DB2 Performance Monitor for OS/390

Data Collector Application Programming Interface Guide

Version 6

SC26-9173-01

DB2 Performance Monitor for OS/390

Data Collector Application Programming Interface Guide

Version 6

SC26-9173-01

Note

Before using this information and the product it supports, be sure to read the information in "Appendix E. Notices" on page 135.

Second Edition, June 2000

This edition applies to Version 6 of IBM® DATABASE 2™ Performance Monitor for OS/390®, a feature of IBM DATABASE 2 Universal Database Server for OS/390 Version 6 (5645-DB2), and to all subsequent releases and modifications until otherwise indicated in subsequent editions.

Contents

Preface

This book describes the Data Collector Application Programming Interface of IBM DATABASE 2 Performance Monitor for OS/390 Version 6 and its use.

IBM DATABASE 2 Performance Monitor for OS/390 (DB2 PM) is the performance analysis tool to monitor and tune DB2® systems and DB2 applications. It is one of the optional features that complement IBM DATABASE 2 for OS/390 (DB2).

DB2 PM provides the host processor-based MVS™ ISPF Online Monitor and the Workstation Online Monitor to interactively monitor DB2 performance, and to execute DB2 commands. DB2 PM also provides a batch facility to generate performance reports.

New with DB2 PM Version 6 (as program temporary fix) is the DB2 PM Data Collector Application Programming Interface (API). It provides a programming interface to DB2 performance data. Workstation-based application programs can use this interface to connect to the host-resident DB2 PM Data Collector through TCP/IP to access DB2 performance data in real time or from the Data Collector's own history data set.

The DB2 PM Data Collector API allows programmers to develop their own customized application programs.

All DB2 PM Data Collector API functions can be accessed using calls from C or C++ programs or any other programming language that supports C function calls.

Who Should Read This Book

This book is for system and application programmers who want to design and implement workstation application programs that take advantage of the API to gain access to DB2 performance data.

It is assumed that the reader is familiar with the DB2 PM Online Monitor functions and the C programming language.

How to Use This Book

Read "Chapter 1. Overview of Elements and Concepts" on page 1 to gain an understanding of the elements and concepts of the DB2 PM API and how these elements interact with each other. This knowledge helps to decide whether the use of the API is adequate to solve your task. This knowledge is also required to effectively use the API functions.

"Chapter 3. The Data Collector API Functions" on page 15 describes the Data Collector API functions, its calling conventions, and its parameters, and gives code examples for an application program. Functions with similar objectives are grouped together; every group begins with a comprehensive description of the concepts of which you should be aware. Use this information when you apply these functions to your application program.

Availability of the Data Collector API

The DB2 PM Data Collector API is available for DB2 PM Version 6 as PTF, as a recommended maintenance level for Version 6. Future versions of DB2 PM might provide enhanced versions of the API as an integral part of DB2 PM.

Applicability of the Data Collector API

The DB2 PM Data Collector API is applicable with:

- IBM DATABASE 2 Universal Database Server for OS/390 (DB2 UDB for OS/390) Version 6, program number 5645-DB2
- IBM DATABASE 2 Server for OS/390 (DB2 for OS/390) Version 5, program number 5655-DB2
- IBM DATABASE 2 for MVS/ESA (DB2 MVS/ESA) Version 4, program number 5695-DB2

or any higher version.

The API data sharing group support requires at least DB2 Version 6.

DB2 PM provides the necessary workstation development kit and workstation run-time environment as either Dynamic Link Library or Shared Library for the following operating system platforms:

Table 1. Supported Operating Systems

Operating System	**Library**
IBM OS/390 Version 2.7, or higher	Shared library
IBM AIX® Version 4.3, or higher	Shared library
Microsoft® Windows® NT 4.0	Dynamic link library
Sun Solaris 5.6, or higher (32 bit) Sun Solaris 5.7, or higher (64 bit)	Shared library Shared library
Linux Version 2.2, or higher, on Intel x86	Shared library

DB2 PM Installation and Customization describes how to install the DB2 PM API on a workstation. You may want to copy or use the DB2 PM API as often as required, provided you hold a valid DB2 PM license.

The workstation requires TCP/IP to be installed, either as part of the operating system, or as a separate licensed program. The Data Collector requires IBM TCP/IP Version 3 Release 2 for MVS/ESA to be installed on the host processor. These requirements are specific to the use and operation of the DB2 PM API. The general prerequisites for DB2 PM are to be applied.

Summary of Changes

- DB2 PM Version 6, APAR PQ38542, results in the following changes and enhancements:
  - The following functions are replaced by extended functions that support performance monitoring in data sharing environments:

Old:	**New:**
pmInitializeStore()	**pmInitializeStoreEx()**

Old:	New:
pmGetSnapshot()	**pmGetSnapshotEx()**
pmReset()	**pmResetEx()**
pmExecDB2Command()	**pmExecDB2CommandEx()**

The API continues to support the replaced functions, however, they should not be used for new developments.

– Support for Unicode-enabled workstations is added. The API can also accept input data, and returns data, in the UCS-2[1] encoding form.

1. Universal Character Set coded in two octets.

Chapter 1. Overview of Elements and Concepts

A DB2 subsystem running on the OS/390 operating system generates and collects data about its own performance, but does not provide facilities to evaluate and report about its performance. Companies with a distinctive need to evaluate the performance of DB2 can use the DB2 Performance Monitor (DB2 PM) feature. DB2 PM provides interactive and batch-oriented facilities to monitor the DB2 performance and generate reports about it.

However, sometimes you might want to use customized applications that are external to DB2 and DB2 PM and which need access to DB2 performance data. For example, a company uses several database systems from different suppliers. Every of these database systems has an own user interface to performance-related data, which generates increased complexity and requires specialized skill. The solution is an application program that monitors performance aspects of all these database systems and provides the results in a common user interface to your support personal. Then, the application would need real-time access to performance data of every of these database systems.

The DB2 PM Data Collector Application Programming Interface (API) provides a structured set of functions that you can use in an application program to process DB2 performance data in real time. Thus, the API does not require the DB2 PM Online Monitor or DB2 PM Batch.

The use of the API at application development time and at application run time involves several elements of DB2 PM and DB2 as well as connectivity aspects. The remaining part of this chapter gives an overview of the involved elements, explains the user and connection concept, and describes how they work together.

The DB2 PM Data Collector

The DB2 PM Data Collector is the host component of the DB2 PM API. It provides an access path from a client application program to DB2 internal performance data.

The Data Collector resides in an OS/390 address space. It needs to run as an OS/390 started task before any client application can connect to it. Once started, the Data Collector takes the role of a server.

Upon startup of the Data Collector, TCP/IP is also started on the host. This allows client application programs to connect to the Data Collector through TCP/IP when required. The Data Collector is capable of serving up to 500 logged-on applications, respectively users, concurrently.

After an application has established a TCP/IP connection to the Data Collector, the Data Collector is ready to accept a logon request from a client application. During logon the application identifies itself to the Data Collector.

The Data Collector does not require an application to remain connected after a successful logon. Users can disconnect from and reconnect to the Data Collector from varying places at any time and still remain logged on.

The Workstation Application

The application can reside on workstations that run one of the supported operating systems listed in Table 1 on page vi. You need the corresponding DB2 PM API installed on these workstations to develop, test, and run an application. The DB2 PM API support contains all files and libraries required to use the API.

The application can be written in any programming language that supports the C language calling conventions required to call the API functions.

The application on a workstation communicates with the Data Collector by using TCP/IP. The workstation needs to have TCP/IP support available either as part of the operating system or as a separate licensed program. TCP/IP is not apparent to the application. When it connects to the Data Collector, by calling the corresponding API function, the underlying operating system services are used to establish a TCP/IP connection to the Data Collector.

The application controls the connection to the Data Collector. After a successful connection and a successful logon, a user session is established. During this session the application calls any number of API functions to describe the data it wants to access or to be returned. At the end it closes the user session with the Data Collector and terminates the connection to it. This frees all Data Collector resources that the application has occupied.

The Data Collector API Functions

The DB2 PM API gives an application program access to DB2 performance data. Hence, the majority of all API functions is related to DB2 performance data. In addition, an API function is available to execute DB2 commands. The remaining functions maintain TCP/IP connections and user sessions between the application and the Data Collector.

An application using the API has access to the following types of data and can perform the following functions:

- Access to DB2 performance data

  The application can request statistics information on a DB2 subsystem level and on a thread level. It can query DB2 system parameters, for example, installation parameters, or VSAM catalog information.
- Access to DB2 event exceptions

  The Data Collector logs the most recent DB2 exceptions in an exception log. The application can retrieve up to 500 of these recent exception log records and process them individually. The DB2 PM API also supports a synchronous notification of DB2 exceptions. Once set up, the application is notified about DB2 event exceptions when they occur.
- Execution of DB2 commands

  The DB2 PM API supports the execution of DB2 commands from within the application. The Data Collector transfers the commands to DB2 and delivers the response data to the application. Provided that a logged-on user has sufficient DB2 privileges, all DB2 commands can be executed, except the application cannot start or stop a DB2 subsystem.
- Sysplex-wide access to DB2

  If the Data Collector belongs to a DB2 subsystem that is part of a data sharing group in a Sysplex, the application can have group-wide access to DB2 performance data.

- Access to user data in the Data Collector

  The application can save user-specific data in the Data Collector and retrieve this data at any point in time. This allows you to develop applications that support mobile users. Users of the application can disconnect from the Data Collector (while remaining logged on) at one location and reconnect to it from another location. The Data Collector serves as an interim storage for user-specific data that need to be kept in the meantime.
- Parser functions are available for extracting relevant data from Data Collector responses. These functions are helpful in cases where API functions return complex and varying data structures. The parsing functions also check returned data streams for correctness, and convert data between the host processor format and the workstation format.
- Time conversion functions convert host date and time formats and workstation date and time formats. Arithmetic functions simplify time zone adjustments.

DB2 performance data that is accessible through the Data Collector API is raw DB2 instrumentation data and data that the Data Collector derives from raw DB2 instrumentation data. The information about the data fields is provided in a separate text file that accompanies the DB2 PM API. "Appendix B. Field Table Summary" on page 121 describes how the data in the flat file is organized and shows a short sample of the field table.

"Getting DB2 Performance Data" on page 27 gives a comprehensive introduction to the Data Collector counter concepts, snapshot and history processing, the snapshot store concept, and shows how data can be filtered to reduce the amount of available performance data. The knowledge about these concepts is required to effectively use the API functions that deal with performance data.

Note that the Data Collector can be customized during its installation and DB2 PM user exit routines may be active. User exit routines allow user-written routines to perform customized processing. This may limit the access of the application to some counters, regardless of the fact that they are listed in the field table.

The Connection Concept

The communication between the application and the Data Collector happens through TCP/IP. The application uses the **pmConnect()** function to open a TCP/IP connection to the Data Collector. Several connections can be opened, if required. The maximum number of connections is specified as a start parameter of the Data Collector itself.

Once a communication path to the Data Collector is established, it is uniquely identified by an identifier, called a *handle*. All subsequent API functions use this handle to identify the physical connection path to the Data Collector they want to use. If several connections are opened, different handles are used to distinguish among the connections.

Users logging on to the Data Collector are identified by a *work profile*. All subsequent API functions use this work profile to assign a function to a user.

This way, connections to the Data Collector are completely separated from users of the Data Collector. The application can use every opened TCP/IP connection to issue requests to the Data Collector for every user. Vice versa, the application can use a single TCP/IP connection also for several users.

Connections to the Data Collector are required when the application actually needs to communicate with it. Users who are already logged on to the Data Collector remain logged on even when no connection is opened. This allows you to write applications where users can disconnect at one work place (the application calls the **pmDisconnect()** function) and reconnect from another work place (the application calls the **pmConnect()** function).

The User Concept

The workstation application must identify its users to successfully log on to the Data Collector. The Data Collector, when receiving a logon request, always uses the System Authorization Facility (SAF), and the IBM Resource Access Control Facility (RACF) or any other security system, to verify a user's authorization.

The terms "SAF user" and "SAF group" are used in this book in accordance with their definitions in the RACF® literature. See the appropriate manuals, if required.

The DB2 PM API requires you to define a "user" in terms of a work profile. A work profile is a character string that contains the following information:

- SAF user ID

  The user ID of a DB2 PM user as known to RACF. This information is always required to identify a user logging on to the Data Collector.
- SAF group ID

  The group identifier an SAF user is associated to in RACF. This field is optional, but if it is used it must specify a valid SAF group ID.
- Profile ID

  An optional field that the application can use as required. For example, you can use it to specify subusers of an SAF user ID. To simplify matters in this book, this field is used to specify a subuser.

Once the application has specified a user by means of a work profile, this information is used as parameter **workProfile** of the **pmLogon()** function. The Data Collector manages the authorization of the SAF user ID and, if specified, the SAF group ID.

After a user is authorized to request services from the Data Collector, all subsequent requests are identified by a user's work profile. For security reasons, every combination of an SAF user ID and an SAF group ID requires authorization.

Subusers, if specified in the Profile ID field, do not need separate authorization. Upon proper authorization of an SAF user ID and SAF group ID combination, all subusers are implicitly authorized.

Every user specified by a unique work profile must log on and off individually. This ensures that all resources used by a user in the Data Collector are released when the user logs off.

The combination of the connection concept and the user concept ensures that every request to the Data Collector is identified by:

- A handle, which specifies the connection path
- A work profile, which specifies a user.

To preserve security, the application must request authentification of a user on every connection path the user wants to use.

The Security Concept

The DB2 PM Data Collector is one of many OS/390 system resources that client applications might want to access. Access to OS/390 system resources is usually controlled by an OS/390 security server, for example, by RACF. This requires that every potential application and every potential user is known by RACF. Access to a resource is given if an application or user is successfully identified by the security server and sufficient authority is given.

Further, the communication between the application and the Data Collector happens through TCP/IP, which does not provide any security mechanism that prevents hostile intrusion. All data is transferred as plain text. Everyone who knows the IP address and port number can connect to the Data Collector.

The Data Collector API supports two levels of security implementations. Both levels build on how the application logs on to the Data Collector. Once the application has successfully logged on, all further API function calls are bound to this logon.

Logon with a Password

The application is granted access to the Data Collector when it identifies itself by a correct user identification and password. The **pmLogon()** function passes both parameters, the user ID and the password, over to the Data Collector for authorization. The Data Collector then requests the OS/390 Security Server (RACF) to verify the user identity. If the authorization succeeds, a user is successfully logged on, and the application can call further API functions as required.

This method provides only limited protection. The password is transmitted as plain text to the Data Collector. You should only consider this method if the application runs inside a secure and protected TCP/IP network.

Logon with a PassTicket

For enhanced security you can implement a logon process that uses an encrypted password, called a *PassTicket* (a RACF term). A PassTicket has two properties that make it difficult to be misused:

1. It remains valid for only 10 minutes. Once you have generated a PassTicket, it should be used as parameter of the **pmLogon()** function within the next 10 minutes, otherwise it becomes unusable.
2. It can be used only once. For every logon the application must generate a new PassTicket.

The **pmGenPassticket()** function generates a PassTicket. The algorithm to generate a PassTicket uses:

1. The hardware clock of the host processor where the Data Collector resides.
2. The user ID for which the application requests authorization by the OS/390 Security Server (RACF).
3. A fixed application name and a secure signon key that are known in the OS/390 Security Server.

   A copy of the secure signon key must reside on the workstation to enable the **pmGenPassticket()** function to access it. You should use any of the operating system functions to protect or hide it, or use an access control device connected to the workstation (like a smart card reader) to make it available during the logon process.

After the **pmGenPassticket()** function has generated a PassTicket, the application uses it as a logon parameter of the **pmLogon()** function.

How the Components Interact

This section briefly describes scenarios how an application program interacts with the Data Collector and which API functions are involved in this process. The timely order is pointed out, and the previously introduced components and their actions are explained.

- The first scenario describes how an application program establishes and terminates a user session with the Data Collector.
- The second scenario describes how an application program disconnects and reconnects a mobile user and maintains the user session with the Data Collector.

  This scenario deploys the independence between connections and user sessions.

For these scenarios it is assumed that the Data Collector is started and can be reached by client applications through a TCP/IP connection. If it is not started, any attempt to connect to it fails. Further, it is assumed that the workstation is properly set up and that its TCP/IP services are available.

Establishing and Terminating a User Session

1. The application must establish a TCP/IP connection to the Data Collector before any other API function can be called. To do this, the application calls the **pmConnect()** function and passes along some parameters, for example, the TCP/IP port where the Data Collector is listening, and the host name where DB2 is installed. Assumed the Data Collector can be reached successfully and the host name can be resolved to a valid IP address, a TCP/IP connection between the application and the Data Collector is then opened.
2. The API returns a unique *handle*, which identifies this connection path. Subsequent requests to the Data Collector use this handle to identify the communication path.

   If the Data Collector or DB2 is not available, the **pmConnect()** function returns the appropriate return code, which the application can evaluate.
3. Next, the application program needs to log on. The Data Collector authenticates the application and verifies the authorization of the user, or group, or subuser, logging on. The logon process passes security-sensitive information to the Data Collector. Therefore, the API provides two alternatives:
   - The less secure logon calls the **pmLogon()** function and passes along parameters that identify the user or group or subuser, together with a password.
   - The secure logon generates an encrypted password first, using the **pmGenPassticket()** function. Thereafter, the **pmLogon()** function uses the encrypted password as a logon parameter.

   The work profile that was used to log on to the Data Collector is used for all subsequent function calls to associate Data Collector requests with a user.
4. After a successful logon, it is recommended that the application gets some information about the Data Collector to ensure that it is functionally compatible with the Data Collector version installed on the host processor. It calls the **pmGetInfo()** function and performs the necessary checks.

   A user session is now established. The application can proceed according to its purpose.

5. If no more tasks are to be executed, the application starts the termination step. It calls the **pmLogoff()** function to log off from the Data Collector, which releases all snapshot stores in the Data Collector and frees all resources.
6. The application drops the TCP/IP connection to the Data Collector with a **pmDisconnect()** function call.

Disconnecting and Reconnecting while Preserving a User Session

1. An application is already connected to the Data Collector; the connection is identified by a handle. A user is logged on and identified by a work profile. The application continuously monitors a DB2 process. When the application recognizes a user interaction that shows that the user wants to disconnect from the Data Collector but wants to remain logged on, it stops gathering and monitoring data.
2. The application calls the **pmSaveUserData()** function to save user-specific data in the Data Collector. This data can include everything that is required for a later reconnect to the Data Collector, for example, the current workstation settings, or parameters the application was using while monitoring the DB2 process.

