ORACLE Loader
-------------------------------------------------------------*/
CDC_ExitControl:

ORA_Logfile  = 'Z:\tcSCRIPT\log.txt '
ORA_Datafile = 'Z:\tcSCRIPT\indata.txt '

/* Get dynamic input file name */
FPOS = INDEX(Input_Parms, 'INPUT_SOURCE_NAME=')
IF FPOS > 0 THEN DO
  ORA_Datafile = SUBSTR(Input_Parms, FPOS + LENGTH('INPUT_SOURCE_NAME='))
```

```
  FPOS = INDEX(ORA_Datafile, ',')
  IF FPOS > 0 THEN DO
    ORA_Datafile = LEFT(ORA_Datafile, FPOS - 1)
  END
  ORA_Datafile = STRIP(ORA_Datafile)
END

/* Prepare ORACLE Loader Command line Parameters */
Parms =       ' SCOTT/788445'                            /* Userid */
Parms = Parms ' CONTROL=Z:\tcSCRIPT\incontrol.ctl'      /* Control file */
Parms = Parms ' DATA='ORA_Datafile                      /* Data file */
Parms = Parms ' LOG='ORA_Logfile                        /* Log file */
/* Additional options if required
Parms = Parms 'DIRECT=TRUE '
Parms = Parms 'PARALLEL=TRUE '
*/

/* Call the Loader */
CALL sqlldr Parms
SAY RESULT

CALL STREAM ORA_Logfile, 'C', 'OPEN READ'
IF RESULT <> 'READY:' THEN DO
      CALL STREAM ORA_Logfile, 'D'
      RETURN '12, Unable to access File('ORA_Logfile')' RESULT
END

/* Check log file for error messages */
ORA_Message = ''
ORA_Error = ''
SIGNAL ON NOTREADY
DO FOREVER
      InLine = LINEIN(ORA_Logfile)
      IF LEFT(InLine, 11) = 'SQL*Loader-' THEN ORA_Error = ORA_Error InLine
      IF INDEX(InLine, 'successfully loaded') > 0 THEN ORA_Message = ORA_Message
InLine
END
NOTREADY:
CALL STREAM ORA_Logfile, 'C', 'CLOSE'

SAY 'MSG' ORA_Message
SAY 'ERR' ORA_Error
IF LENGTH(ORA_Error) = 0 THEN DO
      RETURN '0, ORA Msg:' ORA_Message
END
RETURN '12,' ORA_Error
```

For a more detailed description refer to the document "Using the Oracle Loader".

7 Performance Monitoring

Performance monitoring can be activated using Input_Flags = INPUT_FLAGS_PM_ACTIVE.
During or after a script execution the following variables can be evaluated:

PM_O.PM_Start_TS	TS: Script begin
PM_O.PM_StartWork_TS	TS: Begin of processing after initialization
PM_O.PM_End_TS	TS: End of processing
PM_O.PM_Duration	T: Duration of processing
PM_O.PM_Initialization	T: Duration of initialization
PM_O.PM_Waiting_Source	T: Waiting for data source
PM_O.PM_Reading_Source	T: Reading data source
PM_O.PM_Writing_Target	T: Writing output
PM_O.PM_Reading_Structures	T: Reading structure information
PM_O.PM_Reading_Dictionaries	T: Reading Db2 z/OS 'compression dictionaries'
PM_O.PM_CDC_Exit	T: Processing in CDC Exits
PM_O.PM_Last_NearRealTime_Gap	T: Last time difference between log record timestamp and current UTC
PM_O.PM_Last_NearRealTime_Gap_TS	TS: time when PM_Last_NearRealTime_Gap has been set
PM_O.PM_Max_NearRealTime_Gap	T: Maximum time difference between log record TS and current UTC
PM_O.PM_Max_NearRealTime_Gap_TS	TS: time when PM_Max_NearRealTime_Gap has been set

TS: Variable contains a timestamp
T: Variable contains the total elapsed time as well as a percentage relative to PM_O.PM_Duration

All timings are in UTC.

Example for a script epilog:

```
IF PM_O.PM_Start_TS <> 'PM_O.PM_Start_TS'  THEN DO
      SAY 'PM_Start_TS:                ' PM_O.PM_Start_TS
      SAY 'PM_StartWork_TS:            ' PM_O.PM_StartWork_TS
      SAY 'PM_End_TS:                  ' PM_O.PM_End_TS
      SAY 'PM_Duration:                ' PM_O.PM_Duration
      SAY 'PM_Initialization:          ' PM_O.PM_Initialization
      SAY 'PM_Waiting_Source:          ' PM_O.PM_Waiting_Source
      SAY 'PM_Reading_Source:          ' PM_O.PM_Reading_Source
      SAY 'PM_Writing_Target:          ' PM_O.PM_Writing_Target
      SAY 'PM_Reading_Structures:      ' PM_O.PM_Reading_Structures
      SAY 'PM_Reading_Dictionaries:    ' PM_O.PM_Reading_Dictionaries
      SAY 'PM_CDC_Exit:                ' PM_O.PM_CDC_Exit
      SAY 'PM_Last_NearRealTime_Gap:   ' PM_O.PM_Last_NearRealTime_Gap
      SAY 'PM_Last_NearRealTime_Gap_TS:' PM_O.PM_Last_NearRealTime_Gap_TS
      SAY 'PM_Max_NearRealTime_Gap:    ' PM_O.PM_Max_NearRealTime_Gap
      SAY 'PM_Max_NearRealTime_Gap_TS: ' PM_O.PM_Max_NearRealTime_Gap_TS
END
```

8 Processing Examples

This chapter covers some examples of special tcVISION tasks.

8.1 Adabas PLOG Processing for Bulk Data Changes

Adabas provides several utilities that can change the data of a database during production processing and DO NOT maintain the log (PLOG). This is done to get a better performance.
To ensure that the databases in the target system will stay in sync with the changed source data in Adabas, these utility functions must be reproduced in the target system. Two cases must be differentiated:

- Deletion of all records in a table (ADADBS REFRESH)
- Bulk insertion of data into a table (ADALOD UPDATE)

For the first case the DBID and file number from the corresponding entry in PLOG can be directly used to generate an SQL command 'DELETE FROM table'.

In the second case the utility ADALOD uses an input file (DDEBAND) that contains the data to be inserted. The file is in the special ADACMP compressed format.
This input file is also the base for tcVISION to perform a bulk data import for the corresponding SQL loader (or SQL INSERT commands) on the target system. The file can be used directly as input for an Adabas bulk transfer. For organizational reasons it can be considered to copy the file to the target system so that the same input data will be used.
FTP or a tcVISION data script can be used for the copy process.
Before the start of the ADALOD UPDATE, a C5 user entry must be written into the PLOG. This is necessary to save the required parameters for the subsequent ADALOD UPDATE (e.g. the name of the input file, etc.). Later this information will be read by a tcVISION data script when processing the PLOG data and it will be matched with the corresponding entry of the ADALOD UPDATE. The exit CDC_ExitDBControl can be called with the corresponding parameter values and the exit can start a script that inserts the data in the correct logical sequence into the target system using SQL loader or SQL commands.
After this script has been processed successfully, the calling script can continue the PLOG processing.

8.1.1 Examples for the Adabas UPDATE Job

An example for an Adabas UPDATE job may look like the following:

```
//ADALODU   JOB ,'ADABAS UPDATE',
//          CLASS=A,MSGCLASS=S
//*
//*   tcVISION PLOG Hook
//*   We write an Adabas User CP-Record
//*
//STEP1    EXEC PGM=TCSCRIPT,PARM='SYSSCRT',REGION=0M
//STEPLIB  DD DISP=SHR,DSN= TCVISION.LOADLIB
//STDENV   DD DISP=SHR,DSN=TCVISION.TCSCRIPT.CNTL(ENVFAST)
//SYSSCRT  DD *
ADA.Command = 'C5'
ADA.RB =        'TCVISION ADALOD INFO'
ADA.RB = ADA.RB 'DATE='DATE(S)
```

```
ADA.RB = ADA.RB 'TIME='TIME(N)
ADA.RB = ADA.RB 'JOBNAME='JOBNAME()
ADA.RB = ADA.RB 'JOBID='JOBID()
ADA.RB = ADA.RB 'FUNCTION=UPDATE'
ADA.RB = ADA.RB 'DBID=1'
ADA.RB = ADA.RB 'FILE=1'
ADA.RB = ADA.RB 'DDEBAND=ADABAS.EMPL.BIN'

RC = CALLADA('','ADA', 'DQ1')
SAY 'ADABAS RC('RC') ADABAS RC('DQ1.ReturnCode')'
IF RC = 0 THEN RETURN 0
RETURN 12
//*
//*   Copy the data file to the remote system
//*
//STEP2    EXEC PGM=FTP,REGION=4096K
//OUTPUT   DD SYSOUT=*
//INPUT    DD *
192.168.0.148
UserID
Password
binary
put  'ADABAS.EMPL'  ADABAS.EMPL.BIN
quit
/*
//*
//*   ADALOD: LOAD AN ADAM FILE
//*
//STEP3     EXEC PGM=ADARUN,REGION=0M
//STEPLIB   DD   DISP=SHR,DSN=ADABAS.LOAD.CENOBIT
//*
//DDASSOR1  DD   DISP=SHR,DSN=ADABAS.DATA.DB001.ASSOR1
//DDDATAR1  DD   DISP=SHR,DSN=ADABAS.DATA.DB001.DATAR1
//DDTEMPR1  DD   DISP=OLD,DSN=ADABAS.DATA.DB001.TEMPR1
//DDSORTR1  DD   DISP=OLD,DSN=ADABAS.DATA.DB001.SORTR1
//DDEBAND   DD   DISP=OLD,DSN=ADABAS.EMPL
//DDDRUCK   DD   SYSOUT=*
//DDPRINT   DD   SYSOUT=*
//SYSUDUMP  DD   SYSOUT=*
//DDCARD    DD   *
ADARUN PROG=ADALOD,MODE=MULTI,SVC=249,DEVICE=3390,DBID=001
/*
//DDKARTE   DD   *
ADALOD UPDATE FILE=1,LWP=400K
ADALOD TEMPSIZE=50,SORTSIZE=50
/*
```

Step1 makes the entry in Adabas PLOG that uses keywords to contain the parameters for the subsequent ADALOG UPDATE. The entry must start with the keyword TCVISION so that it can be recognized as a user entry for tcVISION.

Step2 copies the input file of the ADALOD UPDATE to a remote system using FTP. The name used on the remote system is ADABAS.EMPL.BIN.
Instead of FTP a tcVISION data script with the function bulk transfer can be used to store the data locally or remote.

Step3 performs the normal Adabas UPDATE.

8.1.2 Example for an Adabas PLOG Data Script

The base is a data script that analyses Adabas PLOG data as input. During the processing it will find a C5 user entry and will save the entry internally. In one of the following PLOG records it will find an ADALOD UPDATE entry with the same job name, DBID, and file number. If defined, the exit CDC_ExitDBControl will be called with this information.
The full file name of the ADALOD input file can be built and a data script that processes this file can be started:

```
CDC_ExitDBControl:
IF X_DBC_I.CallType = 1 THEN DO
      say X_DBC_I.CallType
      say X_DBC_I.ADABAS_DBNR
      say X_DBC_I.ADABAS_FINR
      say X_DBC_I.ADABAS_JOB
      say X_DBC_I.ADABAS_C5

      File_name = ''
      rc = INDEX(X_DBC_I.ADABAS_C5, 'DDEBAND=')
      IF rc > 0 THEN DO
            File_name = WORD(SUBSTR(X_DBC_I.ADABAS_C5, rc + 8), 1)
      END
      File_name = 'c:\temp\' File_name

      X_DBC_O.ReturnFlags = 1
      X_DBC_O.ScriptName  = 'ADALOD.tsf'
      X_DBC_O.ScriptParms = 'Input_Source_Name=' File_name
END

IF X_DBC_I.CallType = 2 THEN DO
      say 'delete'
      say X_DBC_I.CallType
      say X_DBC_I.ADABAS_DBNR
      say X_DBC_I.ADABAS_FINR
      say X_DBC_I.ADABAS_JOB