   If the application runs on a portable PC, it could also save this data on the PC's local disk.
3. The application calls the **pmDisconnect()** function, with the previously used handle as parameter. This drops the specified TCP/IP connection to the Data Collector.
4. The application can now close down. The connection is dropped. The user is still logged on to the Data Collector.
5. When the application is started again (from another workstation), and the user wants to continue monitoring the DB2 process, the application reconnects to the Data Collector with the **pmConnect()** function. The new connection is identified by a different handle.
6. Even so the user is still logged on to the Data Collector, the application now has to issue a new **pmLogon()** function call. This allows the Data Collector to authenticate the application and to verify the authorization of the user logging on. Because the user is still logged on, the Data Collector returns a warning that the user is already logged on — which is a normal response here.
7. After the logon, the application retrieves the previously saved user-specific data from the Data Collector with the **pmGetUserData()** function. It uses this data to reconstruct the previous state, for example, the workstation's settings, or the parameters used.
8. The application is now ready to continue the monitoring process. Subsequent function calls to the Data Collector use the new handle to identify the connection path, and the same work profile to identify the user.

Chapter 2. Considerations for Using the Data Collector API

The DB2 PM API is delivered as a set of C programming language functions, which can be called from the application program you are writing. This book assumes that you write an application in the C programming language. For this purpose the API is delivered as a dynamic link library or shared library, together with the header files, for the operating systems listed in Table 1 on page vi.

If you write an application in a programming language other than C, consult the appropriate programming manuals about making mixed-language calls.

The DB2 PM API is delivered together with header files that contain the necessary declarations for all functions that make up the API. "Chapter 3. The Data Collector API Functions" on page 15 provides the relevant header file name together with every function description.

In "Chapter 3. The Data Collector API Functions" on page 15 parameters are either marked as input or as output parameters. Input parameter values are send to the Data Collector; output parameter values are returned by the Data Collector to the application in response to a function call.

Return codes and reason codes that are issued by the API functions are described in "Appendix A. Return Codes and Reason Codes" on page 103.

Note that all API function names are case-sensitive.

Workstation Memory Handling

Whenever the application program calls an API function, the Data Collector returns data to it. Most functions allocate the necessary data space in the workstation's memory to store the returned data. The Data Collector itself does not provide data space that you can use for this purpose. An exception is the 1-MB buffer in the Data Collector that is intended to store user-specific data. See "Saving and Retrieving User Data" on page 80 for details.

The application is responsible to release allocated memory when the data space is no longer needed. The API provides two functions to release memory when required: the **pmFreeMem()** function, and the **freeHashTable()** function, which releases memory that is allocated to store data of variable length in hash tables.

Code Page Conversions

The Data Collector and the workstation applications may use different standards to represent data. The Data Collector uses EBCDIC (Extended Binary-Coded Decimal Interchange Code). Most workstation applications usually use ASCII (American National Standard Code for Information Interchange); however, some use EBCDIC.

Further, both sides may use different code pages to accommodate for different national languages. Code pages specify how the EBCDIC and ASCII codes are presented for a specific language.

When the application exchanges data with the Data Collector, it automatically converts the data between ASCII and EBCDIC (if the workstation uses ASCII), and

it converts text data according to the code pages in use. This conversion is transparent to the application, however, the code page used on the workstation must be specified with the **pmConnect()** function.[2]

Considerations for Unicode-Enabled Workstations

If the application runs on a Unicode-enabled workstation, the API can convert input data from the UCS-2[3] encoding form to EBCDIC (the host character set) and output data from EBCDIC to UCS-2. As with code pages, the conversion is transparent to the application, however, you must specify this type of conversion with the **pmConnect()** function. Instead of a code page you specify the constant `PM_UCS2_CP`.

If you write applications that run on Unicode-enabled workstations, the following remarks apply to all API functions:

- The API's conversion capability does not cover the full range of UCS-2, but is limited to data that is usually exchanged with the Data Collector (EBCDIC character set). If UCS-2 characters are specified as input strings that cannot be converted, the appropriate function calls return an error condition.
- The decision whether a workstation's code page or Unicode is used for character conversions is based on the **codePage** parameter of the **pmConnect()** function. Once a connection is made to the Data Collector, all further function calls that use the same connection will execute the same conversion. The connection is identified by parameter **handle**, which is returned by the **pmConnect()** function call.
- If an application uses multiple connections and serves different workstations with individual code pages as well as Unicode (for example, a client/server application), or if different applications use the API at the same time, each **pmConnect()** function call can be associated with a specific code page, or Unicode.
- API functions that request data from the Data Collector are tied to a specific connection through parameter **handle**. However, several functions that perform post-processing of returned data without involving the Data Collector are not bound to a specific connection (they do not have a parameter **handle**). An example are hash table functions that let you extract details from hash tables. They work exclusively on the workstation and do not communicate with the Data Collector.

  For these functions it is the application's responsibility to maintain the relation between a connection (and its underlying character conversion) and data that is not yet finally processed. For example, if an application maintains two connections with different character conversions, it should not receive data through connection 1, process it, and use part of the outcome as input to connection 2.
- Several API functions transfer data to and from the Data Collector intentionally without any character conversion, even if the data is supplied as character strings. Examples are the **pmSaveUserData()** and **pmGetUserData()** functions, which use the Data Collector as an intermediate storage for user- or application-specific data, and several parameters, which are generated by one function and used by another function without change.

2. Internally, the **pmConnect()** function sends the workstation's code page number to the Data Collector. The Data Collector, which "knows" already the host's code page number, builds a conversion table from both code pages, which is then returned to the workstation. The conversion table is stored internally and ensures correct conversions of outgoing and incoming data.

3. Universal Character Set coded in two octets.

You find respective hints together with the individual function descriptions.

- With several functions you need to specify the length of data blocks to work with. The length is always expressed in the number of bytes, also for UCS-2 encoded strings (2-byte Unicode). For example, a UCS-2 string of eight characters has a length of 16 bytes.

  You find respective hints together with the individual function descriptions.

Preparations for Using PassTickets

If the application uses the **pmGenPassticket()** function to generate encrypted passwords, ensure that the OS/390 Security Server (RACF) is properly prepared.

Log on to the OS/390 system as a TSO/E user and execute the following RACF commands to create a profile for DB2 PM. You may need sufficient authority to issue these commands.

1. **SETROPTS CLASSACT(PTKTDATA)**
2. **SETROPTS RACLIST(PTKTDATA)**
3. **RDEFINE PTKTDATA** *profile_name* **SSIGNON(KEYMASKED(***key_value***)) UACC(NONE)**

   This command defines an application name to the OS/390 Security server (RACF) and associates a secure signon key with the name of the application.

   A PTKTDATA class *profile_name* must be specified as an application name, optionally qualified by a RACF user ID. The application name must be `MVSDB2PM` for DB2 PM. You should concatenate the application name with a RACF user ID, like in `MVSDB2PM.PMUSER`. If only the application name is used, all user IDs can log on using a PassTicket generated with the secure signon key specified in *key_value*. This is usually a security exposure and should be avoided.

   key_value is 16 characters long and can contain numeric (0-9) and alphabetic (A-Z) characters.

After you have set up the profile for DB2 PM, the application can call the **pmGenPassticket()** function to generate a PassTicket. **pmGenPassticket()** uses `MVSDB2PM` as its **application** parameter, and *key_value* as its **secureSignonKey** parameter. See "Generate RACF PassTicket" on page 19 for more details.

Compiler Considerations

To compile your C program use any C or C++ compiler compatible to the operating system platform on which you are working. For example, you can use IBM VisualAge® C++ for Windows NT, or Microsoft Visual C++.

For IBM VisualAge C++ use compiler option `/Su4` to use four bytes for enumerations instead of the `SAA` default setting.

For Visual C++ 6.0 use the default compiler settings.

Linking the DB2 PM API Library on Windows NT

If you plan to run the application on Windows NT, it is highly recommended that you resolve entry points in the DGOKAPI.DLL library dynamically. This prevents problems that might occur with some compilers.

This is not required for the other supported operating systems because their DLLs support dynamic entry point resolving.

The following example shows how to do this for Microsoft Windows NT 4.0:

```
/*********************************************************************/
/* Example how to resolve entry points of C-API functions dynamically */
/* on Windows NT 4.0.                                                */
/*********************************************************************/

#include <windows.h>
#include <stdio.h>
#include "pmConnect.h"
#include "pmGenPassticket.h"
#include "pmLogOnOff.h"

/* Type definitions for dynamically loaded functions used in this     */
/* program. See corresponding C-API header files for parameters.      */
typedef pmReturnCodes __cdecl ppmConnect(char *, char *, unsigned int, pmHost *);
typedef pmReturnCodes __cdecl ppmDisconnect(pmHost *);
typedef pmReturnCodes __cdecl ppmGenPassticket(pmHost *, char *, char *,
                                               char *, char *);
typedef pmReturnCodes __cdecl ppmLogon(pmHost *, char *, char *);
typedef pmReturnCodes __cdecl ppmLogoff(pmHost *, char *);

/* Name and handle of DB2 PM C-API DLL */
char              *dllName = "DGOKAPI.DLL";
HINSTANCE          dllHandle = NULL;

/* Function pointers for all dynamically loaded functions */
ppmConnect        *fnConnect = NULL;
ppmDisconnect     *fnDisconnect = NULL;
ppmGenPassticket  *fnGenPassticket = NULL;
ppmLogon          *fnLogon = NULL;
ppmLogoff         *fnLogoff = NULL;

/* Function prototypes for helper functions */
FARPROC  LoadFunction(char *funcName);
void     LoadAPI(void);
void     UnloadAPI(void);


int main(void)
{
  pmHost        myHandle;
  pmReturnCodes error;
  char          passticket[8];
  char          *workprofile = "PMUSER  GROUPID PROFILEID      TERMINALID     ";

  /* Load DB2 PM C-API functions */
  LoadAPI();

  /* Connect to the data collector at IP address 10.0.0.1 */
  /* at port 4711 using workstation codepage 850.         */
  error = (*fnConnect)("10.0.0.1", "4711", 850, &myHandle);

  /* Create a passticket for user PMUSER for application   */
  /* MVSDB2PM using the secure signon key E001033FAF00007B */
  error = (*fnGenPassticket)(&myHandle, "PMUSER", "MVSDB2PM",
                             "E001033FAF00007B", passticket);

  /* Logon to DB2 PM Data Collector using the workprofile */
  error = (*fnLogon)(&myHandle, workprofile, passticket);

  /* Execute other DB2 PM commands */
  /* ... */
```

```
  /* Logoff from DB2 PM Data Collector */
  error = (*fnLogoff)(&myHandle, workprofile);

  /* Disconnect from Data Collector */
  error = (*fnDisconnect)(&myHandle);

  /* Free DB2PM C-API DLL */
  UnloadAPI();
  return(0);
}


/* Load the DB2 PM DLL if necessary and resolve a single */
/* entry point for the specified function name.          */
FARPROC LoadFunction(char *funcName)
{
  FARPROC   pfn=NULL;

  /* Load DLL if necessary */
  if (dllHandle == NULL)
    dllHandle = LoadLibrary(dllName);

  if (!dllHandle)
  {
    fprintf(stderr,
            "Error loading DB2 PM C-API DLL! Program terminated!\n");
    fflush(stderr);
    exit(-1);
  }
  else
  {
    /* Get entry point for the specified function */
    pfn = GetProcAddress(dllHandle, funcName);
    if (!pfn)
    {
      fprintf(stderr,
              "Error loading function %s from DB2 PM C-API DLL!\n",
              funcName);
      fprintf(stderr, "Program terminated!\n");
      fflush(stderr);
      exit(-1);
    }
  }

  return pfn;
}

/* Load all functions used in this program from DB2 PM C-API DLL */
void LoadAPI(void)
{
  fnConnect = (ppmConnect *) LoadFunction("pmConnect");
  fnDisconnect = (ppmDisconnect *) LoadFunction("pmDisconnect");
  fnGenPassticket = (ppmGenPassticket *) LoadFunction("pmGenPassticket");
  fnLogon = (ppmLogon *) LoadFunction("pmLogon");
  fnLogoff = (ppmLogoff *) LoadFunction("pmLogoff");
}

/* Unload DB2 PM C-API DLL */
void UnloadAPI(void)
{
  FreeLibrary(dllHandle);
}
```

Using the DB2 PM API Trace Facility

If you need to diagnose what happens when the application communicates with the Data Collector, you can activate the DB2 PM API trace facility. You can choose to let the trace facility record connection information, commands transferred, and data transferred. By default, the trace facility displays the required information. Alternatively, you can redirect this information to files for later analysis.

You activate the trace facility by adding the following environment variables to the operating system and by setting them to **ON**.

- The **PM_CONNECTION** environment variable controls tracing of connection information. By default, the trace data is written to STDOUT, which is usually the workstation's screen. If you want to capture the information in a file, add the environment variable **PM_CONNECTION_FILE** to the operating system and set its value to a file name.
- The **PM_COMMAND** environment variable controls tracing of DB2 PM commands. By default, the trace data is written to STDOUT, which is usually the workstation's screen. If you want to capture the information in a file, add the environment variable **PM_COMMAND_FILE** to the operating system and set its value to a file name.
- The **PM_DATA** environment variable controls tracing of DB2 PM API data that is transferred between the application and the Data Collector. By default, this data is written to STDERR, which is usually the workstation's screen. If you want to capture the information in a file, add the environment variable **PM_DATA_FILE** to the operating system and set its value to a file name. Note that this data is shown as binary data.

For information about adding, setting, resetting, or removing environment variables see the operating system manuals.

The output files created by the trace facility are not erased from the workstation's hard disk. Subsequent trace data is appended to existing files. Make sure that you remove the environment variables when no longer required.[4]

"Appendix C. Sample Traces" on page 125 shows a sample of a connection trace, a command trace, and a data trace.

4. In the current version of the DB2 PM API only the existence of the environment variables is checked.

Chapter 3. The Data Collector API Functions

Maintaining a TCP/IP Connection to the Data Collector

The DB2 PM API uses TCP/IP to communicate with the Data Collector. Your application needs a direct network connection to the Data Collector and a configured and running TCP/IP environment on your workstation. The API does not support connections through proxy servers.

The following functions connect your application to the Data Collector by establishing a TCP/IP connection between your workstation and the Data Collector, or disconnect both. These are always the first and last steps your application must perform when it needs to communicate with the Data Collector.

The next step after a successful connection is to log on to the Data Collector, as described in "Log On to Data Collector" on page 18.

A user remains logged on to the Data Collector, regardless of the connection state. An application can disconnect from and reconnect to the Data Collector without changing a user's status in the Data Collector. However, after a reconnect the application must issue a **pmLogon()** function call to allow the Data Collector to verify the user's authorization. See also "Log On to Data Collector" on page 18.

The **pmConnect()** function establishes a handle, which identifies a unique TCP/IP connection to the Data Collector. The handle is used by subsequent function calls to identify this TCP/IP connection. It remains available until the connection is terminated by a **pmDisconnect()** function call. The handle is also used to ensure the correct character conversion between the host's and the workstation's code pages.

An application can establish multiple connections to one or more Data Collectors by issuing multiple **pmConnect()** function calls. It might also, in case of a client/server application, connect multiple client workstations to Data Collectors. More complex, these client workstations might use different code pages and character representations.

To ensure correct character conversion between a Data Collector and a workstation, each connection maintains its own conversion table. With each **pmConnect()** function call a workstation's character representation must be specified (either as its code page number, or a constant if it uses UCS-2). Related function calls must use the same handle to ensure correct EBCDIC to ASCII or EBCDIC to UCS-2 conversions.

Connect to Data Collector

Function Call

```
pmReturnCodes pmConnect (char*        host,
                         char*        servicePort,
                         unsigned int codePage,
                         pmHost*      handle)
```

Header File

pmConnect.h

Description

This function opens a TCP/IP connection to a Data Collector and returns parameter **handle**, which identifies the opened TCP/IP connection. Subsequent function calls that communicate with the Data Collector use this handle to identify this connection. If several **pmConnect()** function calls are made, every connection gets a unique handle.

Before the connection is confirmed by a return code of 0, the API loads an internal conversion table and a copy of the field table from the Data Collector to the workstation's memory. The conversion table ensures correct EBCDIC to ASCII or EBCDIC to UCS-2 (and vice versa) conversions and code page conversions (see "Code Page Conversions" on page 9). The field table contains a list of all DB2 PM counters that the Data Collector supports. The field table is used internally to validate the counters you specify with the API functions. This operation is not apparent to your application.

Note that every **pmConnect()** function call causes a download of a conversion table and a field table. This is because the connections could be done to different Data Collectors, whereby each Data Collector could use different code pages and field tables.

Parameters

1. **host** (input)

   The host where DB2 is installed. The host processor is identified by either:

   - The name of the host as specified in the local *hosts* file. The local *hosts* file contains the mappings of the IP addresses to host names.
   - The name of the host, as registered in the Domain Name System server.
   - The IP address of the host, in dotted-decimal form.

2. **servicePort** (input)

   The TCP/IP port where the Data Collector is listening. The port is specified as either:

   - A decimal number between 1024 and 65535, as string.
   - The service name, as registered in the local *services* file.[5] The local *services* file contains port numbers for well-known services as defined by # RFC 1060 (Assigned Numbers). This is an alias for the decimal port number.

3. **codePage** (input)

   The number of the code page (ASCII or EBCDIC) used by the application program, or the constant `PM_UCS2_CP`, if the application program uses UCS-2 character representation. All strings sent to or received from the Data Collector are converted between the host code page and this code page, respectively character representation.

4. **handle** (output)

   A platform-independent handle that identifies this TCP/IP connection to the Data Collector. The application must provide the memory area for this handle.

Example

```
#include "pmConnect.h"

pmHost myHandle;

// connect to data collector
```

5. You find this file, for example on Windows NT, in directory C:\WINNT\SYSTEM32\DRIVERS\ETC.

```
error = pmConnect("10.0.0.1", "5000", 850, &myHandle);
if(error.returnCode)
{
  printf("Error [%d/%d]\n", error.returnCode, error.reasonCode);
  exit(-1);
}
```

Disconnect from Data Collector

Function Call

```
pmReturnCodes pmDisconnect (pmHost* handle)
```

Header File

pmConnect.h

Description

This function terminates the TCP/IP connection to the Data Collector that is identified by parameter **handle**.

Other TCP/IP connections, if any, remain open. Users logged on to the Data Collector remain logged on.

The code page and the field table that were downloaded when this connection (identified by parameter **handle**) was established are removed from the workstation's memory. Other code pages and field tables that are associated with a different handle remain in memory.

Parameters

1. **handle** (input)

   The platform-independent handle that identifies the TCP/IP connection to be terminated. The handle was set by the **pmConnect()** function.

   This handle becomes unusable for other function calls after this function is called.

Example

```
#include "pmConnect.h"

pmHost       myHandle;
pmReturnCodes error;
...

// connect to data collector
error = pmConnect("10.0.0.1", "4711", 850, &myHandle);
if(error.returnCode)
{
  printf("Error [%d/%d]\n", error.returnCode, error.reasonCode);
  exit(-1);
}

// do DB2 PM commands
...

// disconnect
error = pmDisconnect(&myHandle);
if(error.returnCode)
{
  printf("Error [%d/%d]\n", error.returnCode, error.reasonCode);
  exit(-1);
}
```

Maintaining a User Session to the Data Collector

These functions are used to log on users to the Data Collector or log off from it, to generate encrypted passwords for use during the logon, and to gather version information from the Data Collector to verify its compatibility with the API.

You use these functions after connections to the Data Collector are made, or when tasks associated to a user are completed and the Data Collector resources are to be released.

Note:

- Every work profile specifying an individual SAF user ID and SAF group ID combination must be authorized during logon.
- Work profiles that specify subusers of an already authorized SAF user ID and SAF group ID combination do not need explicit authentication.
- Every work profile must be authorized on every TCP/IP connection it uses.
- Every work profile allocates resources in the Data Collector. To release these resources, log off individual users if no longer required.
- The DB2 PM API does not limit the number of subusers per SAF user.
- After the first successful connection and logon to the Data Collector, call the **pmGetInfo()** function to allow the Data Collector to detect and report a potential version mismatch to the API.

Log On to Data Collector

Function Call

```
pmReturnCodes pmLogon (pmHost*  handle,
                       char*    workProfile,
                       char*    identification)
```

Header File

pmLogOnOff.h

Description

This function logs on an SAF user to the Data Collector. It is required before the application can call any other function for this user and all subusers.

If the application logs on a user (with an identical work profile) a second time, without intermediate logoff, the Data Collector returns a warning message (“user already logged on”).

Parameters

1. **handle** (input)

   The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.

2. **workProfile** (input)

   A 48-character string representing the work profile that identifies a user’s working environment. It contains the following fields:

   - SAF User ID: 8-character field; content left aligned; padded with blanks, if required. This field is required.

     The user ID of the DB2 PM user.

- SAF Group ID: 8-character field; content left aligned; padded with blanks, if required. This field is optional and must be filled with blanks, if not used.

  The group ID of the DB2 PM user.
- Profile ID: 16-character field; content left aligned; padded with blanks, if required. This field is optional and must be filled with blanks, if not used. Only alphanumeric characters in the range from A to Z and 0 to 9 are allowed to specify an SAF group ID.

  An application-specific or user-specific field that can be used, for example, to specify subusers.

  This field is not used to authenticate subusers; only the SAF user is authenticated.
- Terminal ID: 16-character field. Reserved. Fill this field with blanks.

3. **identification** (input)

   This character string represents the password, or the PassTicket generated by **pmGenPassticket()**. The OS/390 Security Server (RACF) uses this string to authenticate the SAF user. The character string must be 1 to 8 characters long.

   If the OS/390 Security Server (RACF) is used, **identification** must be a RACF password or a RACF PassTicket.

   Note: Be aware about the limited security when using a password with the **identification** parameter. Use a PassTicket, if possible. Refer to "The Security Concept" on page 5.

Example

```
#include "pmConnect.h"
#include "pmLogOnOff.h"

pmHost   myHandle;
char     workProfile[] = "PMUSER          DB2PM           10.0.0.1:0001   ";
//                        <-UID -><-GID -><- Profile ID -><-Terminal ID->

// connect to data collector
...

// log on with password
error = pmLogOn(&myHandle, workProfile, "TEST");
if(error.returnCode)
{
  printf("Error [%d/%d]\n", error.returnCode, error.reasonCode);
  exit(-1);
}
```

Generate RACF PassTicket

Function Call

```
pmReturnCodes pmGenPassticket (pmHost        *handle,
                               char          *userID,
                               char          *application,
                               char          *secureSignonKey,
                               char          *passticket)
```

Header File

pmGenPassticket.h

Description

This function generates a RACF PassTicket. Use it after a **pmConnect()** and before a **pmLogon()** function call. You can use a PassTicket only once. Once generated, it remains usable for 10 minutes.

PassTickets are unique. The Data Collector ensures that no identical PassTickets are build.

Note: Note that the letter T is in uppercase in the RACF term “PassTicket”. As a function call you must spell it “pmPassticket” with a lowercase t because function names are case sensitive.

The use of the **pmGenPassticket()** function requires that:

- The OS/390 Security Server (RACF) is set up up recognize the name of the application using the **pmGenPassticket()** function. Also, a secure signon key must be associated with the application name. See “Preparations for Using PassTickets” on page 11 for details.
- The application can access the secure signon key on the workstation. For information on how to define a secure signon key and the RACF PassTicket mechanism see *OS/390 Security Server (RACF) - Introduction*.

Parameters

1. **handle** (input)

   The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.

2. **userID** (input)

   The RACF user ID. It must be 1 to 8 characters long.

3. **application** (input)

   The name of the application as specified in the OS/390 Security Server (RACF). Use `MVSDB2PM` as the application name. This name is required for **pmGenPassticket()**.

4. **secureSignonKey** (input)

   The secure signon key associated with the name of the application in RACF. This key must be 16 characters long.

5. **passticket** (output)

   Pointer to an 8-character string storing the generated PassTicket. For UCS-2 output data ensure that enough memory is available.

Example

```
#include "pmConnect.h"
#include "pmLogOnOff.h"
#include "pmGenPassticket.h"

pmHost   myHandle;
char     passticket[9];
char     workProfile[] = "PMUSER          DB2PM           10.0.0.1:0001   ";

// connect to data collector
...

// generate passticket
error = pmGenPassticket(&myHandle, "PMUSER", "MVSDB2PM",
                        "E001033FAF00007B", passticket);
if(error.returnCode)
  ...

// log on with password
error = pmLogon(&myHandle, workProfile, passticket);
if(error.returnCode)
```

```
{
  printf("Error [%d/%d]\n", error.returnCode, error.reasonCode);
  exit(-1);
}
```

Get Data Collector Information

Function Call

```
pmReturnCodes pmGetInfo (pmHost*      handle,
                         char*        workProfile,
                         pmHashTable  result)
```

Header File

pmGetInfo.h

Description

This function requests the Data Collector to return information about itself, about the DB2 PM API in use, and various other information, including:

- DB2 version number and release number
- Data Collector version number and release number
- DB2 PM API version number and release number
- Trial period status
- Time information
- Information about users logged on to the Data Collector
- Information about the data sharing group, if the application is connected to a DB2 subsystem in a Sysplex
- Data sharing group support status.

This function also verifies whether the Data Collector release is sufficient for the DB2 PM API release on the workstation. If it is not, the Data Collector returns a warning, and some API functions might fail. Therefore, it should be used after a successful connection to the Data Collector to allow the Data Collector to return a warning, if necessary. Notice that this function does not detect a down-level DB2 PM API.

You can use this function before any user is logged on to the Data Collector. This allows, for example, to check whether a specified user is logged on.

This function requires that the output data area is initialized with the **clearHashTable()** function.

Parameters

1. **handle** (input)

   The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.

2. **workProfile** (input)

   A user's work profile for which this function is to be executed. The user was specified by the **pmLogon()** function.

   If you want to issue this function before a user is logged on to the Data Collector, or if you do not want to associate a work profile to this function call, specify `NULL`.

3. **result** (output)

Pointer to the output data area. See "Working with Returned Data" on page 36 for how to retrieve individual counter values. Use the **freeHashTable()** function to release the memory area, if the output data is no longer needed.

result	REPDCINF	Groups all Data Collector information counters.		
		QR4TID	Time difference between the local time of the host processor and the universal time UT[6] (note 3a)	
		QR4DB2	DB2 version	
		QR4PM	Data Collector version (as string)	
		QR4PMI	Data Collector release (as integer)	
		QR4TBS	DB2 PM license status and the installation status (note 3b)	
		QR4TBD	Trial period duration (note 3c on page 23)	
		QR4TIME	Current host processor time, expressed as UT	
		QR4APIV	DB2 PM API version (as string)	
		QR4APIR	DB2 PM API release (as string)	
		QR4APII	DB2 PM API version (as integer) (note 3d on page 23)	
		REPUIDS	Groups user information. Contains one repeating block per SAF user ID and SAF group ID combination.	
			QR4UID	The SAF user ID
			QR4GID	The SAF group ID
			QR4SEA	Number of open APPC sessions (note 3e on page 23)
			QR4SET	Number of open TCP/IP sessions (note 3f on page 23)
			QR4CRDT	Collect Report Data status (note 3g on page 23)
			QR4CRDR	Collect Report Data status (note 3g on page 23)
			QR4CRDP	Collect Report Data status (note 3g on page 23)
		QR4GRPN	Name of the data sharing group (note 3h on page 23)	
		QR4MBRN	Name of the data sharing group member (note 3i on page 23)	
		QR4DS	Flag for data sharing group support availability (note 3j on page 23)	

Notes:

a. QR4TID: Because DB2 and the Data Collector use universal time (UT) internally, this time difference must be used to adjust counter values that represent date and time information back to local times. See "Converting and Adjusting Dates and Times" on page 89 for details.

b. QR4TBS:

6. For practical reasons within the scope of this guide universal time (UT) is used synonymously with coordinated universal time (UTC) and Greenwich mean time (GMT).

- PAYED indicates that the DB2 PM installation is a licensed version.
- TRIAL indicates that the DB2 PM installation is a trial version.
- RESTRICTED indicates that the trial period is exceeded.
- INSTALLERROR indicates that the DB2 PM Buy feature is not installed correctly.

c. QR4TBD: Number of days left until the trial period expires. If the trial period has expired, this value is 0.

d. QR4APII: If this counter shows an integer value of 6 or greater, the API supports data sharing groups.

e. QR4SEA: Number of open APPC sessions with the Data Collector for this SAF user ID and SAF group ID combination. APPC sessions with the Data Collector may be started from another program. Therefore, the information is returned as well.

f. QR4SET: Number of open TCP/IP sessions with the Data Collector for this SAF user ID and SAF group ID combination.

g. QR4CRDx: The information is returned here because Collect Report Data might be managed in the Data Collector from the DB2 PM Workstation Online Monitor for this SAF user ID and SAF group ID combination.

h. QR4GRPN: If the application is not connected to a data sharing group, the counter attribute contains NA (counter not available).

i. QR4MBRN: If the application is not connected to a data sharing group, the counter attribute contains NA (counter not available).

j. QR4DS: This counter works as a flag. It indicates whether the application program can successfully use API functions that request data from data sharing group members. Data sharing group functions are active if all of the following conditions are true:

   - The version of DB2 to which the Data Collector is connected is 6.1 or higher.
   - The Data Collector must be started with data sharing group support active (startup parameter DATASHARINGGROUP=YES).
   - The DB2 subsystem to which the Data Collector is connected must be a member of a data sharing group.

   If these conditions are true, the counter contains a value of 1, otherwise the counter is set to 0.

Example

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <time.h>
#include "pmConnect.h"
#include "pmGetInfo.h"
#include "pmTrace.h"

void printCounter(pmHashTable table, char *name);

int main(void)
{
  pmHost        myHandle;
  pmReturnCodes error;
  char          *workprofile = "PMUSER  GROUPID PROFILEID       TERMINALID      ";
    pmCounter*    aCounter;
  pmCursor      aCursor;
  pmHashTable*  DCInfo;
  pmHashTable*  UserInfo;
```

```
time_t        time;

/* connect to data collector */
error = pmConnect("10.0.0.1", "4711", 850, &myHandle);
if(error.returnCode)
{
  printf("pmConnect - Error [%d/%d]\n", error.returnCode, error.reasonCode);
  exit(-1);
}

/* prepare result area */
clearHashTable(&result);

/* get information about data collector and API */
error = pmGetInfo(&myHandle, workprofile, result);

/* check for sufficient data collector release */
if(error.returnCode == PM_APIWARNING && error.reasonCode == PM_OLD_DC)
{
  printf("Be aware that you are using an old DC version.\n");
  printf("Some API functions might fail!\n");
  error.returnCode = 0;
}

if(error.returnCode)
{
  printf("pmGetInfo - Error [%d/%d]\n", error.returnCode, error.reasonCode);
  exit(-1);
}
else
{
  /* first locate repeating block for DC info */
  aCounter = pmGetCounter(result, "REPDCINF");
  if(aCounter != NULL)
  {
    /* access general info */
    aCursor = initCursor(*aCounter);
    DCInfo = getRepBlockItem(aCursor);
    printf("Time difference:          "); printCounter(*DCInfo, "QR4TID");
    printf("DB2 Version:              "); printCounter(*DCInfo, "QR4DB2");
    printf("PM Version:               "); printCounter(*DCInfo, "QR4PM");
    printf("PM Version (internal):    "); printCounter(*DCInfo, "QR4PMI");
    printf("Time Bomb Status:         "); printCounter(*DCInfo, "QR4TBS");
    printf("Time Bomb Days Left:      "); printCounter(*DCInfo, "QR4TBD");
    printf("Time at host:             "); printCounter(*DCInfo, "QR4TIME");

    /* get the logged on users block */
    aCounter = pmGetCounter(*DCInfo, "REPUIDS");
    if(aCounter != NULL)
    {
      aCursor = initCursor(*aCounter);
      while(!endOfBlock(aCursor))
      {
        UserInfo = getRepBlockItem(aCursor);

        printf("  UserID:                   "); printCounter(*UserInfo, "QR4UID");
        printf("  Group ID:                 "); printCounter(*UserInfo, "QR4GID");
        printf("  APPC sessions:            "); printCounter(*UserInfo, "QR4SEA");
        printf("  TCP/IP sessions:          "); printCounter(*UserInfo, "QR4SET");
        printf("  Trace collectors total:   "); printCounter(*UserInfo, "QR4CRDT");
        printf("  Trace collectors running: "); printCounter(*UserInfo, "QR4CRDR");
        printf("  Trace collectors pending: "); printCounter(*UserInfo, "QR4CRDP");

        setToNext(&aCursor);
      }
    }
  }
```

```
    /* free memory for DC info */
    freeHashTable(result);
  }

  /* disconnect */
  error = pmDisconnect(&myHandle);
  if(error.returnCode)
  {
    printf("pmDisconnect - Error [%d/%d]\n", error.returnCode, error.reasonCode);
    exit(-1);
  }

  return(0);
}


void printCounter(pmHashTable table, char *name)
{
  pmCounter *aCounter = pmGetCounter (table, name);
  time_t t;
  char *asHex;
  unsigned int helpI;
  unsigned short helpS;

  if(aCounter != NULL)                          /* if not contained     */
                                                /* NULL is returned     */
  {
    /* check if value is provided by Data Collector                     */
    switch(aCounter->attribute)
    {
      case VALUE:
         /* print counter value                                         */
         if(aCounter->type == PMINT)
         {
           helpI = *((unsigned int*)aCounter->value);
           printf("%d\n", helpI);
         }
         else if(aCounter->type == PMSHORT)
         {
           helpS = *((unsigned short*)aCounter->value);
           printf("%d\n", helpS);
         }
         else if(aCounter->type == PMSTRING || aCounter->type == PMVARCHAR)
           printf("%s\n", aCounter->value);
         else if(aCounter->type == PMTIME)
         {
           printf"PMTIME(%d bytes)\n",aCounter->length);
         }
         else if(aCounter->type == PMDATE)
         {
           tod2time(aCounter->value, &t);
           printf("%s", asctime(gmtime(&t)));
         }
         else if(aCounter->type == PMBIN)
         {
           printf("PMBIN(%d bytes)\n", aCounter->length);
         }
         else
           printf("invalid type!\n");
         break;
      case NC:
         printf("n/c\n");
         break;
      case NP:
         printf("n/p\n");
         break;
      case NA:
```

```
            printf("n/a\n");
            break;
         default:
            /* error */
            printf("invalid attribute!\n");
            break;
      }
    }
    else
    {
      printf("not returned!\n");
    }
}
```

Log Off from Data Collector

Function Call

```
pmReturnCodes pmLogoff (pmHost*  handle,
                        char*    workProfile)
```

Header File

pmLogOnOff.h

Description

This function logs off an SAF user or an individual subuser from the Data Collector. It stops exception processing and clears all snapshot stores and user-specific data in the Data Collector for the specified user.

You use this function to log off an SAF user and all of its subusers, or individual subusers only.

If an individual subuser is logged off, the logoff applies to all active TCP/IP connections.

Parameters

1. **handle** (input)

   The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.

2. **workProfile** (input)

   A user's work profile for which this function is to be executed.

   If you want to log off an SAF user and all associated subusers, fill the Profile ID field in **workProfile** with blanks.

   If you want to log off an individual subuser, identify the subuser in the Profile ID field in **workProfile**.

Example

```
#include "pmConnect.h"
#include "pmLogOnOff.h"
#include "pmConnect.h"
#include "pmLogOnOff.h"

pmHost   myHandle;
char     workProfile[] = "PMUSER          DB2PM           10.0.0.1:0001   ";
char     safUserProfile[] = "PMUSER                                         ";

// connect to data collector
...
```

```
// log on with password
...