      X_DBC_O.ReturnFlags = 0
END

RETURN '0,No Error'
```

In this exit script ADALOD.TSF will be prepared with a start parameter of `Input_Source_Name=`. Parameter value is the directory `C:\temp\` plus the file name from the C5 entry. Because of `X_DBC_O.ReturnFlags = 1`, the calling script starts the corresponding subscript and waits for the termination of that script. If the script terminates with no error the PLOG processing continues.

8.2 Adabas Real-Time PLOG Processing

During processing and if configured accordingly, Adabas writes all changes applied to the user data to the PLOG files. At any time, a PLOG file from the set of defined PLOG files (2 to 8 files) is the active PLOG file.

A data script with the complete SET of all PLOG files processes the PLOG files in the correct sequence until processing has completed the last block written by the Adabas nucleus to the active PLOG file.
If flag INPUT_FLAGS_NRT_CONTINUE_EOF has been set, the script waits for the next entry in the active PLOG file or a PLOG switch and continues processing.
A data script automatically recognizes a PLOG switch. The same applies to the START or STOP of the Adabas nucleus.
A processing restart of the log reading script with archived PLCOPY files and a subsequent change to the set of active PLOG files is also possible using the same restart file.
In order to read the active PLOG files correctly, a tcVISION ADABAS Heartbeat File is used, see parameter ADA_PLOG_HEARTBEAT=n.
An ADABAS Userid also has to be specified with the parameter ADABAS_USERID.

8.2.1 Adabas Spanned Records

Adabas 'spanned records' in PLOG files are supported starting with rev. 1730+. If Adabas files with 'spanned records' selected for tcVISION have updates in secondary records, it has to be ensured - with the help of the ADARUN parameter SRLOG=PART/ALL - that all required records are written in the PLOG.

8.2.2 Adabas LUW

If the configuration has been made accordingly, Adabas LUW logs the changes applied to user data to the PLOG files. These files are written continuously to the specified directory. These PLOG files may consist of one or multiple files per session (PLOG-NR). For example after 'adaopr db=xx noet_sync feof=plog'. The tcVISION data script processes these files directly from this directory.
Adabas LUW Option NOBI is not supported because the user data cannot be reconstructed in this case.

8.2.3 Adabas Parallel/Cluster Services

Adabas Parallel/Cluster Services PLOG data is supported.
The parameter ADA_INTERNAL_NUCID is used to selectively process the log data of the different nuclei.
The default scenario consists of a sender script which directly reads the corresponding PLOG files and sends the log data in stage 0 to a receiver script.
The receiver script writes the log data in stage 1 with LUW manager actively into a tcVISION pipe per nucleus.
One or more Apply scripts can read the LUWs from these pipes in the correct sequence and process them further. For this purpose, all pipe names must be specified concatenated with the '&' operator in the Input_Source_Name parameter.

Active PLOG files:
For sender scripts that process active PLOG files, the ADA_INTERNAL_NUCID parameter must be used to specify the Internal Nucleus ID of the corresponding nucleus.

Merged PLCOPY files:
For scripts that process merged PLOG files, the Internal Nucleus ID of the corresponding nucleus can be specified to selectively process only the log data of that one nucleus.
This type of processing can be used to selectively reprocess log data from the corresponding merged PLOGs for the corresponding nucleus after failure of the active PLOG reader. So the active PLOG reader can subsequently be restarted.

If only processing of the merged PLOGs is desired and no subsequent switch to the active PLOGs is intended, the value 0 (or 1) can also be specified for this parameter. In this case the log data for all nuclei will be processed.

Switch from merged PLCOPY files to PLOG files:
After all merged PLCOPY files per nucleus (using ADA_INTERNAL_NUCID = n) have been processed, the scripts for the active PLOG files can be started. These scripts search for the correct start position in the active PLOG per nucleus and start processing from this position.

Switch from active PLOG files to merged PLCOPY files:
This change is usually not necessary, unless a script for active PLOG files (e.g. after an interruption, error, etc.) can no longer start again at the last position in the active PLOG, as this has already been overwritten with new log data by the active nucleus.
In this case, the corresponding merged PLOG files must be specified in the sender script.
So after startup the receiver script uses the restart information in the merged PLOG files to set up at the correct position and read out the missing log data.

8.2.4 Scenario for Adabas Real-Time PLOG and Remote Scripts

A typical scenario is a sending script that sends the data in stage 0 format with optional preselection to a remote receiving script.
In this scenario the starting point in the active PLOGs is always defined by the receiving script either through a restart file (State_Save_File ...) of a previous processing or using the parameter ADA_FromPLOG or Input_From_Timestamp. Without these start values the processing starts based on the oldest log entry in the active PLOGs.
The sending script does not use the parameter ADA_FromPLOG or Input_From_Timestamp, and is not supposed to use a restart file (State_Save_File ...), because it is controlled as a server from the receiving script.

8.2.5 Parameter for Adabas Real-Time PLOG

The following parameters are used:

NRTLR_MinWaitInterval = n.
n is a value in milliseconds and specifies the ‘age‘ of an Adabas PLOG block from which the data should be processed to ensure that all preceding PLOG blocks are written to the disk. The default value is 600.
The parameter should be set to e.g. 2000 if the system is overloaded or in case of asynchronous I/O.
The value n also specifies in which time interval the script checks for new PLOG data at the end of the active PLOG.
If there is no new data, this time interval is increased by the value n until the maximum value m of the parameter NRTLR_MaxWaitInterval is reached.

NRTLR_MaxWaitInterval = m.
m is a value in milliseconds that specifies the maximal time interval in which the script checks for new PLOG data at the end of an active PLOG. The default value is 1500.

For Adabas Parallel Services the following additional parameter is used:
DS_MEMBER_TIMEOUT_CHECK = n.
n is a value in seconds that specifies the time difference between the current system time and the last timestamp of a member of the cluster from which the processing is cancelled by the apply script due to a possible disruption in the respective PLOG LogReader. The default value is 10.

This value can be increased if the system times are not synchronous.

NRTLR_ADAPLOGMaxBackInterval = n.
n is a value in milliseconds with default value 2000. The processing assumes that there is no PLOG data with a timestamp older than the current system time minus the value n to be processed for the respective PLOG Reader. If this happens nonetheless, the processing will be cancelled with the respective error message and the value n should be increased accordingly.

8.3 Adabas ADASAV Processing

Adabas allows a backup with the ADASAV utility while the system is running.
These ADASAV files can subsequently be used as input for a tcVISION bulk process, just like ADAULD files.
With the ADASAV online, however, there is the possibility of user UPDATEs to the data to be backed up while the ADASAV utility is active. In this case the ADASAV utility writes a corresponding note in the job log:

```
The following files have been modified while ADASAV was running:

I Files                                                                  I
I------------------------------------------------------------------------I
I     013                                                                I
I------------------------------------------------------------------------I
```

For this case additional entries were written in the PLOG reflecting these user UPDATEs.
The tcVISION ADABAS CDC process recognizes these additional entries and allows by means of the CDC_ExitDBControl the preparation of these additional entries into a separate file as an ADASAV PLOG extract, which has to be used again in the subsequent bulk processing of the ADASAV file.
For ADABAS files with LOB fields the corresponding LOB file is automatically processed as well.
Passing on the ADASAV data to another tcVISION bulk process (e.g. send-receive) is only possible in stage 0 because only in this format the corresponding PLOG and LOB data can be passed on.

8.3.1 Write ADASAV-PLOG Extract

During tcVISION ADABAS CDC processing in stage 1+, the process detects the entries for ADASAV online checkpoints (SYN1, SYN2, SYN4, and SYN5) and associated ADASAV DATA entries in the PLOG.
When these CPs are reached, the CDC_ExitDBControl is called with CallType 55.

Exit example:

```
CDC_ExitDBControl:

X_DBC_O.ReturnFlags = 0

/* ADASAV CT=55 */
IF X_DBC_I.CallType = 55 THEN DO
      SAY 'ADASAV: 'X_DBC_I.ADABAS_INFO
      SAY ' JOB   ='X_DBC_I.ADABAS_JOB
      SAY ' DBNR  ='X_DBC_I.ADABAS_DBNR
      SAY ' PLOG  ='X_DBC_I.ADABAS_PLOG
      SAY ' BLOCK ='X_DBC_I.ADABAS_BLOCK
```

```
      SAY ' OFFSET='X_DBC_I.ADABAS_OFFSET
      SAY ' TS    ='X_DBC_I.ADABAS_TS
      SAY '***'
      X_DBC_O.ReturnFlags = 8
      X_DBC_O.ADASAV_PATH = 'C:\tmp\ADASAV'
END

RETURN '0,No Error'
```

with the corresponding output on the console:

```
ADASAV: SYN4 online file(s), start of operation
 JOB   =ADASAV10
 DBNR  =10
 PLOG  =1791
 BLOCK =20713
 OFFSET=3413
 TS    =2021-03-22-14.45.26.290183
***
ADASAV: SYN5 online file(s), end of operation
 JOB   =ADASAV10
 DBNR  =10
 PLOG  =1791
 BLOCK =20719
 OFFSET=7894
 TS    =2021-03-22-14.45.26.528351
***
```

Using `X_DBC_O.ReturnFlags = 8` the writing of a PLOG extract with the ADASAV DATA entries for this ADASAV instance is activated.
Optionally `X_DBC_O.ADASAV_PATH = 'C:\tmp\ADASAV'` can be used to specify a target directory for the extract, e.g. the directory to which the ADASAV files will be copied later.
Without specifying X_DBC_O.ADASAV_PATH = the extract will be written to the tcVISION work directory.

The naming convention for the extract is as follows:
ADASAV_DBID_*dbid*_PLOG_*plog*_BLOCK_*block*
e.g. ADASAV_DBID_10_PLOG_1791_BLOCK_20713

The values for *dbid*, *plog,* and *block* correspond to the SYS1/4 PLOG entries or the joblog of the ADASAV utility:

```
I        I Protection Log  PLOGNUM=1791, SYN4=20713, SYN5=20719       I   I
```

Additionally, an index file with the following name is written when the SYN2/5 CP is reached:
ADASAV_DBID_*dbid*_PLOG_*plog*_BLOCK_*block.row.tcvhdb.*

This completes the writing of the ADASAV-PLOG extract and it is now available for a subsequent tcVISION bulk processing of the ADASAV file.
The ADASAV-PLOG extract is used by the script that directly reads the ADASAV file.
If the ADASAV PLOG extract is not available or cannot be found, the ADASAV data blocks are taken over unchanged - without any possible update from the PLOG due to updates during the duration of the ADASAV job.

8.3.2 ADASAV-PLOG_Extract without CDC processing

If no tcVISION ADABAS CDC processing has taken place for the PLOG in question, a second file can be specified to the bulk process in addition to the ADASAV input file, which contains the archived PLOG for the corresponding ADASAV.
The bulk process will then also provide an ADASAV PLOG extract in a first step as described above.

8.3.3 Read ADASAV-PLOG Extract

During the bulk of an ADASAV file the tcVISION process automatically checks whether a corresponding ADASAV PLOG extract is available - first in the directory of the ADASAV file and if necessary in the tcVISION work directory.
If this is the case, it is ensured that with the help of the ADASAV-PLOG extract that the state of the data from the ADASAV file corresponds to the state of the data in the ADABAS-DB at the time of the SYN2/5-CP.

8.4 Db2 z/OS Data Sharing

A structure import always addresses the data sharing group, meaning that there are no member names in the database name of the input table. Therefore, the replication continues even if the active Db2 member is changed (e.g. for load distribution or for ensuring fail safety).

Two main aspects should be noted for Db2 data sharing:

- An image copy is always made from a specified member, so the member name is found in the image copy itself. The parameter

  ```
  ImageCopySubSystem='databasename'
  ```

  has to be set, so that tcVISION can assign the respective metadata in the repository.

During a direct bulk transfer using a DRDA connection, a connection to a member is made. So the member name is displayed instead of the group name. It has to be overwritten in the connection string

```
DB2NAME='datasharinggroupname'
```

so that tcVISION can assign the respective metadata in the repository.

8.5 DB2 z/OS Image Copy Processing

tcVISION can process Db2 image copy files directly.
Compressed data with the algorithms 'Fixed Length', 'Huffman' and zlib for LOB's can be processed.
The following Db2 options are important when creating an image copy file:

SYSTEMPAGES YES
Must be left at default YES to store important meta information in the image copy.

FULL YES
Must be left on default YES to get complete data.

SHRLEVEL NONE or REFERENCE
SHRLEVEL NONE or REFERENCE ensures that no uncommitted data is written to the image copy file and that the CDC process can start with the start RBA of the image copy.
SHRLEVEL CHANGE can also be used, but may process uncommitted as well as duplicate records resulting in a subsequent CDC process with duplicate INSERT errors.

FLASHCOPY NO YES
FLASHCOPY YES can be used. But these image copy files can only be processed on the host of tcVISION and forwarded e.g. in stage 0.

8.6 Db2 z/OS Log Processing

The following requirements must be met to process data from Db2 z/OS:

- All affected tables must have the 'DATA CAPTURE CHANGES' attribute in order to process all INSERT, DELETE, and UPDATE statements.
- The tablespaces should not be defined as 'NOT LOGGED', otherwise Db2 does not write log entries for objects in this tablespace.