// log off from data collector for subuser DB2PM of user PMUSER
error = pmLogoff(&myHandle, workProfile);

// log off from data collector for user PMUSER and all subusers
error = pmLogoff(&myHandle, safUserProfile);
```

Getting DB2 Performance Data

This section introduces the concept of snapshot processing and history processing, explains how to interpret returned data, and describes the API functions relevant to DB2 performance data.

Introduction to Counters and Snapshot Stores

The functions described in this chapter are the core DB2 PM API functions. With these functions the application accesses DB2 statistics data on a subsystem level or on a thread level and DB2 system parameter values.

DB2 performance data is available through the Data Collector from two sources:

- From real-time DB2 instrumentation data. Whenever the application requests data, it gets a snapshot of the present performance of a DB2 subsystem.
- From the *history data set* where the Data Collector continuously records snapshots of DB2 performance data. This gives the application access to older performance snapshots.

The Data Collector provides counter arrangements that ease the application's job to deal with DB2 performance data. The following sections describe the DB2 PM counter concepts. A thorough understanding of these concepts is required to effectively use these API functions.

DB2 PM Counters

DB2 PM counters are the core element of these functions. Counters are data fields that contain DB2 performance data. All fields listed in the field table represent counters.

Counters can contain various information:

- Numeric information, for example, the number of specific DB2 events that occurred, or the length of a data field.
- Alphabetic information, for example, a text string that represents the status of an activity.

Depending on the content of a counter, every counter is associated with a data type. For example, if a counter holds a time stamp of an event, it is associated with a data type of ″TIME″. The field table lists the data type with all counters.

The application addresses counters by names. Even if the field table lists counters by identifiers, there is no need to address a counter by its identifier.

Counters reside in the Data Collector. The Data Collector ensures that the counters are updated with actual DB2 instrumentation data. The application usually examines the counter values for further processing. It does not change counter values.

Counters have different operational behavior:

- Some counters count DB2 activities that occurred since the DB2 subsystem started, for example, the number of checkpoints that DB2 took since startup.
- Some counters hold high water marks, for example, to show the greatest amount of storage that is allocated to a process since it was started. They do not actually count. They might not even change.
- Some counters hold percentage values, for example, to indicate the latest usage of a DB2 buffer pool.

All are generally called counters in this context.

Snapshot Stores

You are rarely interested in a single counter or all available counters. Depending on the task the application wants to perform, you are interested in a defined subset of the available counters. The group of counters you are interested in is called a counter *store*. Because these counters contain data from a snapshot of performance data, the store is called a *snapshot store*. The performance-related API functions use this snapshot store when they request new counter values (take a snapshot) from the Data Collector. Snapshot stores also limit the amount of central storage resources that need to be allocated in the Data Collector.

When you specify the content of a snapshot store, you can mix any of the counters listed in the field table, regardless of the purpose of a counter. You can specify several snapshot stores in the application. Once a snapshot store is initialized, it is given a unique identifier number. The identifier serves as a reference for other functions that work with the same snapshot store.

Qualifying Counters: After you have decided which counters you want to include in a snapshot store, you can apply criteria to these counters. The Data Collector returns counter values only if these criteria are met. The process of specifying the criteria is called *qualifying*. "How Qualifying Works" on page 33 shows a comprehensive example.

Counters are qualified when a snapshot store is initialized. After you have decided which counters to include in a snapshot store, and which qualification criteria to apply to them, the definition of a snapshot store cannot be changed. The definition remains unchanged until the snapshot store is released, or until the user owning this snapshot store is logged off from the Data Collector.

Getting and Viewing Counter Values: Snapshot stores in the Data Collector can be used in two different modes:

- The GET mode causes the Data Collector to refresh the contents of the counters in a snapshot store with DB2 instrumentation data *before* it returns the counter values to the application.

  You use the GET mode, for example, to frequently scan the contents of a set of counters that make up this snapshot store.
- The VIEW mode causes the Data Collector to return the contents of the counters in a snapshot store *without* refreshing them with DB2 instrumentation data.

  Before useful data can be retrieved in VIEW mode, the counters in this snapshot store must have been refreshed with DB2 instrumentation data at least once.

  You use the View mode, for example, if you need to access counter values that were taken at a certain point in time multiple times.

Both modes are used in combination with the two-buffer concept described next.

The mode is controlled by parameter **snapshotParms** of the **pmGetSnapshotEx()** function. You can regard the GET and VIEW modes also as a keyword-controlled toggle flag that controls whether the snapshot store is refreshed before its content is returned to the application.

Buffers are Associated with Counters: Whenever the application deals with counters that represent DB2 performance data, it also deals with time. The application often compares the value of a counter at two given points in time. To ease your programming effort a snapshot store supports simple calculations. This is accomplished by associating two buffers with each counter in a snapshot store.

- The buffer called **Latest** always contains the most recent counter value.

  In GET mode this is the newest DB2 instrumentation data (counter values are refreshed before the results are returned).

  In VIEW mode this is the last value that was stored while in GET mode (counter values are not refreshed in VIEW mode).
- The buffer called **Stored** serves as a reference value for the calculations described next. It can contain:
  - A previous value of buffer **Latest** to serve as a moving reference value.
  - The most recent value of buffer **Latest** to serve as a fixed reference value (set by the **pmResetEx()** function).

Buffers are Used for Interval Processing and Delta Processing: The two-buffer concept enables time-based processing of counter values in different ways. In the simplest way the content of buffer **Latest** can be requested for further processing without reference to buffer **Stored**.

- Buffer **Stored** can be set as a *fixed* reference point (by the **pmResetEx()** function) and subsequent **pmGetSnapshotEx()** functions return the cumulative differences between the content of buffer **Stored** and the content of buffer **Latest**. This is called *interval processing*.

  In GET mode the cumulative difference increases (supposed that the refreshed counter value in buffer **Latest** increases).

  In VIEW mode the cumulative difference remains unchanged (because no counter is refreshed).
- Buffer **Stored** can be set as a *moving* reference point. Each time a **pmGetSnapshotEx()** function is called in GET mode it:
  1. Copies the current content of buffer **Latest** to buffer **Stored**
  2. Refreshes the buffer **Latest** with the latest DB2 instrumentation data counter value
  3. Returns the difference between both buffer contents.

  This is called *delta processing*.

  In VIEW mode the difference remains unchanged because no counter is refreshed.

Again, parameter **snapshotParms** of the **pmGetSnapshotEx()** function controls whether you want to request buffer **Latest**, perform interval processing, or perform delta processing with a snapshot store. The keywords are LATEST, INTERVAL, and DELTA.

More about interval and delta processing is described in "Working with Snapshot Stores" on page 32.

Possible Modes to Control the Snapshot Store Operation: **pmGetSnapshotEx()** is the function that requests counter values from a snapshot store. It uses the described modes and buffers. Its parameter **snapshotParms**, member *mode*, controls:

- Whether the contents of counters are requested in GET mode or VIEW mode.
- Which result the function returns (buffer **Latest**, the interval result, or the delta result).

The following list shows valid *mode* keywords for parameter **snapshotParms** and summarizes their counter operations. You must specify one of these modes with each function call.

Note that the described activities affect *all* qualified counters in a snapshot store.

- `GET_LATEST`
  1. Does not copy the content of buffer **Latest** to buffer **Stored**.
  2. Refreshes buffer **Latest** with performance data.
  3. Returns the content of buffer **Latest**.
- `VIEW_LATEST`
  1. Does not copy the content of buffer **Latest** to buffer **Stored**.
  2. Does not refresh buffer **Latest** with performance data.
  3. Returns the content of buffer **Latest**.
- `GET_INTERVAL`
  1. Leaves the buffer **Stored** unchanged.
  2. Refreshes buffer **Latest** with performance data
  3. Calculates the difference between the values in buffers **Latest** and **Stored**.
  4. Returns the difference.

  This mode should be applied only to statistics counters.
- `VIEW_INTERVAL`
  1. Leaves the buffer **Stored** unchanged.
  2. Does not refresh buffer **Latest** with performance data.
  3. Calculates the difference between the values in buffers **Latest** and **Stored**.
  4. Returns the difference.

  This mode should be applied only to statistics counters.[7]
- `GET_DELTA`
  1. Copies the content of buffer **Latest** to buffer **Stored**.
  2. Refreshes buffer **Latest** with performance data.
  3. Calculates the difference between the values in buffers **Latest** and **Stored**.
  4. Returns the difference.

  This mode should be applied only to statistics counters.[7]
- `VIEW_DELTA`
  1. Leaves the buffer **Stored** unchanged.
  2. Does not refresh buffer **Latest** with performance data.
  3. Calculates the difference between the values in buffers **Latest** and **Stored**.
  4. Returns the difference.

7. If this mode is applied to other counter types, no calculations are performed and the content of buffer **Latest** is returned.

This mode should be applied only to statistics counters.[7]

You need to specify one of these keywords with each **pmGetSnapshotEx()** function call (in member *mode* of parameter **snapshotParms**). Additional keywords (described next) allow you to refine the operation of a snapshot store. Valid combinations of *mode* keywords are described in "Snapshot Processing - Get Snapshot Data" on page 45.

Mode to Control the Data Source: Snapshot stores get their data either from DB2 as real-time performance data, or from the history data set, which contains snapshots of DB2 performance data that were taken previously. If you use a snapshot store in GET mode (performance data is refreshed before it is returned to the application), you can choose whether you want it refreshed with actual DB2 performance data, or with data from a snapshot of the history data set. In the latter case, the snapshot to be used is identified by its time stamp.

You control whether data should be taken from the history data set by adding the `GET_HISTORY` keyword to the **pmGetSnapshotEx()** function call (member *mode* of parameter **snapshotParms**). `GET_HISTORY` works as a flag. If specified as an additional *mode* keyword, data is taken from the history data set. If it is not specified, data is taken from actual DB2 performance data.

The `GET_HISTORY` keyword has no effect if you use a snapshot store in VIEW mode, because in VIEW mode the contents of the counters in a snapshot store are not refreshed.

Mode to Retrieve Counters about Locked Resources: The **pmGetSnapshotEx()** function also allows you to retrieve counter values about locked resources (IFCID[8] 150). You can specify (in member *mode* of parameter **snapshotParms**) mode `GET_LOCKEDRESOURCES` together with mode `GET_LATEST` or `VIEW_LATEST`. You can also combine this with mode `GET_HISTORY` to control whether actual DB2 performance data or data from a selected snapshot from the history data set should be taken.

If specified together with `GET_LATEST`, buffer **Latest** is refreshed with latest performance data (or with data from the history data set), and the content of buffer **Latest** is returned.

If specified together with `VIEW_LATEST`, buffer **Latest** is not refreshed, and the content of buffer **Latest** is returned. Use this combination if you want to access the counter values more than once, without having them changed.

Mode to Summarize Thread Counter Values: The **pmGetSnapshotEx()** function also allows you to summarize thread counter values. If you specify (in member *mode*) of parameter **snapshotParms**) mode `GET_SUMMARY` as an additional keyword, the **pmGetSnapshotEx()**function returns the sum of the requested counter values. Generally, this mode is applicable only to thread counters. If unsupported thread counters are specified, they are ignored.

You can specify `GET_SUMMARY` together with the modes listed in "Possible Modes to Control the Snapshot Store Operation" on page 30. You can also combine this mode with mode `GET_HISTORY` to control whether actual DB2 performance data or data from a selected snapshot from the history data set should be taken.

8. IFCIDs are Instrumentation Facility Component Identifiers. The instrumentation facility is a DB2 performance monitoring function. An IFCID names a traceable event and identifies the trace record of that event.

Working with Snapshot Stores

This section gets together the elements around counters and snapshot processing, and gives some practical examples.

Functions Involved: The **pmInitializeStoreEx()** function creates a snapshot store. You specify the counters that this snapshot store should contain, and you can specify a qualifier list to apply criteria to these counters.

The **pmGetSnapshotEx()** function request counter information from the Data Collector. Member *mode* of its **snapshotParms** parameter controls whether you want to only view the contents of the counters, or have the Data Collector to refresh them before the results are returned. Optionally, it controls whether you want the refresh to be done from a selected snapshot of the history data set. Parameter member *mode* also controls which calculations you want the Data Collector to perform.

The **pmResetEx()** function sets a reference point that subsequent **pmGetSnapshotEx()** function calls in mode `GET_INTERVAL` use to calculate time differences.

The **pmQueryStores()** function queries which snapshot stores are active. You use it, for example, if you want to know when a snapshot store was refreshed with actual DB2 performance data for the last time.

The **pmGetHistoryContents()** function retrieves a list of snapshots from the history data set. The list can include all snapshots, or a specified subset of snapshots, and includes the IFCIDs[8] and time stamps of the selected snapshots. Use this function to identify a snapshot that you want to use with the **pmGetSnapshotEx()** function in mode `GET_HISTORY` for a refresh. The snapshot is identified by its time stamp.

The **pmReleaseStore()** function releases a previously created snapshot store.

Counter Arithmetic: You could use snapshot stores to return their counter values and let the application execute any arithmetic required. To ease your programming effort you can also direct the **pmGetSnapshotEx()** function to perform some simple calculations and to return the result of this calculation.

- Delta processing

  If you want to know the difference between the content of a counter at two points in time, use the **pmGetSnapshotEx()** function in mode `GET_DELTA`.

  Whenever the function is called, the difference between the content of buffer **Latest** and buffer **Stored** is calculated, and the result is returned.

  Example: You want to periodically examine database read access that was delayed because of unavailable system resources. The corresponding DB2 counter increments with each event since the DB2 subsystem is started. The latest counter value is 1 200. The application calls the **pmGetSnapshotEx()** function in mode `GET_DELTA` every five minutes. Assuming that the next absolute counter values are 1 205, 1 212, and 1 220, the function returns values of 5, 7, and 8 each time it is called.

- Interval processing

  If you want to know the *accumulated* difference between the content of a counter at two points in time, use the **pmGetSnapshotEx()** function in mode `GET_INTERVAL` in combination with the **pmResetEx()** function.

First, issue a **pmResetEx()** function call. This refreshes the buffer **Stored** with the latest performance data (either real-time performance data, or data from the history data set). The content of this buffer then serves as a reference value for the following calculations.

Next, with every subsequent **pmGetSnapshotEx()** function call in mode `GET_INTERVAL` the difference between the content of buffer **Stored** and buffer **Latest** is calculated, and the result is returned.

Example: You want to know how the number of successful Create Thread requests develops between 8:00 a.m. and 9:00 a.m., in intervals of 60 seconds. At 8:00 a.m. the application issues a **pmResetEx()** function call, which copies the latest counter value of 500 to buffer **Stored** as a reference value. Then the application calls the **pmGetSnapshotEx()** function every 60 seconds. Assuming that the next four absolute counter values are 550, 600, 660, and 750, subsequent **pmGetSnapshotEx()** function calls return values of 50, 100, 160, and 250.

Interval Processing with Mixed Data Sources: Interval processing requires that you use the **pmResetEx()** function to set a fixed reference point before you use the **pmGetSnapshotEx()** function. If you want the latter function to use a snapshot from the history data set, you use the `GET_HISTORY` keyword (in member *mode* of its **snapshotParms** parameter).

The **pmResetEx()** function has its own parameter (member *mode* in parameter **resetParms**) to control the data source. If set to `GET_DB2`, actual DB2 performance data is used. If set to `GET_HISTORY`, a selected snapshot from the history data set is used to set a fixed reference point for interval processing.

For interval processing you can use any combination of data sources for the **pmResetEx()** and **pmGetSnapshotEx()** function. However, the snapshot used to set the reference point with the **pmResetEx()** function should be older than the oldest expected snapshot from a **pmGetSnapshotEx()** function call. Meaningful combinations are:

- Both, the **pmResetEx()** and the **pmGetSnapshotEx()** function, use real-time DB2 performance data. If both functions are called in the proper sequence, the reference point is older.
- Both, the **pmResetEx()** and the **pmGetSnapshotEx()** function, use snapshot data from the history data set. Your application needs to ensure that **pmGetSnapshotEx()** does not select a snapshot that is older than the one selected for **pmResetEx()**.
- The **pmResetEx()** function uses a snapshot from the history data set to set the reference point, and the **pmGetSnapshotEx()** function uses real-time DB2 performance data. Here, the reference point is always older.

How Qualifying Works: Qualifying allows you to apply criteria to counter stores. This frees the application from filtering counter values in which you are not interested.

You specify the criteria for a snapshot store by setting up a *qualifier list*, which contains one or more list entries. Each list entry specifies one criteria.

The qualifier list is applied to a snapshot store at the time the store is initialized by the **pmInitializeStoreEx()** function. Qualification criteria cannot be changed after the store is initialized. If subsequent **pmGetSnapshotEx()** function calls request data, only those data is returned that matches all criteria.

A qualifier list can also be applied to the **pmGetHistory()** function. This function retrieves snapshot data from the Data Collector's history data set. If a qualifier list is applied, only those data is retrieved that matches all criteria.

An entry in the qualifier list uses a **qualification ID** that defines the *type of criteria* you can specify. The field table shows the available qualification IDs in section "Qualification IDs". Each qualification ID gets a value assigned that specifies the criteria value.

For example, you use the qualification ID WQALPLAN if you want to specify a thread by its plan name. You assign this qualification ID a value of DGOPMDC. If this is an entry in the qualifier list used when you initialize a snapshot store, only counter values are returned from thread with plan name DGOPMDC.

The following example shows how a qualifier list is created. First, the qualifier list is initialized using the **initQualifierList()** function. Then, the **addQualifier()** function is used to add two list entries. The third and fourth parameter of the **addQualifier()** function represent the qualification ID and the corresponding value. Last, the **pmInitializeStore()** function uses this qualifier list (fifth parameter).

```
pmHost          myHandle;
char           *workprofile = "PMUSER  GROUPID PROFILEID       TERMINALID      ";
pmReturnCodes   error;
char           *fields[] = { "QISEDBD", "QTMAXDS", "QTAUCHK" };
unsigned int    counterNo = 3;
pmQualifierList qualifier;  // the qualifier list
char           *userData = "DB2 PM snapshot store";
unsigned int    store_id;

// initialize qualifier list
initQualifierList(&qualifier);

// disable display of single and multiple held locks
// to show only locked resources with suspensions
error = addQualifier(&myHandle, &qualifier, "TLRSINGL", " ");
error = addQualifier(&myHandle, &qualifier, "TLRMULT", " ");

// initialize snapshot store with qualification
error = pmInitializeStore(&myHandle, workprofile, fields, counterNo,
                          &qualifier, userData, &store_id);
```

Note that qualification IDs of type VARCHAR are not supported. See the column "Field Types" in the field table for applicable types.

See "Miscellaneous API Functions" on page 99 for details about the **initQualifierList()** and **addQualifier()** functions.

More information about qualification can be found in the *DB2 for OS/390 Administration Guide*. It includes a section about "Programming for the Instrumentation Facility Interface (IFI)".

Working with the History Data Set

A history data set is a collection of performance snapshots that the Data Collector collects when it is operating. You use the **pmGetHistory()** function to retrieve stored snapshots from the history data set. As with snapshot processing, you can specify the counters for which you want to retrieve data, and you can qualify the counters to apply criteria.

If the application accesses performance data in a history data set, you should bear in mind the characteristics of a history data set and the method to access individual snapshots.

Characteristics of a History Data Set: Several parameters about the history data set are specified as Data Collector startup parameters. The following parameters influence the actuality, content, and amount of snapshot data:

- First, whether history recording is activated at Data Collector startup time or not. The history data set gets updated with new snapshots only if history processing is active. Nevertheless, you can use all API functions that access the history data set also if history processing is not active (which makes sense only if the history data set holds at least some performance snapshots).
- The size of the history data set. The data set is used as a wrap-around data set. When new snapshots are added, the oldest snapshots disappear after the data set space fills up.
- The content of the data set. Generally, thread counters, statistic counters, and DB2 system parameters are saved in the data set. However, only snapshots of IFCIDs[8] that are specified as Data Collector startup parameters and qualified thread data are written to the data set.
- The frequency at which snapshots are taken and written to the data set.

Therefore, you may not find all counters in the data set. If necessary, you need to change relevant Data Collector startup parameters. See the *IBM DB2 Performance Monitor for OS/390 Command Reference* for more details.

Function Involved: The **pmGetHistory()** function retrieves a stored snapshot from the history data set. You can select snapshots in different ways:

- By specifying a point in time
- By stepping forward and backward in the history data set.

Selecting Individual Snapshots: Every snapshot recorded in the history data set is given a time stamp when the snapshot is taken. To select individual snapshots the application program needs to know at least the time stamp of the snapshot. Once this snapshot is identified, the application program can step forward or backward in the history data set. The **pmGetHistory()** function has two parameters (**requestTime** and **dir**). They control the selection of an individual snapshot and the change of direction in the history data set. The following scenarios show how the **pmGetHistory()** function and both parameters are used:

The first scenario assumes that you want to start with a snapshot record that is younger than t_1, which might be somewhere in the middle of the history data set:

1. The application calls the **pmGetHistory()** function with parameter **requestTime** set to t_1 and parameter **dir** set to `TO`.

   This function call locates a snapshot next to (**dir** set to `TO`) time t_1. The API chooses the younger snapshot per definition.

   The function returns the requested counter data, and returns the time stamp of this snapshot.
2. Every subsequent **pmGetHistory()** function call takes the time stamp of a preceding function call as a reference to locate a record adjacent to this point in time. You select the direction (the older or the younger snapshot adjacent to this point in time) with parameter **dir** set to `BACK` or `FORWARD`.

   For example, assume that the first **pmGetHistory()** function call returned a snapshot with a time stamp of t_2. You want to locate the next older snapshot in

the history data set. Then, the application would call the next **pmGetHistory()** function with parameter **requestTime** set to t_2, and parameter **dir** set to `BACK`.

The second scenario assumes that you want to start from either the beginning or the ending of the history data set.

- To start from the beginning (the oldest snapshot in the history data set), set parameter **requestTime** to `TOD_FIRST` and parameter **dir** to `FORWARD`.
- To start from the end (the youngest snapshot in the history data set), set parameter **requestTime** to `TOD_LAST` and parameter `dir` to `BACK`.

Once you have located a snapshot at either end, use the returned time stamp to step through the history data set, as in the first scenario.

Working with Returned Data

The following functions return data streams of varying length and structure, depending on the number and type of counters requested:

- **pmGetSnapshotEx()**
- **pmGetHistory()**
- **pmGetHistoryContents()**
- **pmGetInfo()**

The data stream is stored in the application's memory, and the output parameter **result** of these functions points to the output data area in memory.

Some counters return a single value when requested. For example, if you want to examine the number of failures that resulted from an "EDM pool is full" condition, you are interested in the value of counter QISEFAIL (field ID 2601 in the field table). Because DB2 has only one EDM pool, the Data Collector returns only one value for this counter.

Other counters return an unknown number of values. For example, if you want to examine the number of requests made to read a page from the group buffer pool, you are interested in the value of counter QBGLXD (field ID 2702 in the field table). Because DB2 can have several active group buffer pools, the Data Collector returns several values for this counter. (In this example you probably use an additional counter that returns the buffer pool identifier.)

More complex, several counters represent a list of counter groups, whereby list members also can represent counter groups, or a list of counters. For example, counter REPSTAT contains all DB2 statistic counters (REPSTDDF, REPSTBUF, REPSTGBF, and so on), whereby counter REPSTGBF represents a list of counters (one counter for each active DB2 group buffer pool). These counters are called *repeating blocks* and are given a data type of `PMPARSEDREPBLOCK`.

To hide this complexity, repeating blocks are stored in hash tables. The API provides several supporting functions to locate counters in hash tables by their names and to extract counter values. See "Extracting Counters from Hash Tables" on page 95 for details.

The Counter Structure

Every counter in the output data area is represented as a structure named *pmCounter*:

```
typedef struct _counter
{
  char*          name;        /* counter name as string            */
```

```
  unsigned short counterID;   /* counter name as ID                */
  unsigned short length;      /* length of value                   */
  pmCounterType  type;        /* counter type, see below           */
  char*          value;       /* points to value storage           */
  pmCounterAttr  attribute;   /* indicating a valid counter value  */
} pmCounter;

/* possible counter types                                          */
typedef enum {
               PMBIN='B',            /* binary value, not converted */
               PMSTRING='C',         /* string in ASCII             */
               PMVARCHAR='V',        /* string with variable length */
               PMSHORT='S',          /* small integer (2 bytes)     */
               PMINT='I',            /* integer (4 bytes)           */
               PMTIME='T',           /* DB2 time format             */
               PMDATE='D',           /* DB2 date format             */
               PMPARSEDREPBLOCK = 'P'/* repeating block             */
             } pmCounterType;

typedef enum {
               VALUE=0x40,
               NA=0xC1,
               NP=0xD7,
               NC=0xC3
             } pmCounterAttr;
```

Structure *_counter* aggregates six members:

1. *name* contains the name of the counter (which is identical with the counter you have requested, respectively with the name of the counter as it is shown in the field table).
2. *counterID* contains the identifier, as shown in column “Field ID” of the field table.
3. *length* contains the length of the *value* member. The length is the same as shown in column “Field Length” of the field table.

   If structure member *type* is PMVARCHAR (which represents a string with variable length), then member *length* contains the maximum length of the string. The actual length of the string is represented by the first two bytes of the string itself, whereas the actual length does not include this 2-byte prefix.
4. *type* contains the type of the counter, which is one of those defined in *pmCounterType*.
5. *value* contains a pointer to the counter value in memory. The counter value is valid if *attribute* contains `VALUE`. If the counter value is a character string, and the workstation uses ASCII, the string is already converted from EBCDIC to ASCII.
6. *attribute* contains an indicator showing how usable the content of *value* is:
   - `VALUE` indicates a valid content.
   - `NC` indicates “Not Calculated”. A calculation resulted in a division by zero.
   - `NP` indicates “Not present”. A counter value was not set because the corresponding DB2 function was not active. For example, if you query a counter about the number of statements dropped from the dynamic statement cache, and the cache is not enabled, *attribute* contains `NP`.
   - `NA` indicates “Not Available”. A counter was not set because the counter is not available for the installed DB2 subsystem version.

Interpreting Repeating Blocks

The example in this section shows how repeating blocks are treated and how the individual counter values are retrieved from the **result** output data area.

The output data area contains a variable number of REPTHRD repeating block items (one per active DB2 thread). The example steps through all REPTHRD repeating block items until no further item is found.

The following table shows the hierarchy of counters within one REPTHRD repeating block. REPTHRD contains counters, which can again represent repeating blocks, for example, REPTHBUF. The REPTHBUF counter contains a list of counters; one for each buffer pool.

result	REPTHRD	Groups all thread counters of one DB2 thread.		
		counter		
		REPTHBUF	counter*	Buffer pool data repeating block.
		REPTHDAG	counter*	Distributed agent data repeating block.
		REPTHDAC	counter*	Distributed accounting data repeating block.
		REPTHBDS	counter*	Group buffer pool data repeating block in thread record.
		REPTHLCK	counter*	Locked resources repeating block.

The output data stream is stored in a hash table. The following demonstrates how the hash table functions are used in the example:

```
pmCursor cursor = initCursor(pmGetCounter(result, "REPTHRD"));
pmHashTable item;

while(!endOfBlock(cursor))
{
   item = getRepBlockItem(cursor);

   // process data in this repeating block
   ...

   setToNext(cursor);
}
```

Details of these functions are described in "Extracting Counters from Hash Tables" on page 95. With these functions you do not need to know in which sequence individual counters appear in the data stream. The **pmGetCounter()** function locates counters by their names. The **initCursor()** function initializes a cursor, which is used to parse through all items of a repeating block.

```
#include "pmGetStatThread.h"
#include "pmHashTableList.h"   // for cursor functions
#include "pmHashTable.h"       // for repeating block

// counters to request
char *fields[] = {
                   "QWHCAID",  // primary thread auth. ID
                   "QBACPID",  // Buffer Pool name
                 };
// output data area
pmHashTable result;
// counter to get values from output data area
pmCounter* aCounter;
// cursor to step through threads
pmCursor cursor1;
// cursor to step through buffer pools of one thread
pmCursor cursor2;
// pointer to counters in thread repeating block
```

```
pmHashTable* threadCounters;
// pointer to counters in BP repeating block
pmHashTable* bufferPoolCounters;
// number of threads
unsigned int threadNo = 0;

// connect, logon
...

// IMPORTANT: before output data area can be filled,
// initialize it !!!
clearHashTable(&result);

// retrieve output data area in 'result'
...

printf(">> Parsing output data area and printing counters.\n\n");

// first locate the repeating block containing the active DB2 threads
// (one block item for each thread)
aCounter = pmGetCounter (result, "REPTHRD");
if(aCounter != NULL)
{
  // step through all returned threads via cursor
  cursor1 = initCursor (*aCounter);
  while (!endOfBlock(cursor1))
  {
    threadNo++;

    // now get access to counters of this thread
    threadCounters = getRepBlockItem(cursor1);


    // get first requested counter QWHCAID
    aCounter = pmGetCounter (*threadCounters, "QWHCAID");
    if(aCounter != NULL)                          // if not contained
                                                  // NULL is returned
    {
      // check if value is returned by Data Collector
      switch(aCounter->attribute)
      {
        case VALUE:
          // ok, DC returned value
          // -> print counter value
          printf("   Thread %d: QWHCAID has value %s.\n", threadNo,
                 (char*)aCounter->value);
          break;
        case NC:
        case NP:
        case NA:
          printf("   QWHCAID not calculated, present or available.\n");
          break;
        default:
          /* error */
          exit(-1);
      }
    }
    else
    {
      printf("   QWHCAID counter not returned by Data Collector!\n");
    }


    // now search repeating blocks contained in one thread
    // (buffer pools, group buffer pools, remote locations, ...)
    //-> first locate the repeating block for buffer pools
```

```
    aCounter = pmGetCounter (*threadCounters, "REPTHBUF");
    if(aCounter != NULL)
    {
      // the repeating block contains a block item for each
      // active Buffer Pool
      // -> step through all returned Buffer Pools
      cursor2 = initCursor (*aCounter);
      while (!endOfBlock(cursor2))
      {
        // now get access to each returned buffer pool
        bufferPoolCounters = getRepBlockItem(cursor2);

        aCounter = pmGetCounter (*bufferPoolCounters, "QBACPID");
        if(aCounter != NULL)                          // if not contained
                                                      // NULL is returned
        {
          // check if value is returned by Data Collector
          switch(aCounter->attribute)
          {
             case VALUE:
               // print counter value
               if(*(unsigned int*)aCounter->value < 80)
                 printf("   Buffer Pool BP%d is activated.\n",
                       *(unsigned int*)aCounter->value);
               else
                 printf("   Buffer Pool BP32K%d is activated.\n",
                        *(unsigned int*)aCounter->value);
               break;
             case NC:
             case NP:
             case NA:
               printf("   Thread does not use any Buffer Pool,");
               printf(" QBACPID is set to n/p.\n");
               break;
             default:
               // error
               exit(-1);
          }
        }
        else
        {
          printf("QBACPID counter not returned by Data Collector!\n");
        }

        // set cursor to next Buffer Pool in repeating block
        setToNext(&cursor2);
      }
    }
    else
    {
      printf("  QBACPID counter not returned by Data Collector!\n");
    }

    // set cursor to next thread in repeating block
    setToNext(&cursor1);

  }
}

// Free memory for output data area. This frees memory for all
// stored counters too.
freeHashTable (result);


// log off and disconnect
...
```

Snapshot Processing - Initialize Snapshot Store

Function Call

```
pmReturnCodes pmInitializeStoreEx (pmHost*          handle,
                                   char*            workProfile,
                                   int              initStoreLevel,
                                   void*            initStoreParms)
```

This function replaces the **pmInitializeStore()** function.

Header File

pmGetStatThread.h

Description

This function requests the Data Collector to create a snapshot store, to initialize it, and to return a snapshot store identifier.

The identifier is used during subsequent requests to identify the store. In addition, you can assign a 32-byte string to the store.

The snapshot store remains active in the Data Collector. It is released by the **pmReleaseStore()** function, or when the user identified by parameter **workProfile** is logged off.

After a snapshot store is initialized, the scope of data being returned cannot be changed. You must create a new snapshot store, if required.

If you initialize a snapshot store for retrieval of performance data from a data sharing group, or a member of a data sharing group, ensure that the Data Collector is started with the `DATASHARINGGROUP=YES` startup parameter. Otherwise, data is retrieved only from the DB2 subsystem to which your application is connected.

You can use the **pmGetInfo()** function to test whether the application program can successfully use API functions that request data from data sharing group members. Counter QR4DS works as a flag that indicates whether all prerequisites for data sharing group support are met.

If your application works with a data sharing group, you can query information about available members of the group and their activity state with the DB2 command **DISPLAY GROUP**. See "Executing DB2 Commands" on page 77 about how to issue DB2 commands by means of the API.

Parameters

1. **handle** (input)

   The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.

2. **workProfile** (input)

   A user's work profile for which this function is to be executed. The user was specified by the **pmLogon()** function.

3. **initStoreLevel** (input)

   A keyword that specifies the parameter structure to be used in parameter **initStoreParms**. The following keywords are allowed:

- PM_INIT_STORE_BASIC specifies that parameter **initStoreParms** uses a parameter structure for use in the currently attached DB2 subsystem (no data sharing group support).
- PM_INIT_STORE_DSG specifies that parameter **initStoreParms** uses a parameter structure for use in a DB2 data sharing group.

4. **initStoreParms** (input/output)

   Pointer to the parameter structure that contains the required parameters to initialize a snapshot store. Note that the same structure holds the returned snapshot store identifier after successful completion of this function call.

   - If you have specified PM_INIT_STORE_BASIC as parameter **initStoreLevel**, use the following parameter structure:

     ```
     typedef struct _pmInitStoreBasic {
       char**           fields;      /* an array of field names to be requested */
                                     /* in this snapshot store.                 */
       unsigned int     counterNo;   /* number of fields requested              */
       pmQualifierList* qualifier;   /* qualifier information                   */
       char*            userData;    /* user specific identifier for snapshot   */
                                     /* store of 32 bytes or NULL.              */
       unsigned int     storeId;     /* output: snapshot store id as generated  */
                                     /* by the Data Collector.                  */
     } pmInitStoreBasic;
     ```

   - If you have specified PM_INIT_STORE_DSG as parameter **initStoreLevel**, use the following parameter structure:

     ```
     typedef struct _pmInitStoreDSG {
       char**           fields;      /* an array of field names to be requested */
                                     /* in this snapshot store.                 */
       unsigned int     counterNo;   /* number of fields requested              */
       pmQualifierList* qualifier;   /* qualifier information                   */
       char*            userData;    /* user specific identifier for snapshot   */
                                     /* store of 32 bytes or NULL.              */
       unsigned int     storeId;     /* output: snapshot store id as generated  */
                                     /* by the Data Collector.                  */
       char*            dsMember;    /* request data from selected member or    */
                                     /* from connected member if set to NULL.   */
       int              dsGlobal;    /* request data with group scope if set to */
                                     /* PM_GROUP_SCOPE, request data with       */
                                     /* member scope if set to PM_MEMBER_SCOPE  */
     } pmInitStoreDSG;
     ```

   The structure members have the following meaning:

   - *fields* (input) specifies an array of DB2 counter names that the snapshot store should include. You can include statistics counter names and thread counter names that are listed in column "Field Name" of the field table.

     Insert the counter names as follows. Note that counter names are not case-sensitive. Invalid counters are ignored.

     ```
     pmInitStoreBasic   storeParms;

     storeParms.counterNo = 25;
     storeParms.fields[] = {
        "QISEDBD",  "QTMAXDS", "QTAUCHK",  "QTAUSUC",  "QBSTMAX",
        "QBSTPID",  "QBSTGET", "QBSTWFT",  "QBSTWFD",  "QBSTWKPD",
        "QBSTWKPD", "QBSTMAX", "QBSTWKPD", "QBSTWDRP", "QBSTWBVQ",
        "QBSTWFR",  "QBSTWFF", "QBSTWFD",  "QISEDSI",  "QISEDSG",
        "QISEDSC",  "QXSTFND", "QXSTNFND", "QBSTWDRP", "QBSTWBVQ"
        };
     ```

   - *counterNo* (input) specifies the number of counters that are specified with *fields*.

- *qualifier* (input) specifies the name of a qualifier list. The list specifies the criteria which data the Data Collector should return. See "How Qualifying Works" on page 33, if required. If no qualification is required, set this field to NULL.
- *userData* (input) can specify a user- or application-specific 32-byte string that is assigned to this snapshot store. This string is returned by the **pmQueryStores()** function. The 32-byte string is not affected by any code page conversion. UCS-2 encoded strings must also have a length of 32 bytes, and remain unchanged. If *userData* is not used, it should be set to NULL.
- *storeId* (output) contains the numeric snapshot store identifier generated by the Data Collector after successful completion of this function call. This identifier is required to identify this store in subsequent snapshot function calls.
- *dsMember* (input) and *dsGlobal* (input) specify whether performance data should be retrieved from a specified member of a data sharing group, or from all members of a data sharing group.

  dsMember (input) specifies the name of a data sharing group member, or NULL. If it contains a name, the name must be specified as an 8-character string, left aligned, and padded with blanks, if required.

  dsGlobal (input) specifies the scope of the data retrieval and contains one of the two keywords PM_MEMBER_SCOPE and PM_GROUP_SCOPE.

  The following combinations are allowed:

dsMember **set to ...**	*dsGlobal* **set to ...**	**Scope of performance data retrieval**
Specific name of a data sharing group member.	PM_MEMBER_SCOPE	Data is retrieved from the specified member of the group.
NULL	PM_MEMBER_SCOPE	Data is retrieved from the member of the group to which the application is currently connected.
NULL	PM_GROUP_SCOPE	Data is retrieved from all members of the group.

Example

The following example shows how a snapshot store is initialized with data sharing group scope, and how a qualifier list is applied to the store.

```
// the qualifier list
pmQualifierList qualifier;
pmInitStoreDSG  initStoreParms;
pmReturnCodes   error;
unsigned int    store_id;

// connect to DC and log on
...