```
ALTER TABLE schema.tblname DATA CAPTURE CHANGES
```

Both log types archive and active log are supported.

To detect ALTER TABLE .. ALTER COLUMN DDL commands, the table SYSIBM.SYSCOLUMNS must have the 'DATA CAPTURE CHANGES' attribute.
ALTER TABLE .. ALTER COLUMN DDL commands can then be treated just like, for example, ALTER TABLE ADD COLUMN DDL commands in exit CDC_ExitDBControl.

Archive Log:
Archive log files can be directly processed with parameter Input_Source_Type = 'DB2M_FILE' and the file name supplied with parameter Input_Source_Name = ...
This type of processing can be performed on different platforms when the log files have been transferred (e.g. using FTP) to the target system.

With the parameter Input_Source_Type = 'DSNJSLR' log files can be processed on the z/OS system using the Db2 DSNJSLR interface. In this case, the parameter Input_Source_Name = ... specifies the Db2 bootstrap dataset. This file is used internally to locate the correct archive log files for the specified RBAs/LRSNs. The specification of the Db2 bootstrap file in the parameter Input_Source_Name can be performed in two different ways:
(1) Specify the Db2 bootstrap file using the following syntax: `//DDN:BSDS`
The DD name BSDS is mandatory and must be defined in the tcVISION Agent startup job with a DD statement.

Example: `//BSDS DD DSN= DSN610.DB1G.BSDS01,DISP=SHR`
This allocates the BDSD file through the JCL (recommended method).
(2) It is also possible to specify the DSN name of the Db2 bootstrap file using parameter Input_Source_Name. In this case the specified BSDS file is dynamically allocated.

Active Log:

Active logs can be directly processed with parameter Input_Source_Type = 'IFI_306'. The Db2 IFI interface is used to process the active log. The parameter Input_Source_Name = ... contains the Db2 connect string information. The corresponding Db2 system or the member of a data sharing group must be started to process log files in real time. However, the script is capable to detect the start and stop of the Db2 system. It can pause or reactivate the processing according to the Db2 system state.

8.6.1 Parameters for Db2 z/OS Active Log Processing

PM_I.NRTLR_MinWaitInterval = n.
n is a value in milliseconds with default 100 and defines in which time interval the script checks at the end of the active log, if new log data are available.
If no new data is available, this time interval is increased by the value n up to the maximum value m of the parameter PM_I.NRTLR_MaxWaitInterval.

PM_I.NRTLR_MaxWaitInterval = m.
m is a value in milliseconds with default 500 and defines the maximum time interval after which the script checks for new log data at the end of the active log.

8.6.2 Scenario for Db2 z/OS Active Log Processing

With `Input_Source_Type = 'IFI_306'` and `INPUT_FLAGS_NRT_CONTINUE_EOF` the data script processes all data in the active log and waits until new data arrives in the active log. This data is processed.
The scripts must not be terminated when the Db2 z/OS is stopped or started.

8.6.3 Scenario for Db2 z/OS Remote Script Processing

A typical scenario is a SEND script that selects the data and sends it in stage 0 format to a remote RECEIVE script.

In this scenario the start RBA is always determined by the receiving script either through restart files (State_Save_File ...) from a previous processing or via parameter `DB2_Start_RBA` and `DB2_End_RBA`.

The sending script does not use the parameters `DB2_Start_RBA` and `DB2_End_RBA` and it does not use restart files (State_Save_File ...), because it is controlled by the receiving script and acts like a server.

8.6.4 Scenario for Db2 z/OS NRT Capturing using User Defined Table (UDT) - Agentless

The capturing of changes from the active log of Db2 on z/OS can be done using a User Defined Table. With this type of processing no active tcVISION component (Host Agent) is required to run in z/OS. This processing is possible in Db2 version 10 with fix PM90568 and in Db2 version 11 or higher without any restrictions.

The tcVISION UDT is defined with the following CREATE statement:

```
CREATE FUNCTION TCVUDT_V7 (VARBINARY(300),VARBINARY(32000))
RETURNS TABLE (OUTVALUE BLOB(200K))
PARAMETER CCSID EBCDIC
EXTERNAL NAME TCSD2UDT
LANGUAGE ASSEMBLE
PROGRAM TYPE SUB
STAY RESIDENT YES
CONTINUE AFTER FAILURE
WLM ENVIRONMENT wlmenv
RUN OPTIONS
'H(,,ANY),STAC(,,ANY,),STO(,,,4K),BE(4K,,),LIBS(4K,,),ALL31(ON)'
PARAMETER STYLE SQL
NO SQL
NO EXTERNAL ACTION
FENCED
SCRATCHPAD
FINAL CALL
DISALLOW PARALLEL;
```

wlmenv is one of the defined Application Environments in the z/OS WorkLoadManager (WLM). The tcVISION loadlib containing the module TCSD2UDT needs to be added to the JCL of the Application Environment defined procedure in the STEPLIB concatenation. All loadlibs of the steplib concatenation need APF authorization because of the used IFI calls.

The user Id of the calling script requires the following Db2 authorizations:

```
GRANT EXECUTE ON FUNCTION TCVUDT_V7 TO userid
GRANT MONITOR2 TO userid
```

The definition of the calling scripts on the workstation is now `Input_Source_Type = 'IFI_306'`. `Input_Source_Name` contains a connection string for DRDA or Db2 Client to the z/OS Db2.
For both connection types, a specific database name (e.g. Db2 subsystem name instead of the Db2 location name) that should be used for further processing (e.g. database in repository, etc.) can be specified with the optional parameter SUBSYS=.

If the creator of the tcVISION UDT does not correspond to the Userid of the calling script, the full name of the tcVISION UDT together with the parameter UDT_Name has to be specified, for example:

```
UDT_Name = 'CREATOR.TCVUDT_V7'
```

For Db2 version 11 or lower: In case of the Db2 error message: `SQL Code: -904. Reason=00C900D1`, the tcVISION UDT is attempting to send more data than the Db2 generation parameter LOBVALA allows.
In this case, the Db2 generation parameter LOBVALA can be increased – the default of 10240 KB is recommended as a minimum, but preferably it should be increased to 40960 KB.
The parameter DB2_LOBVALA can be used to assign a minimum limit in KB to avoid this error:

```
DB2_LOBVALA = '10240'
```

It is recommended that the value of the Db2 generation parameter LOBVALA be less than 10%.

If an update of tcVISION UDT is necessary, the module on the LOADLIB has to be changed and then reloaded:

```
MODIFY WLM,APPLENV=environment-name,REFRESH
```

The parameter IFI306_COMMIT_INTERVAL is used to limit the reading process of the UDT data in time. After this interval in seconds, the Stored Procedure sends the data that was read up until then to the UDT client. Values from 1 to 60 seconds are valid. For the UDT Stored Procedure, a value from 5 to 10 seconds is recommended:

```
IFI306_COMMIT_INTERVAL = 5
```

With the parameter DB2_UDT_MAX_BLOCKS, the reading of UDT data is also limited in terms of volume . After reaching the number of blocks, the Stored Procedure sends the data read up to that point to the UDT client. Values from 0 to 5000 are permitted, whereas the value 0 does not lead to a limitation:

```
DB2_UDT_MAX_BLOCKS = 500
```

8.6.5 Db2 z/OS LOB and XML Columns

The replication of LOB (BLOB, CLOB, DBCLOB) as well as XML columns is supported for all processing types such as bulk (direct and imagecopy) and NRT CDC.
LOB and XML columns with data compression (zEDC or Fixed Length/Huffman) are also supported.

For imagecopy processing of tables with LOB and/or XML columns, in addition to the imagecopy of the base table, the imagecopies of all LOB and/or XML columns must be available in the same directory and the processing can only be done on a Linux/Unix/Windows server.
With the optional parameter `ImageCopy_LOB_Path=` a different path can be specified for these LOB/XML column imagecopies.
For processing the LOB/XML column imagecopies, one temporary hash database per imagecopy is created in the tcVISION “WorkDir” and is not deleted after the processing. Thus, these can be reused for further processing of the same imagecopies without being recalculated.
With the optionale parameter `PM_I.ImageCopyRemoveHashDB=1` these hash databases are deleted after processing.

Restrictions for XML NRT CDC
With the Db2 command XMLMODIFY() columns can be partially modified in such a way that the complete content of the XML column is no longer written in the Db2 log, but only the modified part of it.
In this case an NRT processing is not possible. The NRT process terminates the processing with an error message about missing XML pages and the following addition:
"XMLMODIFY() may cause missing CDC XML Pages".

8.7 IMS/DB (DLI) Log Processing

The tcVISION Data Capture function is based on the standard IMS log record type X'99'. IMS writes these log records asynchronously into the Online Log Datasets (OLDS). The IMS Archive

Log Utility processes these OLDS and creates the SLDS (System Log Datasets). The SLDS are input to the tcVISION Data Capture option.

Data Capture can be used for the following IMS database types: SHISAM, HISAM, HDAM, HIDAM, and DEDB.

Adaptation of the IMS-DBD-SEGM-Generation
The creation of the X'99' log records is activated using the EXIT= parameter in the DBD or SEGM generation macros. Refer to IMS Utilities Ref: System.

```
DBD    NAME=DI21PART,ACCESS=(HISAM,VSAM),                    X
       EXIT=(*,LOG,DATA,NOPATH,(CASCADE,DATA,NOPATH))
```

In addition, the relevant IMS procedures (ACBGEN, etc.) must be executed.

EXIT=-parameter in the DBD macro relate to all following SEGM macros.
EXIT=-parameter in the SEGM macro only relate to the specific segment.

By specifying ...(`CASCADE,NODATA,NOPATH`) only the key values of child segments in the KFA (Key Feedback Area) are included in the log for a cascading delete. Therefore, a cascading delete can only be transferred to the target tables connected to those child segments with fields from the KFA. If key fields of data from segments affected by a cascading delete should also be included, the Exit command should be as follows:

```
DBD    NAME=DI21PART,ACCESS=(HISAM,VSAM),                    X
       EXIT=(*,LOG,DATA,NOPATH,(CASCADE,DATA,NOPATH))
```

If the logging function has already been activated for other purposes and the default KEY has not been deactivated using NOKEY, no changes are necessary. tcVISION will ignore DATA and PATH data in the log records.

8.7.1IMS Logreader

With Input_Source_Type = 'IMS_LOGREADER' the tcVISION IMS Logreader is activated in z/OS.
The JCL must specify a 'hlq.SDFSRESL' for the DSPAPI module and the RECONs member to allocate the RECON files
With the DBRC API, all relevant information is obtained to process the desired log data from the corresponding online log files OLDS and archive log files ALDS.

A typical scenario is a sending script that sends the data in stage 0 format with preselection to a remote receiving script.
In this scenario the start timestamp is always determined by the receiving script, either through a restart file (State_Save_File ...) from a previous run or with the parameter Input_From_Timestamp.
The sending script does not use the parameter Input_From_Timestamp and does not use a restart file (State_Save_File ...), because it is controlled as a server by the receiving script.
A timestamp can be specified for the first start of the IMS logreader. Without this specification the logreader starts processing at the beginning of the most current OLDS.

IMS Write Ahead Datasets WADS:
In an IMS system with low activity the logreader may process the most current log records from the active OLDS system with a delay or not at all, because IMS only writes complete log blocks. In this case the name of the active Write Ahead Data Set (WADS) can be specified with the keyword WADS1= in the parameter Input_Source_Name. When encountering the end of the active OLDS, the logreader processes the WADS for the most current log records.

The IMS command /SWI WADS closes the current WADS and switches to the next WADS(s). In this case the logreader does not find the most current log records and will behave as if no WADS file has been specified.

IMS Data Sharing:
Every member of an IMS-Sysplex- or Non-Sysplex-Data-Sharing environment requires its own tcVISION logreader.
The respective receiver scripts additionally write a status file with the name 'imsdsstat_*SSID*.bin' which contains information about the processing of the transactions in the output directory in the following Apply script in the correct time order.
This Apply script only processes data with an ET timestamp <= the smallest common timestamp of the 'imsdsstat-*SSID*.bin' files.

For an IMS member A with little load traffic, the following issue comes up in addition to the above (WADS): Transactions of other members B,C,... cannot be further processed until A has written log records with a higher timestamp.
This is the task of the log readers that write user entries into the IMS log or send shutdown timestamps to the receiver script if their respective member is inactive or has shut down (no new log records after NRTLR_MinWaitInterval). This way, the latency of an IMS Data Sharing environment can be kept very low with little system load.

Parameter for IMS Logreader:
The following parameters are used:

NRTLR_MinWaitInterval = n.
n is a value in milliseconds with a default of 600. The value n specifies the time interval in which the script checks for new IMS log data at the end of the active IMS log.
If there is no new data, this time interval is increased by the value n until the maximum value of the parameter NRTLR_MaxWaitInterval is reached.

NRTLR_MaxWaitInterval = m.
m is a value in milliseconds with a default of 1500 and specifies the maximum time interval after which the script checks for new IMS logs at the end of the active IMS log.

In a Data Sharing environment, the following parameters are used:
DS_MEMBER_TIMEOUT_CHECK = n.
n is a value in seconds with a default of 10 that specifies from which time difference between the current system time and the last timestamp of a member of a cluster at which the Apply script terminates the processing, because there might be a disruption in the respective IMS log reader.
This value can be increased if, for example, the system times are not synchronous.

NRTLR_MaxBackInterval = n.
n is a value in milliseconds with a default of 2000. The processing assumes that there are no log files with a timestamp older than the current system time minus the value n ready to be processed for the log reader of an inactive member. If this happens, the processing will be terminated with the respective error message and the value n should be increased.

8.8 Db2 for Linux, UNIX, and Windows

8.8.1 Connect String

Connections into a Db2 LUW system are defined as described above with the parameters DATABASE=, UID=, and PWD=.

The parameter DATABASE= references to a local or remote database.

For remote systems this can be defined with a local alias as follows:

Node definition:

```
db2 catalog tcpip4 node my_linux remote 192.168.33.210 server 50001
```

Database alias definition:

```
db2 catalog database SAMPLE as SMPLINUX at node my_linux
```

8.8.2 Log Processing

Currently, the versions 9.7.x, 10.1.x, 10.5.x, and 11.1.x of Db2 for Linux, UNIX, and Windows are supported.

The following requirements must be given to process changes applied to Db2 for Linux, UNIX, and Windows:

The protocol archive method must be specified in Db2:

- DB2 9.7.x:
  - UPDATE DB CFG USING userexit YES, or
  - UPDATE DB CFG USING logretain RECOVERY
- DB2 10.1.x:
  - UPDATE DB CFG USING logarchmeth1 LOGRETAIN

The corresponding tables must have the attribute 'DATA CAPTURE CHANGES'. This is mandatory to completely process INSERT, DELETE, and UPDATE changes.

As of version 9.7.2, tables may have been defined with 'deep compression'. In that case the corresponding log records are compressed and can be processed using the Db2 interface 'ReadLog'.

Longfield and LOB fields are supported as long as the logging of those fields has not been suppressed in Db2 (i.e. wrap around logging), the archive logging is active and the total record length does not exceed the value defined in the parameter LOG_REC_SIZE.

The Data Capture Flag for a specific table can be switched on using the following command:

```
ALTER TABLE schema.tblname DATA CAPTURE CHANGES
```

8.8.3 Scenario for Db2 LUW Remote Scripts

A typical scenario is a sending script that selects the data and sends it in stage 0 format to a remote receiving script.

In this scenario the start LSN/LRI is always determined by the receiving script either through restart files (State_Save_File ...) from a previous processing or via the parameters DB2_Start_LSN/LRI and DB2_End_LSN/LRI.

The sending script does not use the parameters DB2_Start_LSN/LRI and DB2_End_LSN/LRI, and it does not use restart files (State_Save_File ...) because it is controlled by the receiving script and acts like a server.

8.8.4 Bulk Load via Db2-API-db2Ingest()

tcVISION supports the Db2 LUW API function db2Ingest() starting with Db2 LUW V10.1 with the latest FixPack.
This allows large volumes of data to be written directly from a processing script to a Db2 database.
The db2Ingest() call runs in a separate thread, as does writing the data in the script into the operating system pipe for passing to the db2Ingest() thread.

Restrictions:

- It is not possible to load multiple output tables in a script at once.

To use db2Ingest(), s Db2 LUW loader script created for writing the data into a Db2 LUW loader file can be used as basis.
The output target has to be changed from Output_Type = 'FILE' for the loader file to Output_Type = 'DB2', and the connection parameters for the desired Db2 database have to be defined in Output_Target_Name.

Input parameters for db2Ingest()
In the prolog, optional parameters for controlling the db2Ingest thread can be entered if the default values used in the script are not appropriate. The meaning of the parameters and values is described in the manual 'Db2 LUW Administrative API Reference' under db2Ingest().

SQL command
`DB2INGEST_SQL=sql-command`
Default value:
INSERT INTO *table* (*Fieldlist*) VALUES(*valuelist*)

Without the specification of `DB2INGEST_SQL` an INSERT command is generated internally.
Further INGEST options like DELETE, MERGE, REPLACE, and UPDATE can be activated manually with appropriate SQL commands in parameter `DB2INGEST_SQL.`
The syntax can be found in the manual ‘Db2 Command Reference‘.
Example:

```
DB2INGEST_SQL = "UPDATE DEMO.COPY2 SET TEXT=$TEXT WHERE ID=$ID AND TEXT=$TEXT
AND NUMBER=$NUMBER"
```

Name of internal pipe:
`DB2INGEST_PIPE_FILE=`*`pipe-name`*
Default value:
Windows: *ScriptName.UTC-TS-.TableId*
(The Windows-specific prefix „\\.\pipe\“ is internally added by the script.)
UNIX: */tmp/ScriptName.UTC-TS-.TableId*

A pipe must not be used by parallel scripts at the same time.
In UNIX a new pipe is created (mkfifo()) if there is no pipe with the specified name. In this case, the pipe is deleted after processing.

Db2 message file:
DB2INGEST_MESSAGE_FILE=*db2-message-file-name*
Default value:
TraceDirectory.*ScriptName*.*UTC-TS*-db2msg.*TableId*.txt

Db2 dump file:
DB2INGEST_DUMP_FILE=*db2-dump-file-name*
Default value:
No default value, file is not used.

Db2 exception table:
DB2INGEST_EXCEPTION_TABLE=*db2-exception-table*
Default value:
No default value, table is not used.

Db2 job id:
DB2INGEST_JOBID=*db2-jobid*
Default value:
No default value, job id is not used.

Db2 configuration parameters:
The numeric db2Ingest() configuration parameters
DB2INGEST_CFG_COMMIT_COUNT
DB2INGEST_CFG_COMMIT_PERIOD
DB2INGEST_CFG_NUM_FLUSHERS_PER
DB2INGEST_CFG_NUM_FORMATTERS
DB2INGEST_CFG_PIPE_TIMEOUT
DB2INGEST_CFG_RETRY_COUNT
DB2INGEST_CFG_RETRY_PERIOD
DB2INGEST_CFG_SHM_MAX_SIZE

are recorded as shown in the following example if the default values are not appropriate:
DB2INGEST_CFG_COMMIT_COUNT=5

The following default values are used in the script:

Parameter	Min. value	Max. value	Default value
COMMIT_COUNT	0	100000000	5000
COMMIT_PERIOD	0	100000000	0
NUM_FLUSHERS_PER	0	100	0
NUM_FORMATTERS	0	100	0
PIPE_TIMEOUT	0	100	5
RETRY_COUNT	0	100	0
RETRY_PERIOD	0	100	0
CFG_SHM_MAX_SIZE	0	100000000	0

Db2 restart parameters:
DB2INGEST_RESTART_MODE=*db2-restart-mode*
The parameter value *db2-restart-mode* should be entered as number according to the list below.

Default value:
DB2INGEST_RESTART_OFF

```
DB2INGEST_RESTART_DEFAULT      0
DB2INGEST_RESTART_OFF          1
DB2INGEST_RESTART_NEW          2
DB2INGEST_RESTART_CONTINUE     3
DB2INGEST_RESTART_TERMINATE    4
```

Output parameters after db2Ingest() processing
The following REXX variables are specified after processing:

The name of the message file:
PM_O.DB2INGEST_MESSAGE_FILE
The output value of the Ingest thread:
PM_O.DB2INGEST_RowsRead
PM_O.DB2INGEST_RowsSkipped
PM_O.DB2INGEST_RowsInserted
PM_O.DB2INGEST_RowsUpdated
PM_O.DB2INGEST_RowsDeleted
PM_O.DB2INGEST_RowsMerged
PM_O.DB2INGEST_RowsRejected
PM_O.DB2INGEST_NumErrors
PM_O.DB2INGEST_NumWarnings
PM_O.DB2INGEST_MaxMsgSeverity

8.9 Oracle for Linux, UNIX, and Windows Log Processing

The tcVISION Data Capture for Oracle databases is based on the Oracle LogMiner.
Currently, versions 10.1, 10.2, 11.1, 11.2, 12.1, 12.2, from tcScript revision 2165 version 18.3 and from revision 2912 also Oracle 19c and 21c are supported in 'single instance' mode.

We highly recommend to use the Oracle Client version 19.x or later for the tcVISION data capture. This cient can also be used with older Oracle database versions.

To process Oracle-RACs, specify `Input_Source_Type = 'ORA_LOGM'`. In this case the active and archived log files must be available on the shared disk used in the cluster.

The following requirements must be met in order to process changed data from Oracle for Linux, UNIX, and Windows:

- The affected database must have 'Minimal Supplemental Logging' active. The corresponding Oracle command is:
  `ALTER DATABASE ADD SUPPLEMENTAL LOG DATA;`

- To ensure that the required key fields for UPDATE commands in the log are available, the supplemental logging options must be activated.

  1) on database level
  ```
  ALTER DATABASE ADD SUPPLEMENTAL LOG DATA (PRIMARY KEY) COLUMNS;
  ```
  This is the most effective way and does not need any further test on table level.

  2) on table level
  ```
  ALTER TABLE table_name ADD SUPPLEMENTAL LOG DATA (PRIMARY KEY) COLUMNS;
  ```
  This way requests a test on table level.

  3) on table and field level
  ```
  ALTER TABLE table_name ADD SUPPLEMENTAL LOG GROUP lg1 ("Field1") ALWAYS;
  ```
  This way requests a test on table level including a repository access.

  As of revision 5451 a test will be done whether for all the selected tables the requirements noticed above are met. If not, the processing will be stopped to avoid missing or incorrect data.

  Using the parameter PM_I.ORACLE_SL_CHECK = n in the prolog of the script the default behavior (level 2) can be changed as follows:

  2: If 1), 2) or 3) met the processing starts.
  1: If 1) or 2) met the processing starts.
  0: The processing starts without any test

  In case the content of all columns should be transferred (i.e. JSON or Avro objects, also when using the ´all field update‘), the command must be:
  ```
  ALTER DATABASE ADD SUPPLEMENTAL LOG DATA (ALL) COLUMNS;
  ```

  UPDATE commands of any kind, like
  UPDATE TESTTAB SET KEY1=KEY1, FELD1='New Value'

  cause the required key value of KEY1 to be part of the log.

LONG and LOB fields are supported as long as the logging of these fields is not suppressed in Oracle (NOLOGGING). Currently, a data length of up to 4000 bytes is supported.

Privileges required for logmining can be granted using these commands:

- Before 12c:
  ```
  GRANT CREATE SESSION,SELECT ANY
  TRANSACTION,SELECT_CATALOG_ROLE,EXECUTE_CATALOG_ROLE TO myuser;
  ```

- Starting from 12c, non-multitenant:
  ```
  GRANT CREATE SESSION,SELECT_CATALOG_ROLE,EXECUTE_CATALOG_ROLE,
  LOGMINING TO myuser;
  ```
- For a multitenant DB a common user is required. It has to be created in the root container(*). You can use a user-created common user (like c##myuser) or a Oracle-supplied one (like SYS or SYSTEM). The privileges are:
  ```
  GRANT CREATE
  SESSION,SELECT_CATALOG_ROLE,EXECUTE_CATALOG_ROLE,LOGMINING TO
  c##myuser;