// set requested fields
initStoreParms.fields[] = { "QISEDBD", "QTMAXDS", "QTAUCHK", "QTAUSUC", "QBSTMAX",
                            "QBSTPID", "QBSTGET", "QBSTWFT", "QBSTWFD", "QBSTWKPD" };
initStoreParms.counterNo = 10;
initStoreParms.userData = "test snapshot store";

// initialize qualifier list
initQualifierList(&(initStoreParms.qualifier));

// show only fields for plans starting with "D"
error = addQualifier(&myHandle, &(initStoreParms.qualifier), "WQALPLAN","D*");

// select data sharing group scope for requests
```

```
initStoreParms.dsMember = NULL;
initStoreParms.dsGlobal = PM_GROUP_SCOPE;

// initialize snapshot store with qualification and data sharing group scope
error = pmInitializeStoreEx(&myHandle, workprofile, PM_INIT_STORE_DSG,
                            initStoreParms);
store_id = initStoreParms.storeId;
```

Snapshot Processing - Query Snapshot Stores

Function Call

```
pmReturnCodes pmQueryStores (pmHost*          handle,
                             char*            workProfile,
                             pmStore**        stores)
```

Header File

pmGetStatThread.h

Description

This function requests the Data Collector to return information about all active snapshot stores for a user that is identified by parameter **workProfile**.

Parameters

1. **handle** (input)

   The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.

2. **workProfile** (input)

   A user's work profile for which this function is to be executed. The user was specified by the **pmLogon()** function.

3. **stores** (output)

   Pointer to a list of all active snapshot stores for this user. Every list member is represented as a structure named *pmStore*. Use the **pmFreeMem()** function to release the memory area, if the output data is no longer needed.

   ```
   typedef struct _store
   {
      unsigned int    id;             // id of this snapshot store
      pmTOD           timeLatest;     // time of snapshot in buffer Latest
      pmTOD           timeStored;     // time of snapshot in buffer Stored
      char            userData[32]    // contains the user defined
                                      // identifier
      struct _store*  next;           // next snapshot store in list
   } pmStore;
   ```

   - *id* contains the numeric snapshot store identifier. This identifier was assigned by the **pmInitializeStoreEx()** function.
   - *timeLatest* contains a time stamp in Store Clock format, which shows when the buffer **Latest** of all counters in this store was changed.

     timeLatest returns `\0\0\0\0\0\0\0\0` (eight bytes set to `0`) if the counters were not used so far.
   - *timeStored* contains a time stamp in Store Clock format, which shows when the buffer **Stored** of all counters in this store was changed.

     timeStored returns `\0\0\0\0\0\0\0\0` (eight bytes set to `0`) if the counters were not used so far.
   - *userData[32]* contains the user data (a 32-byte string) that was assigned to this snapshot store by the **pmInitializeStoreEx()** function. The string is not affected by any code page conversion.

- *next* contains a pointer to the next member in the list of active snapshot stores.

Example

```
#include "pmGetStatThread.h"
#include "pmTOD.h"             // for time conversions
#include "pmTrace.h"           // for asHexString()
#include <time.h>

// beginning of list with returned stores
pmStore* store;
pmStore* help;

time_t   t1,
         t2;
char     *userInfo;
pmHost   myHandle;
char     workProfile[] = "PMUSER          DB2PM           10.0.0.1:0001   ";

// connect and log on to DC
...

// query all stores for this user
error = pmQueryStores (&myHandle, workProfile, &store);

// print infos about all active snapshot stores
while(store)
{
  // print infos
  tod2time (store->timeLatest, &t1);
  tod2time (store->timeStored, &t2);
  userInfo = asHexString (store->userData, 32);
  printf("Snapshot Store %d :\n           \
          \tUserData : %s\n               \
          \tBuffer Latest time : %s\n     \
          \tBuffer Stored time : %s\n\n",
         store->id,
         userInfo,
         ctime(&t1),
         ctime(&t2));

   pmFreeMem(userInfo);

  // set list to next item
  help = store;
  store = store->next;

  // now free memory for list item, if no longer used.
  pmFreeMem(help);
}

// log off and disconnect
...
```

Snapshot Processing - Get Snapshot Data

Function Call

```
pmReturnCodes pmGetSnapshotEx (pmHost*     handle,
                               char*       workProfile,
                               int         snapshotParmLevel,
                               void*       snapshotParms)
```

This function replaces the **pmGetSnapshot()** function.

Header File

pmGetStatThread.h

Description

This function requests the Data Collector to return counter data from a specified snapshot store for a user identified by parameter **workProfile**.

Member *mode* in parameter **snapshotParms** controls the buffer calculations and the results to be returned. It also controls whether actual DB2 performance data or data from the history data set is to be used.

Member *fields* in parameter **snapshotParms** controls which counters from the specified counter store are to be returned.

This function requires that the output data area is initialized by the **clearHashTable()** function.

Parameters

1. **handle** (input)

   The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.

2. **workProfile** (input)

   A user's work profile for which this function is to be executed. The user was specified by the **pmLogon()** function.

3. **snapshotParmLevel** (input)

   A keyword that specifies the parameter structure to be used in parameter **snapshotParms**. The following keyword is allowed:

   - `PM_SNAPSHOT_BASIC` specifies that parameter **snapshotParms** uses a parameter structure for use in the currently attached DB2 subsystem.

4. **snapshotParms** (input/output)

   Pointer to the parameter structure that contains the required parameters for this function. Note that the same structure holds the returned data after successful completion of this function call.

   - If you have specified `PM_SNAPSHOT_BASIC` as parameter **snapshotParmLevel**, use the following parameter structure:

```
typedef struct _pmSnapshotBasic {
  unsigned int   storeId;         /* DC generated snapshot store identifier*/
                                  /* received from pmInitializeStoreEx     */
  pmSnapshotMode mode;            /* request operation mode (one or more   */
                                  /* og GET_LATEST, GET_DELTA,             */
                                  /* GET_HISTORY, ...)                     */
  char**         fields;          /* an array of field names to be         */
                                  /* requested from this snapshot store    */
  unsigned int   counterNo;       /* number of fields requested            */
  pmTOD          historyTime;     /* requested history dataset time stamp  */
                                  /* for mode GET_HISTORY                  */
  pmTOD          timestampLatest; /* output: time stamp of buffer Latest   */
  pmTOD          timestampStored; /* output: time stamp of buffer Stored,  */
                                  /* when applicable                       */
  pmHashTable   *result;          /* output area (diagnostic information   */
                                  /* and snapshot information)             */
} pmSnapshotBasic;
```

   - No other parameter structures are currently supported.

   The structure members have the following meaning:

- *storeId* (input) specifies the identification of the snapshot store for which this function is to be executed. The identification was set by the **pmInitializeStoreEx()** function.
- *mode* (input) specifies how the counters in the identified snapshot store are treated before being returned. The following table summarizes the possible modes and its major characteristics. For a detailed description see "Possible Modes to Control the Snapshot Store Operation" on page 30.

  You need to specify one of the keywords shown in the left column of the table. If you want to specify additional keywords, provide them as shown in the following example. Internally, the keywords are declared as numeric constants, which need to be bitwise ORed to control the program flow.

  ```
  GET_LATEST | GET_HISTORY | GET_LOCKEDRESOURCES
  ```

Mode	Copies "Latest" to "Stored"	Refreshes "Latest"	Returns ...	Remark
GET_LATEST	No	Yes	"Latest"	Optionally, together with GET_HISTORY, or GET_LOCKEDRESOURCES, or both. Alternatively, together with GET_HISTORY, or GET_SUMMARY, or both.
VIEW_LATEST	No	No	"Latest"	Optionally, together with GET_LOCKEDRESOURCES. Alternatively, together with GET_SUMMARY.
GET_INTERVAL	No	Yes	"Latest"–"Stored"	Preset by **pmResetEx()**. Optionally, together with GET_HISTORY, or GET_SUMMARY, or both.
VIEW_INTERVAL	No	No	"Latest"–"Stored"	Optionally, together with GET_SUMMARY.
GET_DELTA	Yes	Yes	"Latest"–"Stored"	Optionally, together with GET_HISTORY, or GET_SUMMARY, or both.
VIEW_DELTA	No	No	"Latest"–"Stored"	Optionally, together with GET_SUMMARY.

  – The GET_HISTORY keyword selects the history data set (instead of actual DB2 performance data) as the data source to refresh the counters in a snapshot store before their values are returned. It can be used together with one of the GET modes, which refreshes buffer **Latest**. The snapshot in the history data set is identified by the time stamp in *historyTime*.

    See "History Processing - Get History Contents" on page 52 about how to retrieve a list of all snapshots in the history data set and how to identify a snapshot by its time stamp.

  – If you specify GET_INTERVAL, you must first use the **pmResetEx()** function to ensure that the reference values for interval processing are set. See "Snapshot Processing - Reset Interval Data" on page 49 for more details.

    If you specify GET_INTERVAL together with GET_HISTORY, ensure that you have selected an appropriate data source also for the **pmResetEx()** function.

– The GET_SUMMARY keyword causes the Data Collector to summarize all thread-related counters. You can combine this keyword with other mode keywords, except GET_LOCKEDRESOURCES.

- *fields* (input) specifies an array of counter names for which counter data are requested.

  If you set *fields* to NULL, counter data is requested for all counters in this snapshot store.

  If you specify a subset of counter names, counter data is requested only for the specified counters. A subset is any number of counters that was specified by the **pmInitializeStoreEx()** function for this snapshot *storeId*.

  To specify a subset of counters insert the counter names as follows. Counter names are not case-sensitive. Invalid counters are ignored. Counters not available in the stored snapshot record in a history data set are ignored. No warning is returned.

```
char *fields[] = {
                   "QISEDBD",  "QTMAXDS", "QTAUCHK",  "QTAUSUC",  "QBSTMAX",
                   "QBSTPID",  "QBSTGET", "QBSTWFT",  "QBSTWFD",  "QBSTWKPD",
                   "QBSTWKPD", "QBSTMAX", "QBSTWKPD", "QBSTWDRP", "QBSTWBVQ",
                   "QBSTWFR",  "QBSTWFF", "QBSTWFD",  "QISEDSI",  "QISEDSG",
                   "QISEDSC",  "QXSTFND", "QXSTNFND", "QBSTWDRP", "QBSTWBVQ"
                 };
```

  If you specify also thread counters in the array, and *mode* is set to GET_SUMMARY, all thread counters in the array are summarized before being returned. See the field table, section “Thread Data”, for valid thread counter names.

- *counterNo* (input) specifies the number of counters that are specified in *fields*. If *fields* is NULL, set *counterNo* to 0.
- *historyTime* (input) specifies a valid time stamp of a snapshot in the history data set that you want to use (if *mode* is set to GET_HISTORY). The time stamp must be in Store Clock format.

  If *mode* is not set to GET_HISTORY, this member is ignored.
- *timestampLatest* (output) contains a returned time stamp in Store Clock format, which shows when the buffer **Latest** of all requested counters in this store was changed.
- *timestampStored* (output) contains a returned time stamp in Store Clock format, which shows when the buffer **Stored** of all requested counters in this store was changed. The returned time stamp is valid only for modes that involve buffer **Stored**, that is, *mode* is one of the following:
  – GET_INTERVAL
  – VIEW_INTERVAL
  – GET_DELTA
  – VIEW_DELTA
- *result* (output) contains a pointer to the output data area. See “Working with Returned Data” on page 36 for how to retrieve individual counter values. Use the **freeHashTable()** function to release the memory area, if the output data is no longer needed.

  If counters about locked resources are requested (*mode* set to GET_LOCKEDRESOURCES), the output data area does not contain the REPTHLCK repeating block under the REPTHRD repeating block. Instead, the following additional information is returned in the output data area:

result	REPTHLCK	Groups all locked resources.	
		counter*	Resource-related IFCID 150 counters.
		REPLOTHR	Thread-related IFCID 150 counters. Contains one repeating block per thread.

If counters from a data sharing group are returned (**pmInitializeStoreEx()** function, parameter member *dsMember* set to `NULL`, and *dsGlobal* set to `PM_GROUP_SCOPE`), the output data area contains the requested counters of all data sharing group members in multiple iterations of the REPSTAT counter.

The output data area may also contain diagnostic information about this function call.

If diagnostic information is returned, it is contained in the REPDIAG repeating block structure.

result	REPDIAG	Groups all diagnostic information.		
		REPDB2	Groups all DB2 diagnostic information (one block per data sharing group member).	
			DB2RC	DB2 return code
			DB2RS	DB2 reason code
			DB2MBR	Data sharing group member name that caused the DB2 return and reason code.

Currently, information is returned only for the data sharing group member with the highest return code.

Snapshot Processing - Reset Interval Data

Function Call

```
pmReturnCodes pmResetEx (pmHost*        handle,
                         char*          workProfile,
                         int            resetParmlevel,
                         void*          resetParms)
```

This function replaces the **pmReset()** function.

Header File

pmGetStatThread.h

Description

This function requests the Data Collector to preset reference values for an identified snapshot store for subsequent interval processing.

Use the **pmResetEx()** function before you use the **pmGetSnapshotEx()** function in mode `GET_INTERVAL`. **pmResetEx()** makes the buffer **Stored** a fixed reference for interval processing.

1. Buffer **Stored** is filled with latest performance data, or with data from a snapshot of the history data set.
2. Buffer **Latest** is released.

Subsequent **pmGetSnapshotEx()** function calls in mode GET_INTERVAL now use the content of buffer **Stored** to calculate the difference to buffer **Latest**. The value in buffer **Stored** is not changed by subsequent **pmGetSnapshotEx()** function calls in mode GET_INTERVAL.

The **pmResetEx()** function is a companion to the **pmGetSnapshotEx()** function, which uses actual DB2 performance data, or snapshot data from the history data set, to refresh the content of buffer **Latest**. Consequently, you also need to specify the data source for the **pmResetEx()** function. Use member *mode* in parameter **resetParms** to specify either data source:

- If you specify GET_DB2, the **pmResetEx()** function uses actual DB2 performance data to fill buffer **Stored**.
- If you specify GET_HISTORY, the **pmResetEx()** function uses data from a selected snapshot of the history data set to fill buffer **Stored**. The snapshot in the history data set is specified by its time stamp, in the same manner as with the **pmGetSnapshotEx()** function.

Parameters

1. **handle** (input)

   The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.

2. **workProfile** (input)

   A user's work profile for which this function is to be executed. The user was specified by the **pmLogon()** function.

3. **resetParmLevel** (input)

   A keyword that specifies the parameter structure to be used in parameter **resetParms**. The following keyword is allowed:

   - PM_RESET_BASIC specifies that parameter **resetParms** uses a parameter structure for use in the currently attached DB2 subsystem.

4. **resetParms** (input/output)

   Pointer to the parameter structure that contains the required parameters for this function. Note that the same structure holds the returned data after successful completion of this function call.

   - If you have specified PM_RESET_BASIC as parameter **resetParmLevel**, use the following parameter structure:

```
typedef struct _pmResetBasic {
  unsigned int   storeId;         /* DC generated snapshot store identifier*/
                                  /* received from pmInitializeStoreEx     */
  pmSnapshotMode mode;            /* request operation mode                */
                                  /* (GET_DB2 or GET_HISTORY)              */
  pmTOD          historyTime;     /* requested history dataset time stamp  */
                                  /* for mode GET_HISTORY                  */
  pmTOD          timestampStored; /* output: time stamp of data in         */
                                  /* buffer Stored                         */
  pmHashTable   *result;          /* output area (diagnostic information)  */
} pmResetBasic;
```

   - No other parameter structures are currently supported.

   The structure members have the following meaning:

   - *storeId* (input) specifies the identification of the snapshot store for which this function is to be executed. The identification was set by the **pmInitializeStoreEx()** function.
   - *mode* (input) specifies a keyword that controls the data source to use:

  - GET_DB2 uses actual DB2 performance data.
  - GET_HISTORY uses snapshot data from the history data set. Specify the snapshot with member *historyTime*. See "History Processing - Get History Contents" on page 52 about how to retrieve a list of all snapshots in the history data set and how to identify a snapshot by its time stamp.
- *historyTime* (input) specifies a valid time stamp (in Store Clock format) of a snapshot in the history data set that you want to use. This field is ignored if *mode* is not GET_HISTORY.
- *timestampStored* (output) contains a time stamp in Store Clock format, which shows when the buffer **Stored** was filled with latest DB2 performance data (if mode was set to GET_DB2).

  If *mode* was set to GET_HISTORY and the snapshot specified in *historyTime* was found in the history data set, the time stamps in *timestampStored* and *historyTime* are identical.
- *result* (output) contains a pointer to the output data area that may contain diagnostic information about this function call. Use the **freeHashTable()** function to release the memory area, if the output data is no longer needed.

  If diagnostic information is returned, it is contained in the REPDIAG repeating block structure.

result	REPDIAG	Groups all diagnostic information.		
		REPDB2	Groups all DB2 diagnostic information (one block per data sharing group member).	
			DB2RC	DB2 return code
			DB2RS	DB2 reason code
			DB2MBR	Data sharing group member name that caused the DB2 return and reason code.

Currently, information is returned only for the data sharing group member with the highest return code.

Snapshot Processing - Release Snapshot Store

Function Call

```
pmReturnCodes pmReleaseStore (pmHost*          handle,
                              char*            workProfile,
                              unsigned int     id)
```

Header File

pmGetStatThread.h

Description

This function requests the Data Collector to release one or all snapshot stores that are assigned to the user identified by parameter **workProfile**.

Parameters

1. **handle** (input)

   The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.
2. **workProfile** (input)

   A user's work profile for which this function is to be executed. The user was specified by the **pmLogon()** function.

3. **id** (input)

   The identification of the snapshot store for which this function is to be executed. The identification was set by the **pmInitializeStoreEx()** function.

   If you want to release an individual snapshot store, specify the store with parameter **id**.

   If you want to release all snapshot stores of the user identified by parameter **workProfile**, set **id** to 0.

History Processing - Get History Contents

Function Call

```
pmReturnCodes pmGetHistoryContents (pmHost*          handle,
                                    char*            workProfile,
                                    unsigned int*    ifcids,
                                    unsigned int     ifcidNo,
                                    pmTOD            timestampFrom,
                                    pmTOD            timestampTo,
                                    pmHashTable      result)
```

Header File

pmGetStatThread.h

Description

This function requests the Data Collector to return a list of snapshots from the history data set for the user identified by parameter **workProfile**. The information returned includes the recorded IFCIDs per snapshot and their time stamps.

The amount of returned information can be limited by specifying a time frame, and by specifying IFCIDs for which data should be returned.

This function requires that the output data area is initialized by the **clearHashTable()** function.

Parameters

1. **handle** (input)

   The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.

2. **workProfile** (input)

   A user's work profile for which this function is to be executed. The user was specified by the **pmLogon()** function.

3. **ifcids** (input)

   This parameter specifies IFCIDs for which data should be returned. This limits the amount of data returned.

   If you do not want to specify individual IFCIDs, specify `NULL`. The list of snapshots returned contains all available IFCIDs.

   If you want to specify individual IFCIDs, specify them as an array of unsigned integer values. The list of snapshots returned contains snapshots that contain at least one of the specified IFCIDs.

4. **ifcidNo** (input)

   The number of IFCIDs specified with parameter **ifcids**, or `0`, if no IFCIDs are specified.

5. **timestampFrom** (input)

This parameter specifies a time stamp (in Store Clock format) that limits the amount of data returned. The list of snapshots returned contains snapshots that are taken *after* this time stamp. If you also specify a time stamp for parameter **timestampTo**, keep both in timely order.

If you do not want to specify a time stamp, specify `NULL`. The list of snapshots returned contains all snapshots from the history data set (if not limited by the parameters **ifcids** or **timestampTo**).

6. **timestampTo** (input)

   This parameter specifies a time stamp (in Store Clock format) that limits the amount of data returned. The list of snapshots returned contains snapshots that are taken *before* this time stamp.

   If you do not want to specify a time stamp, specify `NULL`. The list of snapshots returned contains all snapshots from the history data set (if not limited by the parameters **ifcids** or **timestampFrom**).

7. **result** (output)

   Pointer to the output data area. See "Working with Returned Data" on page 36 for how to retrieve individual counter values. Use the **freeHashTable()** function to release the memory area, if the output data is no longer needed.

   If no snapshots are in the history data set that match the selection criteria, the output data area is empty (no REPHISNP repeating block is available). The function returns with a return code of `0`.

result	REPHISNP	Groups all history data set snapshots.		
		HISTTIME	Time stamp of history data set snapshot, in Store Clock format.	
		REPHIIFI	Groups all IFCIDs collected at HISTTIME. Contains one repeating block per HISTTIME.	
			HISTIFI	IFCID collected at HISTTIME.

History Processing - Get History Data

Function Call

```
pmReturnCodes pmGetHistory (pmHost*          handle,
                            char*            workProfile,
                            char**           fields,
                            unsigned int     counterNo,
                            pmQualifierList* qualifier,
                            pmTOD            requestTime,
                            pmDirection      dir,
                            pmTOD            snapshotTime,
                            pmHashTable      result)
```

Header File

pmGetStatThread.h

Description

This function requests the Data Collector to retrieve a stored snapshot of DB2 performance data from the history data set for a user specified by parameter **workProfile**.

Counters can be specified and qualified as with the **pmGetSnapshotEx()** function.

Note that the history data set contains only snapshots of IFCIDs and threads that were specified and qualified as Data Collector startup parameters.

This function requires that the output data area is initialized by the **clearHashTable()** function.

Parameters

1. **handle** (input)

   The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.

2. **workProfile** (input)

   A user's work profile for which this function is to be executed. The user was specified by the **pmLogon()** function.

3. **fields** (input)

   An array of counter names for which counter data is requested from a stored snapshot record in a history data set.

   Specify at least one counter name. Counter names are not case-sensitive. Invalid counter names are ignored. Counters not available in the stored snapshot record in a history data set are ignored. No warning is returned.

   ```
   char *fields[] = {
                      "QISEDBD",  "QTMAXDS", "QTAUCHK",  "QTAUSUC",  "QBSTMAX",
                      "QBSTPID",  "QBSTGET", "QBSTWFT",  "QBSTWFD",  "QBSTWKPD",
                      "QBSTWKPD", "QBSTMAX", "QBSTWKPD", "QBSTWDRP", "QBSTWBVQ",
                      "QBSTWFR",  "QBSTWFF", "QBSTWFD",  "QISEDSI",  "QISEDSG",
                      "QISEDSC",  "QXSTFND", "QXSTNFND", "QBSTWDRP", "QBSTWBVQ"
                    };
   ```

4. **counterNo** (input)

   The number of counters that are specified with parameter **fields** (> 0).

5. **qualifier** (input)

   The name of a qualifier list. The list specifies the criteria which data the Data Collector should return. See "How Qualifying Works" on page 33, if required.

6. **requestTime** (input)

   Use this parameter together with parameter **dir** to locate a stored snapshot record in the history data set.

   - If you want to locate a stored snapshot record nearest to a specified point in time (and younger than the given time), set **requestTime** to the point in time you want and parameter **dir** to `TO`.
   - If you want to locate the oldest stored snapshot record, set **requestTime** to `TOD_FIRST` and **dir** to `FORWARD`.
   - If you want to locate the youngest stored snapshot record, set **requestTime** to `TOD_LAST` and parameter **dir** to `BACK`.
   - If you want to locate a stored snapshot record adjacent to a previously located record, set **requestTime** to the time returned by the previous **pmGetHistory()** function call. See parameter **snapshotTime** below. Use parameter **dir** to choose the direction (the older or the younger record adjacent to this point in time).

   See "Working with the History Data Set" on page 34 if you need a more detailed description.

   The time stamp for the **requestTime** parameter is the Store Clock format. If you need a conversion from **time_t** format, see "Convert time_t Format to Store Clock Format" on page 93 for details.

7. **dir** (input)

Use this parameter together with parameter **requestTime** to locate a stored snapshot record in the history data set.

- Use `TO` together with a given point in time (specified by parameter **requestTime**), if you want to locate a stored snapshot record nearest to a point in time. Per definition the younger snapshot in the history data set is chosen.
- Use `BACK` if you want to choose the older snapshot adjacent to a previously located snapshot.
- Use `FORWARD` if you want to choose the younger snapshot adjacent to a previously located snapshot.

8. **snapshotTime** (output)

   The time stamp of a stored snapshot retrieved from the history data set.

   The time stamp is in Store Clock format. If you need a conversion to **time_t** format, see "Convert Store Clock Format to time_t Format" on page 91 for details.

9. **result** (output)

   Pointer to the output data area. See "Working with Returned Data" on page 36 for how to retrieve individual counter values. Use the **freeHashTable()** function to release the memory area, if the output data is no longer needed.

Processing DB2 Exception Events

Introduction to Exception Processing

The DB2 PM API provides functions that you can use in the application to work with DB2 exception events. The following DB2 event types are supported:

- Authorization failure
- Timeout
- EDM Pool full
- Global Trace started
- Thread commit indoubt
- Deadlock
- Data set extension
- Unit of recovery inflight or indoubt
- Logspace shortage
- Coupling Facility events:
  - CF rebuild start
  - CF rebuild stop
  - CF alter start
  - CF alter stop

Exception processing through the API requires that the Data Collector is started with event exception processing or periodic exception processing enabled. The API can access exception events only if the Data Collector records this data. Note that the Data Collector can be started with event exception processing enabled, periodic exception processing enabled, none, or both enabled. However, the API currently has access only to event exception data.

The Data Collector records the 500 most recent DB2 authorization events and other DB2 events in an exception log. The application program can retrieve this exception log and individual log records for further processing.

The application can also request the Data Collector to post event exceptions to the application program as they occur.

The application program controls the event exception processing in the Data Collector with the following API functions:

- The **pmGetEventExceptionLog()** function retrieves the exception log from the Data Collector. You can specify a time frame to limit the number of exception log records that are returned to the application.
- The **pmGetEventDetails()** function retrieves all details about an individual exception log record. You identify an exception log record by a time stamp and an event exception type. Both are returned by a previous **pmGetEventExceptionLog()** function call, and a **pmFetchExceptions()** function call.
- The **pmStartExceptionProc()** function starts event exception processing in the Data Collector. This prepares the Data Collector for posting event exceptions to the application. Parameters allow you to specify the type of event exceptions to process. Each event exception process is assigned to an existent user profile (specified by the **pmLogon()** function at logon time). One event exception process can be assigned to one user's work profile.
- The **pmGetExceptionStatus()** function requests status information about an exception process for a specified user. You use this function to check whether a process is already started, or to gather details about a process.
- The **pmFetchExceptions()** function requests the Data Collector to post exception records to the application program as they occur.

  You use this function to keep track with exception events. The exception records that are returned to the application program are similar to those returned by the **pmGetEventExceptionLog()** function. Therefore, use the **pmGetEventDetails()** function to retrieve all details about an individual exception log record.

  Subsequent **pmFetchExceptions()** function calls request the Data Collector to post yet undelivered exception records. If no new exception records exist, this function waits until the next exception occurs.
- Finally, the **pmStopExceptionProc()** function stops event exception processing for a named user profile.

  Event exception processing also stops if the named user is logged off from the Data Collector.

  However, event exception processing remains active if the application disconnects the named user from the Data Collector. After the application reconnects to the Data Collector, the named user has again access to exceptions that occurred after the **pmFetchExceptions()** function was used last.

Note: The following functions contain provisions for possible future extensions. Therefore, some parameters must be set to fixed values, and some output variables return fixed values.

Retrieve Event Exception Log

Function Call

```
pmReturnCodes pmGetEventExceptionLog (pmHost*       handle,
                                      char*         workProfile,
                                      pmTOD         from,
```

```
pmTOD         to,
char**        data,
unsigned int* length)
```

Header File

pmExcpProc.h

Description

This function requests the Data Collector to retrieve the exception log for a user specified by parameter **workProfile**.

At the most, the Data Collector returns 500 of the most recent exception log records. You can specify a time frame to limit the number of exception log records to be returned.

Parameters

1. **handle** (input)

   The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.

2. **workProfile** (input)

   A user's work profile for which this function is to be executed. The user was specified by the **pmLogon()** function.

3. **from** (input)

   This parameter specifies the earliest date and time (in Store Clock format) for which exception log records should be retrieved. If **from** is `NULL`, the exception log is read from the beginning.

   The **from** parameter requires the Store Clock format. If you need a conversion from **time_t** format, see "Convert time_t Format to Store Clock Format" on page 93 for details.

4. **to** (input)

   This parameter specifies the latest date and time (in Store Clock format) for which exception log records should be retrieved. If **to** is `NULL`, the exception log is read to the end.

   The **to** parameter requires the Store Clock format. If you need a conversion from **time_t** format, see "Convert time_t Format to Store Clock Format" on page 93 for details.

5. **data** (output)

   Pointer to the output data area. The returned data is in a DB2 PM specific format,[9] so use the parsing functions (described in "Parsing Data" on page 83) to extract data. The "Example" on page 59 shows how this can be done. The field table describes the exception IDs returned by this function in section "Exception IDs". Use the **pmFreeMem()** function to release the memory area, if the output data is no longer needed.

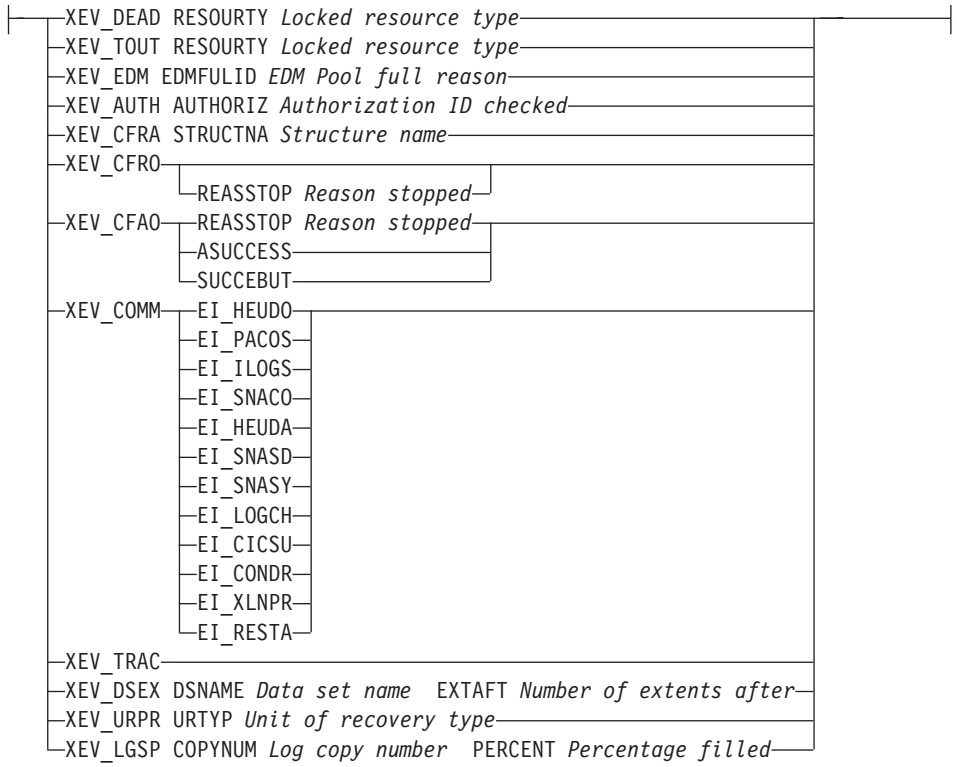
   Output Data Area for pmGetEventExceptionLog()

   ```
   ►►—total length of output area—CNT_X number—DATEATIM date and time————————►
   ```

9. This format is not affected by any workstation-specific character encoding. The parsing functions ensure the appropriate character conversion, based on the code page or character set in use.

Event exception record:

Event information:



CNT_XEVT
The value indicates the number of event exceptions following in the list.

RC_XEVT
Specifies the beginning of an event exception record.

CNT_XEVT
The value indicates the number of event exceptions following in the list.

RC_XEVT
Specifies the beginning of an event exception record.

RC_XEVTE
Specifies the end of an event exception record.

EI_HEUDO
Heuristic decision occurred.

EI_PACOS
Partner cold start detected.

EI_ILOGS
Incorrect logname or syncpoint parm.

EI_SNACO
SNA compare stats protocol error.

EI_HEUDA
Heuristic damage.

EI_SNASD
SNA syncpoint protocol damage.

EI_SNASY
Syncpoint communication failure.

EI_LOGCH
Logname changed on warm/start.

EI_CICSU
CICS or IMS NID unknown.

EI_CONDR
Conditional restart data loss.

EI_XLNPR
XLN protocol error.

EI_RESTA
Error during DB2 restart.

6. **length** (output)

   The length (in number of bytes) of the output data area.

Example

```
#include "pmExcpProc.h"
#include "pmParser.h"
#include "pmCounter.h"
#include "pmConnect.h"
#include "pmLogOnOff.h"
#include "pmTOD.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
...

pmHost         myHandle;
char           workProfile[] = "PMUSER          DB2PM           10.0.0.1:0001   ";
pmReturnCodes  error;
char*          data;
char*          helpPtr;
unsigned int   length,
               pos = 0,
               counterNo;
pmCounter      aCounter;
time_t         eventTime;
pmBoolean skipped;

// connect to data collector and log on
...

// start event exception processing
error = pmStartExceptionProc(&myHandle, workProfile, EVENT, 0, FALSE, NULL);
```

```
// ok, now get the complete event exception log
// from log start to log end (from and to set to NULL)
if(pmGetEventExceptionLog (&myHandle, workProfile, NULL, NULL,
                           &data, &length).returnCode == 0)
{
  helpPtr = data + 4;    /* helpPtr points now to output data area  */
                         /* without 'total length of output' area   */
                         /* field.                                  */
  length -= 4;           /* output data area length without         */
                         /* 'total length of output' area field     */

  /* get number of returned exceptions (CNT_X)                      */
  if(nextTokenValue (&myHandle, helpPtr, &pos,
                     length, "CNT_X", &counterNo) != PM_FAILED)
  {
    printf("Found %d exceptions:\n", counterNo);

    /* skip date and time of pmGetEventExceptionLog() execution     */
    /* (first DATEATIM counter)                                     */
    skipToken(&myHandle, helpPtr, &pos, length, FALSE);

    /* now handle all returned exceptions                           */
    while(counterNo--)
    {
      /* skip start of Event Exception Record (RC_XEVT), it is just */
      /* a flag to indicate start                                   */
      skipToken(&myHandle, helpPtr, &pos, length, FALSE);

      /* now we are at the start of the event information ->        */
      /* get type of event exception                                */
      aCounter = nextToken(&myHandle, helpPtr, &pos, length, FALSE);
      if(aCounter.counterID == 0)
      {
         printf("Parsing failed, data stream is invalid.\n");
         counterNo = 0;
         break;               /* counter invalid                  */
      }


      /* handle all kinds of event exceptions                       */
      if(!strcmp(aCounter.name, "XEV_DEAD"))
      {
        /* deadlock event                                           */
        printf("    >> \'Deadlock\' exception from");
        skipToken(&myHandle, helpPtr, &pos, length, FALSE);
      }
      else if(!strcmp(aCounter.name, "XEV_TOUT"))
      {
        /* timeout event                                            */
        printf("    >> \'Timeout\' exception from");
        skipToken(&myHandle, helpPtr, &pos, length, FALSE);
      }
      else if(!strcmp(aCounter.name, "XEV_EDM"))
      {
        /* EDM pool full event                                      */
        printf("    >> \'EDM Pool full\' exception from");
        skipToken(&myHandle, helpPtr, &pos, length, FALSE);
      }
      else if(!strcmp(aCounter.name, "XEV_AUTH"))
      {
        /* authentification failure event                           */
        printf("    >> \'Authentification Failure\' exception from");
        skipToken(&myHandle, helpPtr, &pos, length, FALSE);
      }
      else if(!strcmp(aCounter.name, "XEV_CFRA"))
      {
```

```
  /* CF rebuild start event                                    */
  printf("    >> \'CF rebuild start\' exception from");
  skipToken(&myHandle, helpPtr, &pos, length, FALSE);
}
else if(!strcmp(aCounter.name, "XEV_CFRO"))
{
  /* CF rebuild stop event                                     */
  printf("    >> \'CF rebuild stop\' exception from");
  if(testToken("REASSTOP", helpPtr, pos, length) == PM_OK)
    skipToken(&myHandle, helpPtr, &pos, length, FALSE);
}
else if(!strcmp(aCounter.name, "XEV_CFAO"))
{
  /* CF alter event                                            */
  printf("    >> \'CF alter\' exception from");
  skipToken(&myHandle, helpPtr, &pos, length, FALSE);
}
else if(!strcmp(aCounter.name, "XEV_COMM"))
{
  /* commit in-doubt event                                     */
  printf("    >> \'Commit in-doubt\' exception from");
  skipToken(&myHandle, helpPtr, &pos, length, FALSE);
}
else if(!strcmp(aCounter.name, "XEV_TRAC"))
{
  /* trace started event                                       */
  printf("    >> \'Global Trace Started\' exception from");
}
else if(!strcmp(aCounter.name, "XEV_DSEX"))
{
  /* trace started event                                       */
  printf("    >> \'Data Set Extention\' exception from");
  skipToken(&myHandle, helpPtr, &pos, length, FALSE);
  skipToken(&myHandle, helpPtr, &pos, length, FALSE);
}
else if(!strcmp(aCounter.name, "XEV_URPR"))
{
  /* trace started event                                       */
  printf("    >> \'Unit of Recovery inflight or indoubt\' exception from");
  skipToken(&myHandle, helpPtr, &pos, length, FALSE);
}
else if(!strcmp(aCounter.name, "XEV_LGSP"))
{
  /* trace started event                                       */
  printf("    >> \'Logspace shortage\' exception from");
  skipToken(&myHandle, helpPtr, &pos, length, FALSE);
  skipToken(&myHandle, helpPtr, &pos, length, FALSE);
}
else
{
  printf("Unknown event exception found, skipping...\n");
  skipped = PM_OK;
  while(testToken("DATEATIM", helpPtr, pos, length) ==
                PM_FAILED && skipped == PM_OK)
    skipped = skipToken(&myHandle, helpPtr, &pos, length, FALSE);
  if(skipped == PM_FAILED)
  {
    printf("Data stream invalid!\n");
    counterNo = 0;
    break;               /* counter invalid                  */
  }
}
/* free internal used memory of counter                        */
deleteCounter(aCounter);