  ALTER USER c##myuser SET CONTAINER_DATA=ALL CONTAINER=CURRENT;
  ```
  or (if **you know** that the log records will come from some specific containers only) (**):

```
ALTER USER c##myuser SET CONTAINER_DATA=(<container name1>,
<container name2>, ...) CONTAINER=CURRENT;
```

(*)It is only allowed to start LogMiner in the root container, thus to use CDC an Oracle user has to be configured there. However, if you only plan to use operations like Bulk or Batch Import and do not need CDC, create an Oracle user in the pluggable database.

(**) For the root container use "CDB$ROOT" in the second ALTER USER ... SET CONTAINER_DATA command.

- To use Binary Reader add this:
  `GRANT CREATE ANY DIRECTORY TO myuser;`

8.9.1 Scenario for Oracle Archive Log Processing

Short description to activate archive logs:
The command

ALTER SYSTEM SET LOG_ARCHIVE_DEST='C:\ora10\logarch' SCOPE=SPFILE;

specifies the target directory for the archived log files. The database is stopped using the command

SHUTDOWN IMMEDIATE;

Then:

STARTUP MOUNT;
ALTER DATABASE ARCHIVELOG;
ALTER DATABASE OPEN;

The command

ALTER SYSTEM SWITCH LOGFILE;

closes the 'current active log' and its content is saved into the next archive log file in the directory that was specified with the above command.

8.9.2 Scenario for Oracle Remote Scripts

A typical scenario is a script that sends the preselected data in the stage 0 format to a remote receiving script.
In this scenario the starting SCN is always determined by the receiving script, either through a restart file (State_Save_File ...) from a previous processing or through the parameters ORACLE_Start_SCN and ORACLE_End_SCN.
The sending script does not use ORACLE_Start_SCN and ORACLE_End_SCN and does not use a restart file (State_Save_File ...), because the script acts as a server and is controlled by the receiving script.

8.9.3 Szenario for Oracle Flashback

A typical scenario for this case runs as follows:
1.) A CREATE RESTORE POINT rp1 ... creates a restore point in the DB.
2) The processing on the DB is continued.
3.) The changes made under 2. to the DB are to be rolled back.
For this the DB is shut down and a
FLASHBACK DATABASE TO RESTORE POINT rp1
is executed, followed by a startup of the DB.

Thus the DB corresponds again to the state as under 1.

The correct replication for this scenario can be achieved by
activating the exit CDC_ExitDBControl in the reader script as follows:

```
CDC_ExitDBControl:
X_DBC_O.ReturnFlags = 0
RP_Name = 'TCV_RP_01'

IF X_DBC_I.CallType = 60 THEN DO
 RestorePoint = STRIP(WORD(X_DBC_I.ORA_RESTORE_POINT, 4),,'"')
 RP_SCN   = D2X(X_DBC_I.ORA_RP_SCN,12)
 RP_RLSCN = D2X(X_DBC_I.ORA_RP_RL_SCN,12)
 DB_RLSCN = D2X(X_DBC_I.ORA_DB_RL_SCN,12)
 IF UPPER(RestorePoint) = RP_Name THEN DO
  IF RP_RLSCN = DB_RLSCN THEN DO
   /* Here we found a CREATE RESTORE POINT with the DB RLSCN equal to our current RLSCN -
> Stop replication */
   Msg = "Create Restore Point '"RestorePoint"', SCN("RP_SCN"), RP_RLSCN("RP_RLSCN") =
DB_RLSCN("DB_RLSCN"), stop replication"
   SAY Msg
   X_DBC_O.ReturnFlags = 8
   RETURN '0,'Msg
  END
  IF RP_RLSCN < DB_RLSCN THEN DO
   /* Here we found after restart the same CREATE RESTORE POINT with a new DB RLSCN
(after the FLASHBACK) -> Resume replication at this RLSCN*/
   Msg = "Create Restore Point '"RestorePoint"', SCN("RP_SCN"), RP_RLSCN("RP_RLSCN") <
DB_RLSCN("DB_RLSCN"), resume replication with DB_RLSCN"
   SAY Msg
   X_DBC_O.ReturnFlags = 9
   RETURN '0,'Msg
  END
 END
 RETURN "0,Create Restore Point '"RestorePoint"' not processed"
END

RETURN '0,No Error'
```

In the above example for the RESTORE POINT TCV_RP_01 when reaching the corresponding CREATE RESTORE POINT...-command in the log, the exit is called and processing is terminated with ReturnFlags=8, because the Resetlogs-SCN of TCV_RP_01 matches the current DB-Resetlogs-SCN and the processing is therefore in step 1.
If the script is started again, it resumes processing at this CREATE RESTORE POINT due to the restart info and terminates again exactly the same way.

If now the steps 2. and 3. are executed on the DB and the script is started, then this recognizes that the DB-Resetlogs-SCN does not correspond any more with the Resetlogs-SCN of TCV_RP_01 and consequently the DB was started again with OPEN RESETLOGS.
With ReturnFlags=9 the script stops processing at TCV_RP_01 and resumes processing at the current DB resetlogs SCN. Thus the area between TCV_RP_01 and the current DB-Resetlogs-SCN is not replicated as desired.

In case the area between TCV_RP_01 and the current DB-Resetlogs-SCN should be replicated anyway or a FLASHBACK is no longer desired, the ReturnFlags=9 or ReturnFlags=8 respectively can be replaced by a ReturnFlags=0. Thus TCV_RP_01 is read over and the processing is continued immediately afterwards.

8.10 Microsoft SQL CDC Processing (SSIS)

The tcVISION Data Capture for MS SQL Server is based on the Change Data Capture mechanism.

Data capture with MS SQL Server CDC is supported for the Enterprise Editions of MS SQL Server 2008 R2, 2012, 2014, and 2016. Starting from MS SQL Server 2016 SP1, the standard edition is supported as well.
The CDC data extraction must be performed on a Windows machine with the installed “SQL Server Native Client” ODBC driver. The supported versions are:

SQL Server Native Client 10.0
SQL Server Native Client 11.0
SQL Server Native Client 12.0
SQL Server Native Client 13.0
SQL Server ODBC driver 17.x

The following requirements must be met to process changes from MS SQL CDC data:

- The database in question must be activated for CDC. The corresponding command sequence can be:
```
use [master]
exec sys.sp_cdc_enable_db
GO
```

- Each table in question must be activated for CDC. The corresponding command sequence can be:
```
use [master]
EXEC sys.sp_cdc_enable_table
@source_schema        = N'schema',
@source_name          = N'tablename',
@role_name            = NULL,
@capture_instance     = N'tcvision_tablename',
@supports_net_changes = 1
GO
```

When setting up the database, pay attention to the setting of the polling interval for the log scan. The default value is five seconds. This setting can be changed via the Stored Procedure ”sys.sp_cdc_change_job“. In an Always On cluster, the values on all machines should be equal.

In a bidirectional replication, a table is created in the output target in a specified schema. This table is used as the identifier for tcVISION. For different output targets to the same MS SQL Server, a different schema has to be chosen to avoid database blockades on this table in parallel operation.

8.11 IDMS Journal Processing

The IDMS journal files will be written to tape by an IDMS instance. IDMS writes journal files INDEPENDENTLY from open and incomplete LUWs. This means that, with the exception of the first journal file after an IDMS start, all subsequent journals can contain references from open LUWs in previous journal files. Therefore, these files cannot be processed independently from the previously created journal files.

These overlapping references can be correctly processed using the following variations:

1. All journal files of an IDMS DBDC instance will be processed after termination of this instance by copying all journal files onto disk and specifying the same sequence that has been used by IDMS to create the files using the parameter Input_Source_Name. The data script internally saves the open LUWs when changing to the next journal file.
2. The journal files of an IDMS DBDC instance will be individually processed. Use parameter State_Save_File_Out to save the state of open LUWs at the end of the processing into a status file. When processing the subsequent journal file, specify this status file with parameter State_Save_File_In and the open LUWs will be "matched" with the current file. You can again use parameter State_Save_File_Out to create a new status file at the end of the processing of this file. You can use the same name for the status file. After processing the last journal file the status file will have a length of zero, because no open LUWs exist anymore. Using this status file you can directly process the journal files of an additional IDMS DBDC instance.
3. The journal files of an IDMS DBDC instance will be passed in sequence from a SENDER script to a RECEIVER script that has been started with option INPUT_FLAGS_CONTINUE_LISTEN. In this case, the instance of the RECEIVER data script remains and internally saves the open LUWs when waiting for the next journal file.

In all of these cases, the CORRECT sequence of the journal files is important. If the sequence is not correct, processing will be terminated because of incorrect timestamps.

8.11.1 Direct Processing of Journal Files

The data script that reads the tape files can directly process the data into the SQL DML format (stage 3). The parameter IDMS_AREA_RECID defines the area and records that must be processed.

8.11.2 Read and Directly Pass Journal Files

The data script that reads the tape files can transfer the data to a remote data script without any processing (stage 0). In this case, the entire journal file will be read and transferred.

This processing should be considered if the changes of a relatively large number of records should be processed on a remote system. The tape files can also be copied to disk and then be transferred to another target system using FTP. Processing takes place on that system using a data script.
Specify parameter Input_Type = 'IDMS_JOURNALREC'.

8.11.3 Preselect and Transfer Journal Data

The data script that reads the tape files can also preselect the data and only pass the data relevant for the selected tables. The output takes place in stage 1 format. The entire journal file is read, but only the selected data will be passed to the remote script.
Consider this processing type if a relatively small volume of changes of a few tables should be processed on the remote system.

8.11.4 Journal Data from the Collector/Pool

Journal data from the IDMS exit should be immediately saved to an active collector/pool when an IDMS instance is started. This is necessary to completely gather the journal data. If the journal data derived from the exit contains time gaps, overlapping references may probably not be correctly resolved and images without the proper reference to the corresponding IDMS area will exist.

8.11.5 Journal Data from the Active Journal Files

Journal data from the active journal files allow near real-time processing. If all JxJRNL files are specified in the parameter Input_Source_Name=, the script can directly read the respective journal data.
It is also possible to continue the processing in the active journal files with the current restart file after processing the latest archive journal by specifying all JxJRNL files in the parameter Input_Source_Name=.

8.12 IDMS SR2/SR3 Processing

Under certain conditions (for example: database reorganization) IDMS data records can be segmented into SR2 and SR3 data records.
For the processing of these data records in journal entries tcVISION must track the corresponding DBKEYs in an internal SR23 database.

The initial load of the internal SR23 database is performed using a script with a definition of

```
Input_Type = 'BULK_TRANSFER'
```

and

```
Input_Source_Type = 'IDMS_SR23'
```

8.12.1 Mandatory Parameters:

```
IDMS_SR23_PROC = 'abc'
Input_Source_Name = 'dmcl'
```

The parameter IDMS_SR23_PROC = '*abc*' specifies the processing options as follows:

'*a*' -> Pre-processing options:
'N': No pre-processing

'T': Existing entries in the SR23 database are terminated with a processing timestamp - 1.
'D': Existing entries in the SR23 database are deleted.
The pre-processing is only executed once per AREA and only SR2/SR3 data records are found for the corresponding area.
Options 'T' and 'D' only refer to existing SR23 database entries with the same PageGroup, AREA, SR2-DBKey, and SR3-DBKey, and a timely overlap with the processing timestamp.

'*b*' -> Processing options:
'N': No direct processing of the SR23 database.
'A': The read SR2/SR3 data records are compared to the existing entries in the SR23 database and the database is updated if necessary.
'I': The read SR2/SR3 data records are inserted into the SR2 database without further checks.
This processing type is faster than 'A', however the user is responsible for the integrity of the SR23 database.

'*c*' -> Post-processing options:
'N': No further post-processing.
'B': The read SR2/SR3 data records are output as bulk transfer.
They can be further processed using the documented functions, for example: Use the LOADER for a fast load into the SR23 database.
The bulk structure (IDMS_SR23) looks like the following:

```
char    GroupId[50];
char    SourceName[24];
char    ValidFrom[26];
char    ValidUntil[26];
char    S3_DBKey[8];
char    S2_DBKey[8];
char    RecordId[4];
```

The parameter Input_Source_Name = '*dmcl*' specifies the name of the DMCL that should be used. The DMCL is processed in binary and interpreted.

8.12.2 Optional Parameters:

```
Input_From_Timestamp = 'ValidFrom'
Input_Source_Name.1 = 'DataPageFileName 1'
...
Input_Source_Name.n = 'DataPageFileName n'
```

The parameter Input_From_Timestamp = 'ts' specifies a user-defined ValidFrom timestamp for the entries in the SR23 database. Without this parameter, the current UTC is used.

Definition of the IDMS data files that are processed

Method 1
With the additional `Input_Source_Name.n = 'DataPageFileName n'`, it is possible to specify the files from 1 to n that should be processed directly. The FILE entries specified in the DMCL are not used.

Method 2
If parameter Input_Source_Name.n = '*DataPageFileName n*' has not been specified, the FILE entries specified in the DMCL are processed and the corresponding IDMS data files are read using the DD names, hence DD statements must be specified in the JCL. If an IDMS file cannot

be opened, processing continues with the next FILE entry but no error message is issued. A corresponding entry is written into the data trace.

Processing on a workstation
The processing of SR2/SR3 can also take place on a workstation. This requires a transfer of the DMCL and the IDMS data files to the workstation.

8.13 IDMS Backup Processing

IDMS records can also be read from IDMS backup images which were created with the utility 'IDMSBCF' and the 'BACKUP' command.

These backup images can be processed directly on the mainframe or on the workstation after a binary transfer e.g. via FTP and the options 'bin' and 'quote site rdw'.

When processing from stage 0 to stage 1+, a connection to an SQL database can be optionally defined with the parameters IDMS_KEY_CONNECT_TYPE and IDMS_KEY_CONNECT_NAME to read the key fields that belong to this member record for the IDMS records with the OWNER-DBKEY values.
For this kind of processing, the respective owner records have to be available in the SQL database at the time of the query. The processing will be slowed down because of the SQL accesses.

For a better throughput, the processing can also be performed without the SQL connection described above. In this case, the owner key fields are filled with dummy values in the member records. The respective constraints have to be deleted temporarily in order to avoid constraint corruptions in the target database. This, in turn, has very positive effects on the throughput. The bulk from the BACKUP image onwards can be executed in any order.
After the bulk, the owner key fields in the member records have to be filled with the correct values. This is done with the following SQL commands:

Example EMPLOYEE from the database IDMS EMPDEMO:

```
UPDATE public."EMPLOYEE" t1 SET "OFFICE_CODE_0450" = (SELECT "OFFICE_CODE_0450"
FROM public."OFFICE" t2 WHERE t2."DBKEY" = t1."DBKEY_OFFICE") WHERE
"DBKEY_OFFICE" > 0;

UPDATE public."EMPLOYEE" t1 SET "DEPT_ID_0410" = (SELECT "DEPT_ID_0410" FROM
public."DEPARTMENT" t2 WHERE t2."DBKEY" = t1."DBKEY_DEPARTMENT") WHERE
"DBKEY_DEPARTMENT" > 0;
```

Afterwards, the constraints have to be redefined.

8.14 Datacom Log Processing

The tcVISION data capturing for Datacom is based on log file 'deferred' processing.

8.14.1 Scenario for Log File Processing

Datacom log files are processed on the mainframe. Log files must be processed in the same sequence in which they have been created by Datacom. The script that processes the data from stage 0 to stage 1+ requires restart files because log files may contain open LUWs and to ensure the correct sequence of the log files. The files are read by the READRXX interface.

8.14.2 Scenario for CDC File Processing

A script can process the Datacom CDC tables TSN and MNT and directly read and process the new log data that has been created by Datacom.
The Datacom installation must be configured according to the instructions provided by the vendor. The CDC tables TSN and MNT must contain the corresponding CDC data records. An example of the URT for reading the Datacom CDC tables TSN and MNT is saved in the Maclib as TVSDCDCU.
Usually, the option 'INPUT_FLAGS_NRT_CONTINUE_EOF' of parameter Input_Flags should be activated, so that the script waits for the next TSN record when an EOF condition is encountered (the current TSN record has been reached).
By specifying the start time with parameter Input_From_Timestamp=, it is possible to skip transactions with an earlier timestamp during the first start.
The script that processes the data from stage 0 to stage 1+ requires restart files to ensure that a restart is possible in CDC tables TSN and MNT based on the last processed data record.

A typical scenario represents a sending script that sends the data in the internal stage 0 format and with a preselection to a remote receiving script.
In this scenario the restart point in the CDC tables is always controlled by the receiving script, either through the use of a restart file from a previous processing (State_Save_File ...), or optionally using parameter Input_From_Timestamp for the first script execution.

8.15 MySQL

The tcVISION scripts use the MySQL/MariaDB interface 'libmysql.dll/so' or 'libmariadb.dll/so' for all accesses to MySQL data.

MySQL is currently supported from version 5.7.x and MariaDB from version 10.1.x.

8.15.1 MySQL Log Processing

The tcVISION Data Capture for MySQL/MariaDB databases is based on MySQL log data that is read via the tool 'mysqlbinlog' or via the direct log data access.

For this, the following parameters have to be activated in the configuration of the MySQL/MariaDB server:

binlog-format = ROW
binlog_row_image = full or *minimal*

A log reader script connects to the MySQL/MariaDB server and reads the variable 'log_bin_basename'. With this path/prefix, the log data is subsequently read directly if the access is possible, i.e. if the script is running on the server. If the script is running on remote, the tool 'mysqlbinlog' is used.

With the connect parameter 'log_bin_basename=', the script control can be adjusted as follows:

log_bin_basename=MYSQLBINLOG:
The script always reads the log data with the tool 'mysqlbinlog'.

log_bin_basename=*path*[/\]*prefix*:
The script always reads the log data directly from the log files with the prefix *prefix* in the path *path*.

Tip:
If direct access to the log files is not possible, mysqlbinlog can also be used to implement a copy of the log files on the server into a local directory. See “Using mysqlbinlog to Back Up Binary Log Files“.

Example:
mysqlbinlog -u tcVISION -p --raw --stop-never -R -h my-sql-server --start-position=0 mysql-bin.000001

From this local directory, tcVISION Data Capture can process the log files for MySQL/MariaDB directly.

8.16 PostgreSQL

tcVISION scripts use the PostgreSQL interface 'libpq.dll/so' to access PostgreSQL data.

The versions 9.4, 9.5, 9.6, 10.18, 11.13, 12.8, 13.4 und 14.0 are currently supported.

8.16.1 PostgreSQL Bulk/Batch Compare Processing

tcVISION uses the streaming-copy functions for bulk and batch compare input and/or output if PostgreSQL is specified as input and/or output.
PostgreSQL tables are always read using streaming-copy functions.

Bulk/batch compare output to loader files are written in standard format and can be loaded to PostgreSQL, for example by using the PostgreSQL copy command:

```
COPY tcvision.mytable FROM '/mnt/nfs/myloader.txt' (ENCODING 'UTF8');
```

A script can also load the bulk/batch compare output in loader format to one or more output tables directly if the output (PM_I_Output_Target_Name) points to a PostgreSQL database instead of an output file.

This way the script keeps control and no loader files are needed. In this case, the streaming-copy functions are used. This scenario implies the following PostgreSQL limitation:

```
COPY FROM will invoke any triggers and check constraints on the destination
table. However, it will not invoke rules.
```

If this is not desirable, it is also possible to load the data using INSERT statements. To do so, OUTPUT_FTYPE_SQL_LOADER must not be set. This method requires considerably more time to run.

8.16.2 PostgreSQL Log Processing

tcVISION change data capture for PostgreSQL is based on PostgreSQL log data that is read using 'logical decoding'.

tcVISION uses the **pgoutput** plugin (binary plugin) that is part of the PostrgreSQL server installation. No action required to make it available.

The following parameters need to be set in the PostgreSQL configuration (postgresql.conf):
wal_level = logical
max_wal_senders = 10
max_replication_slots=10

Important note: tables for which CDC is to be performed have to be properly configured for replication using primary key/unique index and REPLICA IDENTITY parameter. These settings control which information is written to the slot to identify the rows updated or deleted.

Here are the valid combinations:

1) the table has primary key; the REPLICA IDENTITY is not specified (or, which is the same, explicitly set to DEFAULT).

2) the table has a unique index and PostgreSQL is instructed to use this index for row identification by setting REPLICA IDENTITY to USING INDEX <name>.

3) The REPLICA IDENTITY for the table is set to FULL. In this case the rows are identified by all the columns and all the column values are written to the slot. This is the most resource-consuming option; the logs can be huge. Can still be OK for small and rarely updated tables.

The preferred solutions are (1) or (2), so that the rows are identified by the key or index columns.

The REPLICA IDENTITY can be set using ALTER TABLE command:

```
ALTER TABLE tcvision.mytable REPLICA IDENTITY {DEFAULT | USING INDEX name | FULL}
```

A tcVISION CDC script will create a PostgreSQL replication slot when being started for the first time. This slot is not deleted when the script ends. This way the replication can be continued on the appropriate place.
Replication slots store the start LSN from which tcVISION CDC scripts start replicating.

If the flag INPUT_FLAGS_DELET_INPUT_RC0 is set, the start LSN of the replication slot is kept after the script ends. This way the script can resume at the same LSN. However, the log in PostgreSQL will be blocked from that LSN on which can lead to a log of undesirably big size.

8.17 BigData

When reading from a Big Data Source there is no bulk, as we are reading a log. In order to do an initial load you would setup a realtime replication and start from the earliest possible point. The reason for this is that BigData sources are all either event logs or flat files which contain a journal of operations.

Currently supported is the layer Kafka and the protocol JSON.

8.17.1 Import of BigData structures

The import requires a description depending on the used protocol, with possibly some additional information based on the used layer. Currently no additional information is needed with Kafka as the only implemented layer.

For the protocol JSON a JSON schema definition is needed (see www.json-schema.org).
This is passed as a text file and must contain a title field which is used as the name of the input table. The scheme must be available in its entirety in one file.

8.17.2 Transport Layer Kafka

Kafka messages consist of a message key and the text of the message (payload). The payload is read according to the set protocol and parsed as a JSON document, for example.

The message key is viewed as a binary byte array and is stored entirely in the MsgKey field. A split of this is possible via the process functions in the repository.

Since Kafka is a pure message log and not a database, each log entry is considered an Insert, unless only the key fields are filled, then it would be a Delete.

It is possible to define a field whose value determines whether it is an Insert, Update or Delete. To do this, the desired field must be specified in the replication rules for the respective link of input and output tables in the repository, as well as the regular expressions that can be used to determine which operation corresponds to the data record.

There are no transactions when reading messages from Kafka. The only guarantee is that the order of messages from the same partition of a topic is preserved. It is therefore possible to use several scripts (up to 1 per topic) to read data from Kafka. Care must be taken when writing the data to Kafka that there are no referential integrity problems when the data is written back to a database.

The parameters for Kafka match those of the Kafka output target.
The only exception is Topics=
This parameter is no longer called Topic, since several can now be specified with , separated. Tokens cannot be used, the names of the topics must be specified in full, no wildcards or similar are possible.

8.17.3 Protocol JSON

For the protocol JSON the only configurable setting is the method in which the name of the input table is determined. Either by reading a field on the first level of the JSON document or by using the name of the Kafka topic.

This setting is controlled by the parameter TNS= in the connection string. If it is 0 the name of

the topic is used. If it is 1 there must be parameter TNF= which contains the name of the field in which the table name is specified.

8.18 MongoDB

The mongodb processing requires the mongodb c driver consisting of libmongoc and libbson libraries.
To perform a realtime replication with mongodb as a source a mongodb version 6.0.0 or higher is required.

As mongodb normally does not give structure information the import is based on the schema analysis provided by mongodb compass, which is provided with the mongodb installation. To import a mongodb collection into tcVISION the analyses needs to be pasted into a text file after copying it with the share schema as json option.

This text file should ideally be named databasename.collectionname.txt as mongodb compass does not provide any information about this in the json document. Otherwise the name needs to be specified while importing. The name of the collection is case sensitive.

The connection string parameters are identical to a mongodb output target, except for the bulklvl parameter which is not utilized for mongodb as a source.

Due to BSON objects being unstructured tcscript can handle if fields are present that are not defined in the repository. However these only stay in the data till stage 1 and then are discarded.
If new structure information is imported and saved data in s0 or s1 is read these fields would be processed assuming they were present in the BSON documents.

8.18.1 Log processing

The log processing is based on mongodb change streams and utilizes the FullDocument and FullDocumentBeforeChange fields that were introduced with mongodb version 6 to provide the full after and before images.

In order to be able to use change streams the used mongodb database must be a replica set as otherwise no OpLog exists which is the basis of the change streams.

A tutorial on how to transform a standalone mongodb installation into a replica set can be found here: https://www.mongodb.com/docs/manual/tutorial/convert-standalone-to-replica-set/

First we need to enable the mongodb change stream pre and post images for the collections we want to replicate.

For example this can be done in the mongodb shell using the following command:

```
db.runCommand ( { collMod: "myCollection", changeStreamPreAndPostImages: { enabled: true } } );
```

The value myCollection is set by the name of the desired table.

In order to ensure those images don’t get discarded early you can specify for how long you

need them with the following command on the admin database:

db.runCommand({ setClusterParameter: { changeStreamOptions: { preAndPostImages: { expireAfterSeconds: "off" } } } })

With “off” those images are never discarded. A value in seconds can also be specified for the expireAfterSeconds parameter, but this runs the risk that the data set can no longer be processed with tcVISION if the data set is only called up by tcVISION after this time has elapsed.

Due to change streams not having any event for a transaction start or end and no other possibility to recognize these existing tcVISION replicates changes from monogdb without transactions.

8.19 Mainframe: Start of a Batch Script

In a mainframe environment it is possible to start a script in a separate area (z/OS: address space). This is especially useful for:

- DLI/IMS batch processing, because in this case the scripts has to run under the control of the DLI/IMS batch Manager.
- IDMS batch processing, because the IDMS modules only release their storage at the end of the job. When processing these scripts in the Agent partition, these storage areas are not released and the remaining storage that the partition can use is exhausted.

8.19.1 Start a Batch Script

To start a batch script in a separate area, a control script can be used to read the necessary JCL from a file and write the job into the reader queue of the operating system. The control script may even apply changes to the JCL.

The file containing the JCL can be created with the tcVISION Dashboard application using Process Definitions/Control Processes/New Process. The example shown later reads the JCL from such a source.

The operating system starts the batch script depending on the used parameter and the script itself connects to the Agent to send script status messages and to receive Agent commands.

It is also possible to start a batch script by the standard mainframe job management system.

Example for the control script:

```
CDC_ExitControl:

/* Parse our input parameters */
PARSE VALUE Input_Parms WITH . "AGENT=" AgentInfo .
PARSE VALUE Input_Parms WITH . "TCPIP_NAME=" TcpipName .

/* Open the JCL from the VDI*/
File = 'TCCMDSK:CONFIG:SCRIPTS/VSE_JOB.TXT'
Call Stream  File, 'C', 'OPEN READ'
IF Result <> 'READY:' THEN DO
 SAY result
 call Stream File, 'D'
```

```
 SAY result
 RETURN '12, Unable to access VDI File('File')' result
END

/* Read in the JCL*/
i = 0
DO FOREVER
 InLine = LINEIN(File)
 if LENGTH(InLine) = 0 THEN LEAVE
 InLine = CHANGESTR("<agent>", InLine, " TCPIP_NAME="TcpipName "AGENT="AgentInfo)
 i      = i + 1
 Job.i  = InLine
END
Call Stream File, 'C', 'CLOSE'
Job.0 = i

/* z/OS: Open a Internal Reader to submit the job */
File = 'SYSOUT=A;WRITER=INTRDR;LRECL=80'

Call Stream  File, 'C', 'OPEN WRITE REPLACE'
IF Result <> 'READY:' THEN DO
 SAY result
 call Stream File, 'D'
 SAY result
 RETURN '12, Unable to access Reader' result
END

/* Submit job */
DO i = 1 to Job.0
 Call Lineout File, LEFT(Job.i, 80)
END
Call Stream File, 'C', 'CLOSE'

RETURN '0, No Error'
```

8.20 Websphere MQ and Processing of Log Data

A typical scenario for the processing of log data with WEBSPHERE_MQ may be the following:

A first data script processes and selects the log data of the DBMS and writes the data into queue 1 in stage 0 format.

A second data script retrieves the records from queue 1, groups the log data into individual LUWs in stage 1, and writes them into a new queue 2.
During this processing it is possible to group all records belonging to an LUW with a group ID. When a group is completed (End Transaction), a third data script can be automatically started. The queue name and the group ID can be passed to this script as start parameter. This script then can process this LUW.

Another method: At this point, the third data script is not started per LUW with the corresponding group ID, but it is started as an asynchronous script by the Agent that reads the output queue and processes the LUW that are part of this queue in the correct sequence (all LUWs are sorted based on the END timestamp). At the queue end the script waits until the next LUW has been written into the queue by script 2.

8.21 The Informix Loader

Multiple different formats exist for the database server Informix. These format can be used to load a table using the High Performance Loader. tcVISION supports the format "Delimited". This format separates individual fields by a single vertical bar ("|"). There is no differentiation between character and numeric fields.

The presence of character "|" in the data is indicated by a preceding backslash. This also applies for the backslash itself.

When creating a loader file, the file should have the same CCSID as the database. Otherwise wrong data may be processed or data records might be rejected.

8.22 tcSCRIPT as FTP Replacement

A data script reads a source file and passes the data in the stage 0 format to the receiving script. If this script outputs the data in stage 1 format with option 'After Image', the file gets transferred with no modifications similar to FTP binary processing.

9 Db2 z/OS as of Version 9 and New Function Mode (NFM)

When processing Db2 log data (archive log and active log or IFI306), it is important to note that a table space reorganization is mandatory after an ALTER TABLE ... ADD COLUMN has been performed to tables in table spaces with Reordered Record Format (RRF) that contain variable data types. At the same time these changes to the tables must be applied to the tcVISION Repository (re-import).

10 Adabas: General Considerations

tcVISION/tcSCRIPT processes Adabas change data in either the COMMAND LOG or the PLOG format. If the changed data should be **propagated immediately,** active PLOG files can be processed, or the Adabas COMMAND log can be used by the tcVISION DBMS extension 'Adabas Exit' (refer to manual 'DBMS Extensions'). The tcVISION Adabas exit has only access to fields that have been changed. The consequence of this is that when propagating the changes in output stage 3 (SQL DML statements), a reference is always made to the internal Adabas ISN (Internal Sequence Number). This ISN must be part of the target database in order to have a functional real-time synchronization. Also when using the PLOG, the ISN will be used for all INSERTs.

If a change is performed from the target DBMS to the Adabas database (back replication or bidirectional replication) and if this change results in a new ISN (for an INSERT), the new ISN has to be propagated back to the target DBMS, hence changing the INSERT to an UPDATE. This processing is fundamental and is called BackUpdateCheck or LOOPBACK. The conditions required to perform that check are defined in the repository.

For further information refer to the tips & hints document "Adabas Loopback Processing".

NOTE:

During the synchronization phase, it has to be organizationally ensured that a target database must be reloaded (BULK_TRANSFER) if the ISN of a database has changed due to maintenance work (e.g. UNLOAD/RELOAD).

11 SMF: General Considerations

tcVISION/tcSCRIPT processes z/OS SMF data from SMF Dumps, SMF VSAM files and SMF logstreams.

The processing of SMF data from VSAM or logstreams can be restarted if the last processed SMF block is still available in the active VSAM file or has not yet been deleted from the logstream and processing can continue with the next block.

A typical scenario can be a sending script which sends the data in the stage 0 format and with preselection to a remote receiving script.
In this scenario the restart point in the SMF data is always specified by the receiving script either through the definition of a restart file (State_Save_File ...) from a previous processing or through the parameter Input_From_Timestamp for the first start.
The sending script does not use any start parameters and should not use a restart file (State_Save_File ...) because it is controlled from the receiving script like a server.

When using SMF logstreams, certain record types (e.g. 100-102 for Db2) can already be assigned to specific logstreams in the SMF definitions.
In this case no selection should be specified in the sending script when reading this specific logstream. The SMF logstream is processed in the multi block mode and the blocks are sent to the receiving script unchanged, reducing the CPU consumption to an absolute minimum.

Db2 SMF records in compressed format (SMFCOMP=ON) are supported.

11.1 SMF Dumps:

SMF dumps are created by the SMF programs 'IFASMFDP' and 'IFASMFDL'.
These dump files can be directly processed by a data script. If the SMF dump files should be processed on a workstation, the corresponding z/OS PS file should be transferred to the workstation using FTP with options 'binary' and 'quote site rdw'.
Multiple input files can be specified.

The processing terminates after the last input file.

11.2 SMF VSAM files:

SMF data is written into a set of z/OS VSAM files. One file is always in the 'ACTIVE' state, the others are first in the 'DUMP REQUIRED' and then in 'ALTERNATE' state.

The data script automatically checks the state of the VSAM files and processes the SMF data records up to the most current data block. At this point, processing is either terminated or waits for new SMF data if INPUT_FLAGS_NRT_CONTINUE_EOF is active.

If the VSAM files are specified using the cluster name (e.g. 'SYS1.MAN1'), these VSAM files are processed. This processing type results in multiple open and close of the VSAM files to avoid conflicts with the SMF dump programs.

If the VSAM files are specified using the name of the data file (e.g. 'SYS1.MAN1.DATA'), the files are processed as PS files. This processing type only opens the files once during start and is the recommended type of processing.

Processing terminates after the last input file.

11.3 SMF Logstreams:

For newer versions of z/OS SMF, data can be written to logstreams.

The data script processes the data directly from the logstream until the most current data block. At this point, processing is either terminated or waits for new SMF data if INPUT_FLAGS_NRT_CONTINUE_EOF is active.

12 tcVISION Logstreams

tcVISION change data can be written into a z/OS Logstream. The logstream is coupled with a collector definition. The processing method is only available in z/OS.

No pool definition is required for this processing method and the collector does not occupy a storage area. The corresponding tcVISION DBMS extension emits the data into the z/OS logstream.

The processing of logstreams can be restarted if the last processed data block is still available in the active logstream file or has not yet been deleted from the logstream and processing can continue with the next block.

A typical scenario can be a sending script which sends the data in the stage 0 format to a remote receiving script.

In this scenario, the restart point in the logstream data is always specified by the receiving script through the definition of a restart file (State_Save_File ...) from a previous processing.

The sending script does not use any start parameters and should not use a restart file (State_Save_File ...) because it is controlled from the receiving script like a server.

The data script processes the data directly from the logstream up to the most current data block. At this point, processing is either terminated or waits for new logstream data if INPUT_FLAGS_NRT_CONTINUE_EOF is active.

13 IDMS: Updates Received from a Target DBMS

The following chapter is based on these requirements:

- Updates from IDMS must be propagated into a target DBMS.
- Applications can perform updates in the target DBMS that also have to be propagated into the mainframe IDMS database.
- All tables in the target DBMS that are related to corresponding IDMS tables must have the additional fields DBKEY and also the field DBKEY_OWNER if an owner-member-relationship exists. These fields are required to recognize the correct relational connections when processing the IDMS journal data.

When updating the target IDMS, the problem of tracking DBKEY and DBKEY_OWNER fields in the target system with the IDMS values arises.

13.1 General Considerations to IDMS

tcVISION/tcSCRIPT processes IDMS changes in the internal LOG format. All changes provided by IDMS can be referred back to the original record ID. However, there is one important exception from this rule. IDMS internally manages the space of the databases. As a result of this space management, IDMS creates and maintains so-called SYSTEM RECORDS. Refer to the corresponding IDMS documentation to obtain detailed information about system records. System ID record S3 identifies a user record as relocated. A typical cause for this can be a dynamic change of the database layout (RESTRUCTURE SEGMENT). S3 records that are part of an IDMS log do NOT contain any references to the original record ID.

It is strongly recommended to ensure that NO S3 records exist in a database BEFORE the actual data synchronization process starts. This can be achieved by UNLOADing and RELOADing the database. It should become part of the operational procedures that the database must be UNLOADed and RELOADed whenever the structure of a database has been changed.

If an UNLOAD/RELOAD function of a database has been performed, all internal DBKEYs have been changed. To keep the IDMS database in sync with the target database, the initial load (BULK) must be performed again.

13.2 INSERT Top-Level Table:

The record that has been inserted into the target system contains all fields with the exception of the DBKEY. The INSERT into IDMS will create a new internal IDMS DBKEY.

It is required that the new DBKEY is propagated back to the target DBMS, hence changing the INSERT to an UPDATE. This processing is fundamental and is called BackUpdateCheck. The conditions required to perform that check are defined in the repository.

For further information refer to the tips & hints document "IDMS Loopback Processing".

UPDATE Table SET DBKEY = value WHERE UniqueKey = value

Requirements:

The table MUST have a Unique Key

13.3 INSERT Non-Top-Level Table, MANDATORY and AUTOMATIC

The record inserted into the target system contains all fields except DBKEY and DBKEY_OWNER. The INSERT into IDMS results in a new internal DBKEY and at the same time the corresponding DBKEY_OWNER.

It is required that the new DBKEY and DBKEY_OWNER is propagated back to the target DBMS, hence changing the INSERT to an UPDATE. This processing is fundamental and is called BackUpdateCheck. The conditions required to perform that check are defined in the repository.

For further information refer to the tips & hints document "IDMS Loopback Processing".

The WHERE condition of the internal UPDATE command contains all fields of the Unique Keys of the table that can be obtained from the IDMS INSERT.
The fields of the Unique Keys that belong to the owner tables can be obtained from the target system using the repository value "Key-SQL". This is sufficient to uniquely identify the target record.
These tables may contain fields as Unique Keys that belong to tables that are higher in the hierarchy (owner(s)).

UPDATE Table SET DBKEY = value, DBKEY_OWNER = value WHERE UniqueKey = value

Requirements:
The table MUST have a unique key. The fields that build the unique key can also belong to owner tables. If the table itself has a unique key, this should always be used.

13.4 INSERT Non-Top-Level Table, OPTIONAL and MANUAL

The record inserted into the target system contains all fields except for DBKEY and DBKEY_OWNER. The INSERT into IDMS results in a new internal DBKEY and the corresponding DBKEY_OWNER will be set to -1.

It is required that the new DBKEY is propagated back to the target DBMS, hence changing the INSERT to an UPDATE. This processing is fundamental and is called BackUpdateCheck. The conditions required to perform that check are defined in the repository.

For further information refer to the tips & hints document "IDMS Loopback Processing".

The WHERE condition of the internal UPDATE command contains all fields of the Unique Keys of the table that can be obtained from the IDMS INSERT.
These tables MUST NOT contain fields as part of the unique key that belong to owner tables.

UPDATE Table SET DBKEY = value, DBKEY_OWNER = -1 WHERE UniqueKey = value

If a CONNECT is performed later in IDMS, the journal or DBMS extension exit will simply change the DBKEY_OWNER from -1 to the correct owner DBKEY.
Using this journal or DMBS extension exit, the data script uses the DBKEY_OWNER field and the repository value "Key-SQL" to obtain the field values from the target table and create the correct UPDATE statement to apply all necessary changes to the target table.

UPDATE Table SET DBKEY_OWNER = value, fields of owner tables = value... WHERE UniqueKey = value

Requirements:
The table must have a unique key. The fields that build the unique key must be fields from the table.

Fields from owner tables MUST NOT be defined with NOT-NULL.

13.5 UPDATE Top-Level Table:

The unique key of the record in the target table MUST NOT be changed.
The UPDATE into IDMS changes the desired fields except for the key fields. This UPDATE will be processed by a data script attached to a journal or DBMS extension exit. The LUW Manager flag 'IDMSBackUpdate' deletes the UPDATE because it has already been performed in the target system.

Requirements:
None

13.6 UPDATE Non-Top-Level Table:

The unique key fields and the fields of the owner tables of the record in the target system MUST NOT be changed.
The UPDATE into IDMS changes the desired fields except for the key fields. This UPDATE will be processed by a data script attached to a journal or DBMS extension exit. The LUW Manager flag ‚IDMSBackUpdate' deletes the UPDATE because it has already been performed in the target system.

Requirements:
None

13.7 DELETE:

The DELETE into IDMS physically deletes the record. This DELETE will be processed by a data script attached to a journal or DBMS extension exit. The LUW Manager flag ‚IDMSBackUpdate' deletes the DELETE because it has already been performed in the target system.
The sequence of the DELETEs in IDMS must be considered (first delete records in dependent tables, then the records in the corresponding top-level tables. It may also be possible to only delete the top-level record and instruct IDMS to internally remove all depending records).

Requirements:
None

13.8 Further Notes

In the target system, all fields of a record for level n that belong to owner tables of higher levels 1 to n – 1 will be directly obtained from the table of the next higher level
n -1. The consequence is that on level n fields from levels 1 to n – 2 must be defined in the next higher table of level n -1.

Example:

Level 1: Tab1 (Field11, Field12, Feld13)
Level 2: Tab2 (Field11, Field21, Field22, Field23)
Level 3: Tab3 (Field11, Field21, Field31, Field32, Field33)

Tab3 can use Field11, because it has also been defined in Tab2. Field 12 cannot be used, because it is not defined in Tab2.

14 Pipes, Usage and Implementation

Pipes are storage targets that hold internal data of a tcVISION data script. At the same time, these pipes can be input to several subsequent tcVISION data scripts.

A pipe can only be created by a single tcVISION data script, but can be processed as input from multiple tcVISION data scripts.

14.1 Restart Function

The write process (creation) is independent from the read process and has its own restart files. In case of a restart, the input source of the write process can correctly continue processing.

The read processes can also use restart files to continue processing from the correct restart point.

14.2 Deleting Expired Files in the Pipe

If read processes use restart files, those restart files are written in the directory of the feedback files per read process. With this information, the write process can ensure that the file that is required for the restart of read processes is not prematurely deleted because it expired.

Through the flag INPUT_FLAGS_DELET_INPUT_RC0, read processes signal the write process that files are not required in the pipe anymore if they are older than the current restart timestamp. If this flag is not set, the write process will receive a NULL restart timestamp and will therefore not delete any expired files in the pipe.

With the parameter Output_Pipe_Expiration_Min, old files in the pipe can be automatically deleted by the write process and the freed space is reused for new files. This processing is only activated if no read processes with feedback files control the deletion of data that is not required anymore.

14.3 Implementation

On Linux/Unix/Windows platforms, pipes are implemented through a control file and a dynamic set of files. On the z/OS platform, the implementation is based on logstreams.

On Linux/Unix/Windows platforms, files are dynamically created in the directory of the control file. The file names are *'name of the control file'.Pnnnnnnn,* with *nnnnnnn* being a running counter from 1 to the maximum number of files (9999999), then the counter is reset to 1.

A Pipe can logically fill up if the volume of the data that should be stored and is not yet expired is larger than the number of files in Output_Pipe_Max_File_Nr multiplied with the size specified in Output_Pipe_File_Size. In this case, the write process can be restarted with a higher value.

14.4 Skipping a Transaction of a Pipe

While an incorrectly processed transaction can simply be removed by moving it to a different directory during processing with separate staging files, this cannot be done when using Pipes. There is an additional mechanism that enables the removal for later evaluation:
The terminated script is started with the parameter LUW_TO_FILE=1 or LUW_TO_FILE=2. This can be done

a) via the command line, or
b) via the dashboard through pressing the “Ctrl“ key at system start, or
c) via Process parameter PIPE_LUW_TO_FILE

This parameter causes the transaction to be written as S1 file into the directory specified as “WorkDir”. The name is made up of managername, scriptname, timestamp, luwid, „.debug.bin“. With this, the transaction is skipped and the restart is delayed to the following transaction.The script terminates itself and can then be started normally (without parameters).
With LUW_TO_FILE=2, the transaction is not skipped and the restart stays unchanged.

15 Parrallel Apply

Parallel Apply is a procedure that allows CDC LUWs and bulk data to be applied in parallel into a target database to improve throughput.

1.) CDC LUWs:
Caution:
Using this procedure can no longer ensure the referential integrity of the data in the target database. All LUWs are applied correctly, but the chronological order in which the LUWs are applied is no longer guaranteed.

The procedure consists of a CDC log reader script, which writes the log data in stage 1 with the LUW manager active into an output pipe.
The procedure for the CDC log reader script is activated with the parameter PARALLEL_APPLY_MAX_SLOTID = n. The LUWs are given a SlotID, which takes values from 1 to n.
With these 'marked' LUWs, several Apply scripts can now read out the pipe in parallel and, due to the SlotID, only apply selected LUWs into the target database.
The parameter PARALLEL_APPLY_CHECK_SLOTID determines which SlotIDs an Apply script should process. All other SlotIDs are skipped because they are processed by other Apply scripts.
It must be ensured that the various Apply scripts cover all SLOTIDs assigned by the CDC log reader script exactly once. Otherwise, LUWs that have not been selected will not be applied into the target database or LUWs that have been covered more than once will be applied into the target database more than once.

2.) Bulk data
The procedure consists of bulk apply scripts, which read bulk data directly from a source database backup (e.g. Db2 ICOPY etc.) or as stage 0 data from a tcVISION file or pipe.
The procedure for bulk apply scripts is activated with the parameter PARALLEL_APPLY_MAX_SLOTID = n. The bulk records are given a SlotID, which takes values from 1 to n.
With these 'marked' bulk records, each bulk apply script can now read out the bulk data in parallel and, based on the SlotID, only apply selected bulk records into the target database.
The parameter PARALLEL_APPLY_CHECK_SLOTID determines which SlotIDs a bulk apply script should select. All other SlotIDs are skipped because they are processed by the other bulk apply scripts.
It must be ensured that the various bulk apply scripts all use the same value n for the parameter PARALLEL_APPLY_MAX_SLOTID = n and cover all assigned SLOTIDs exactly once. Otherwise, unselected bulk records will not be applied into the target database, or bulk records that have been covered several times will be applied into the target database more than once.

16 HDFS: Requirements

tcSCRIPT is capable of writing or reading data directly into or from an HDFS.
This requires either the libhdfs API or the libhdfs3 API.

The libhdfs API is part of the Apache-Hadoop distribution.

If this solution is used, it is essential that the required jars are part of the system on which tcSCRIPT is executed and the CLASSPATH has been set accordingly.

To provide the required jars for the current Hadoop distribution, the following command is available:

hadoop classpath

It is important to note that NO wildcards are allowed in CLASSPATH. All jars must be specified individually.

Documentation concerning this command can be obtained from:
http://hadoop.apache.org/docs/r2.7.0/hadoop-project-dist/hadoop-common/CommandsManual.html#classpath

If jars are missing in CLASSPATH, the script will be cancelled.

Another possibility is to use the libhdfs3 API. This API can be located at:

https://github.com/ContinuumIO/libhdfs3-downstream/tree/master/libhdfs3

The advantage of this API is the fact that it has been developed completely in C and does not use JNI. This results in a faster performance for the write processes to HFDS. No CLASSPATH is required.

However, the disadvantage of this API is the fact that there is no functional version for Windows available.

Both APIs support a remote file access. It is not necessary to install tcSCRIPT on the main node. However, the remote write access leads to a higher time expenditure.

17 Mainframe: Internal Invocation of DFSORT

The batch compare function supports the processing of unsorted input files like IMS/DLI HDAM, sequential files, VSAM ESDS, etc., and sorts the data before the actual compare processing takes place.

The internal sort is activated through flag 'INPUT_FLAGS_DATA_SORT'.

The sorting can be performed in data scripts running under the control of an Agent as well as in external scripts.

The sorting is performed in the script by calling the DFSORT modules and using the DFSORT exits E15 and E35.
The required RECORD and SORT parameters are dynamically created by the script, but can be overwritten using script variables.
In addition, script variables can be used to supply 'OPTION' statements to DFSORT.

Additional JCL statements (SORTWK... etc) for DFSORT can be supplied as part of the execution JCL.

The following script variables can be specified when using DFSORT:

SORT.SORT = ' SORT statement '
Using this variable will overwrite the internally generated SORT statements.

SORT.RECORD = ' RECORD statement '
Using this variable will overwrite the internally generated RECORD statements.

SORT.OPTION = ' OPTION statement '
Using this variable will pass the supplied OPTION statement to DFSORT.

18 Copyright

License information on third-party components used can be found in the manual ‘Copyright’.