/* get time of event                                           */
```

```
        aCounter = nextToken(&myHandle, helpPtr, &pos, length, FALSE);
        if(aCounter.counterID == 0)
        {
          printf("Parsing failed, data stream is invalid.\n");
          counterNo = 0;
          break;               /* counter invalid                    */
        }
        tod2time(aCounter.value, &eventTime);
        printf(" %s", ctime(&eventTime));
        deleteCounter(aCounter);

        /* skip end of event record token                             */
        skipToken(&myHandle, helpPtr, &pos, length, FALSE);
      }
    }
    else
    {
      printf("Parsing failed, data stream is invalid.\n");
    }
    pmFreeMem(data);

  }

  // log off and disconnect (stops also event exception processing)
  ...
```

Retrieve Event Exception Details

Function Call

```
pmReturnCodes pmGetEventDetails (pmHost*          handle,
                                 char*            workProfile,
                                 pmTOD            eventTime,
                                 char*            eventType,
                                 char**           data,
                                 unsigned int*    length)
```

Header File

pmExcpProc.h

Description

This function requests the Data Collector to retrieve detailed information about an event exception. The exception log record to retrieve is specified by the time stamp and the event exception type that was returned by a previous **pmGetEventExceptionLog()** or **pmFetchExceptions()** function call. The information returned depends on the event exception type.

Parameters

1. **handle** (input)

   The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.

2. **workProfile** (input)

   A user's work profile for which this function is to be executed. The user was specified by the **pmLogon()** function.

3. **eventTime** (input)

   The time (in Store Clock format) at which the event exception occurred. Use the content of field **DATEATIM** returned by a previous **pmGetEventExceptionLog()** or **pmFetchExceptions()** function call.

4. **eventType** (input)

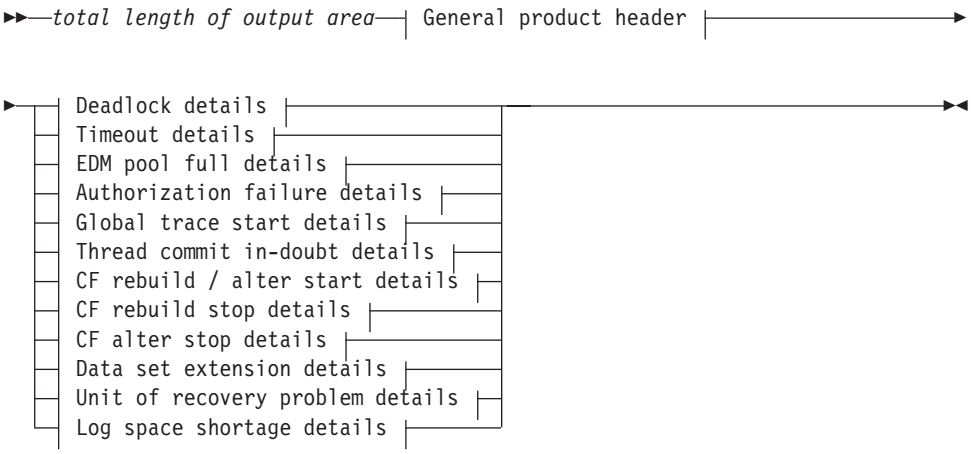
The event exception type. Use the event exception type ("XEV_xxxx") returned by a previous **pmGetEventExceptionLog()** or **pmFetchExceptions()** function.

For example, "XEV_DEAD" specifies a "deadlock" event. See the output data area description of the **pmGetEventExceptionLog()** function for possible event exception types.

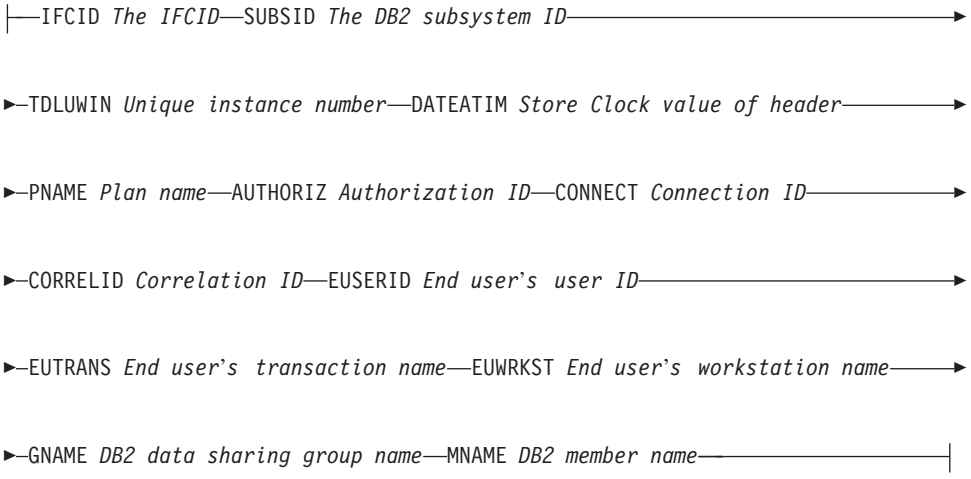
5. **data** (output)

   Pointer to the output data area. The returned data is in a DB2 PM specific format,[10] so use the parsing functions (described in "Parsing Data" on page 83) to extract data. The "Example" on page 59 shows how this can be done. The field table describes the exception IDs returned by this function in section "Exception IDs". Use the **pmFreeMem()** function to release the memory area, if the output data is no longer needed.

Output Data Area for pmGetEventDetails()

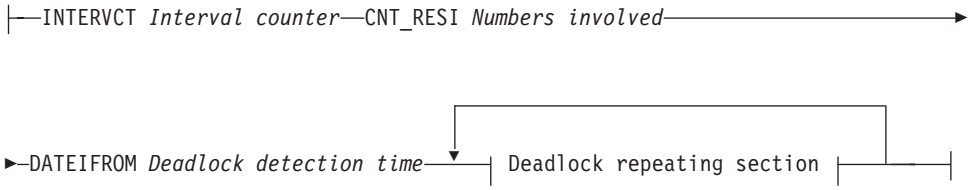


General product header:

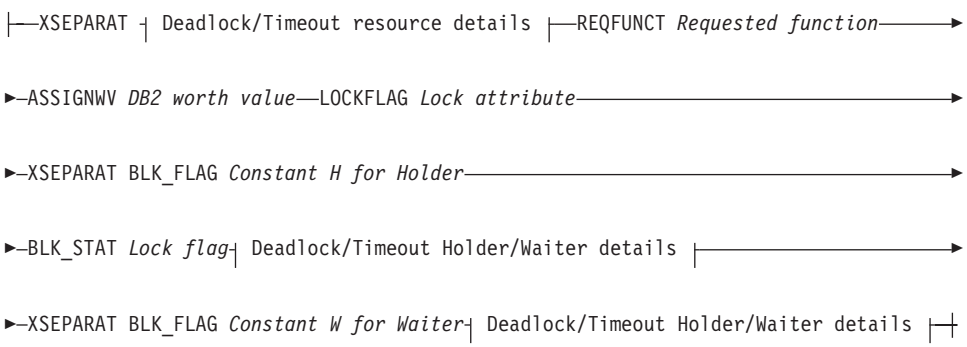


10. This format is not affected by any workstation-specific character encoding. The parsing functions ensure the appropriate character conversion, based on the code page or character set in use.

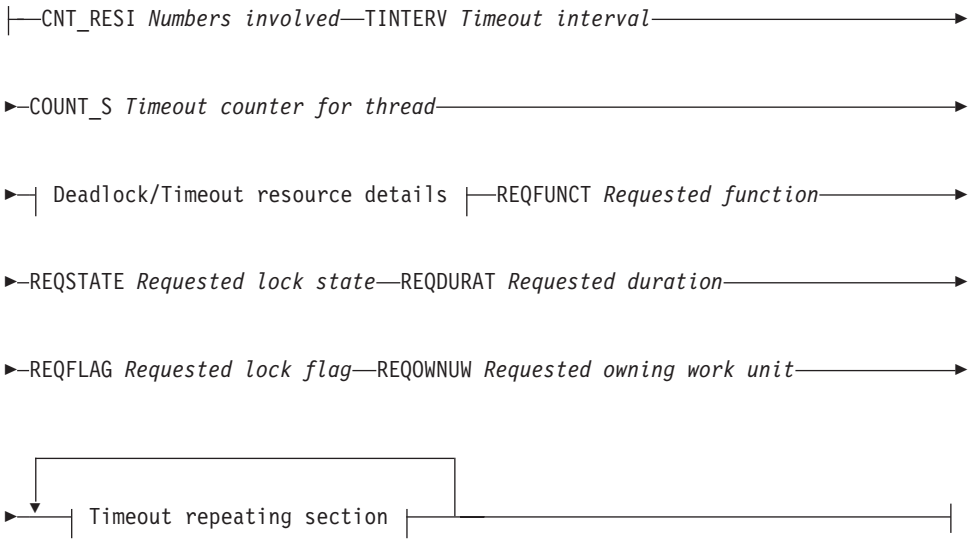
Deadlock details:



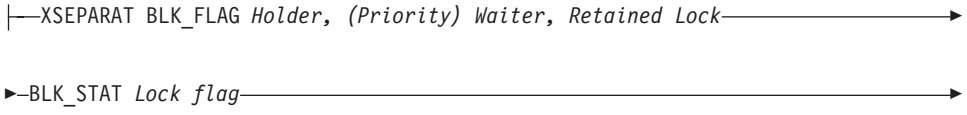
Deadlock repeating section:



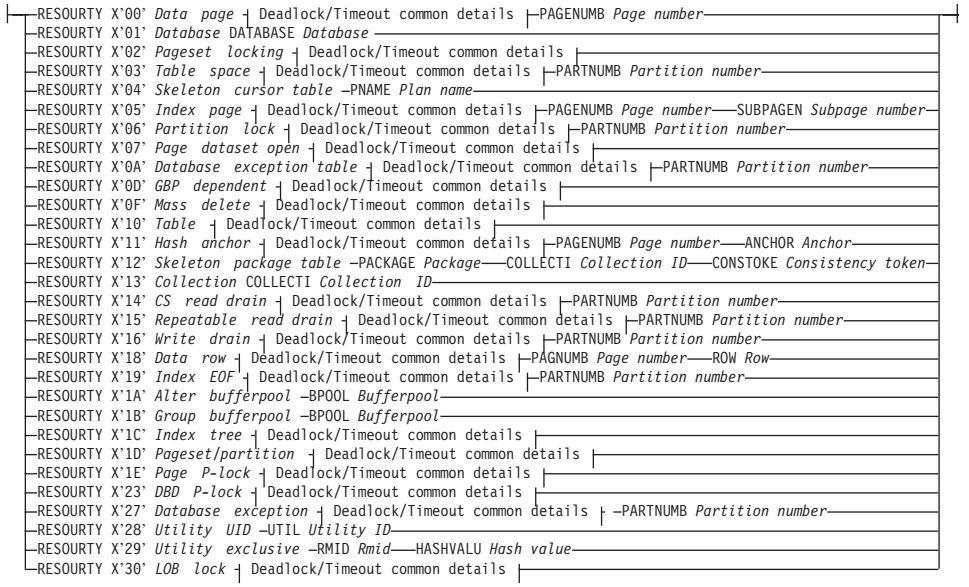
Timeout details:



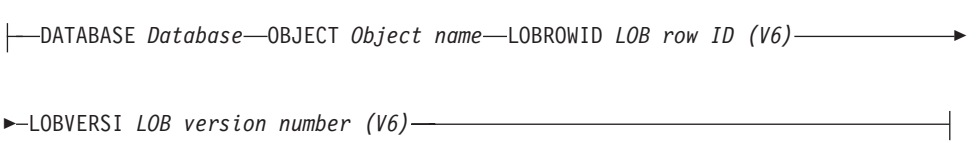
Timeout repeating section:



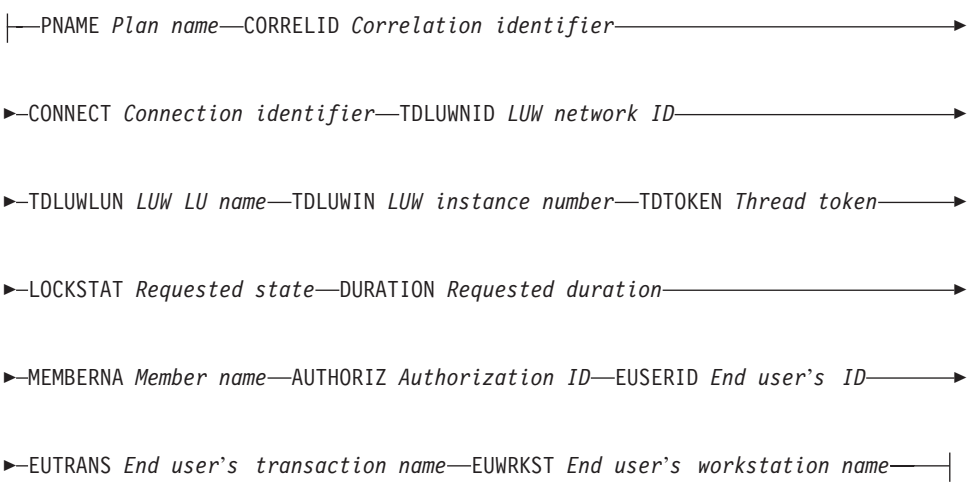
Deadlock/Timeout resource details:



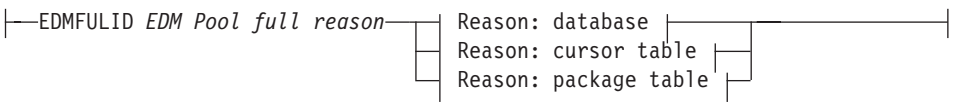
Deadlock/Timeout common details:



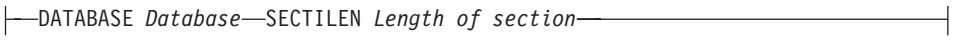
Deadlock/Timeout Holder/Waiter details:



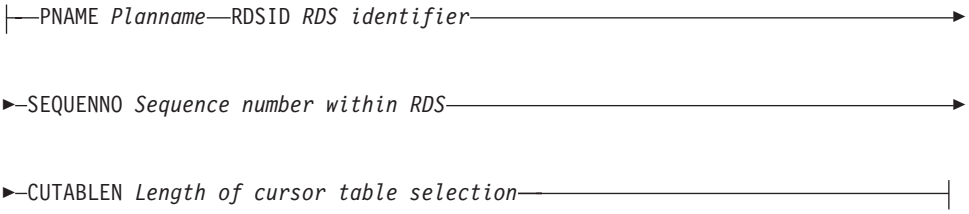
EDM pool full details:



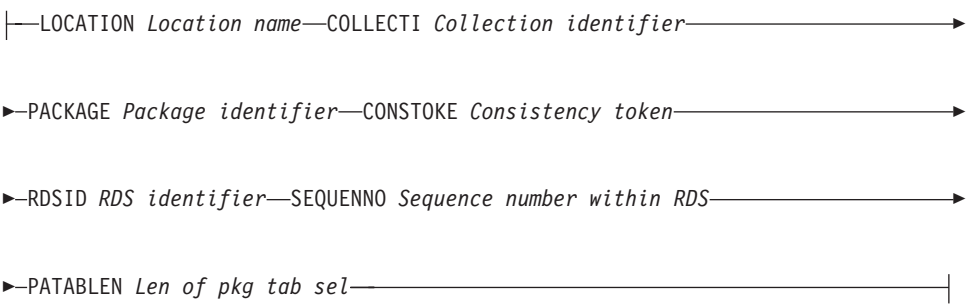
Reason: database:



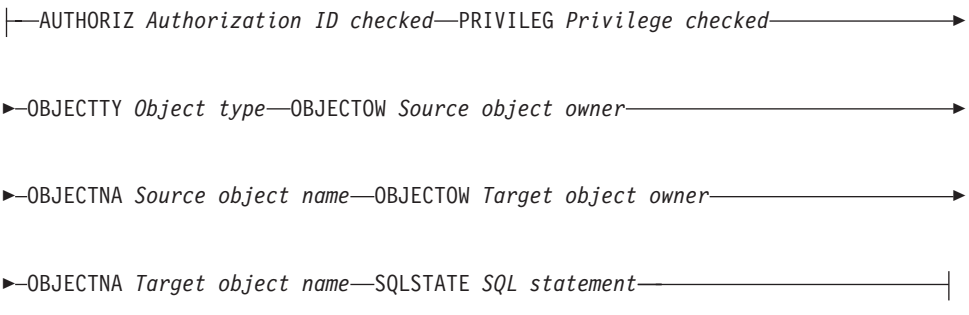
Reason: cursor table:



Reason: package table:

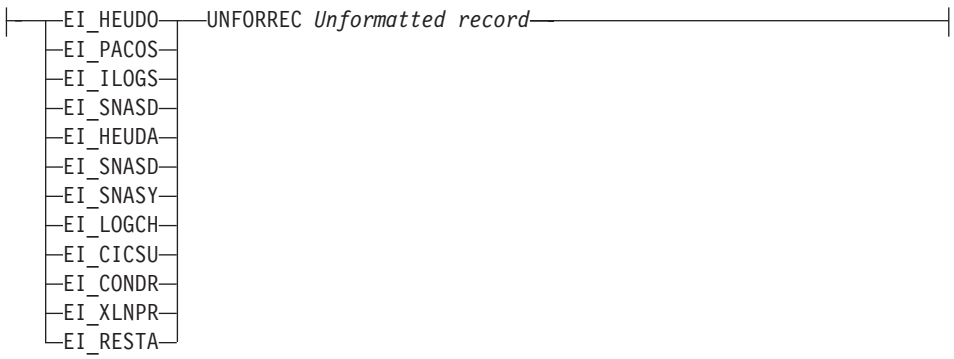


Authorization failure details:

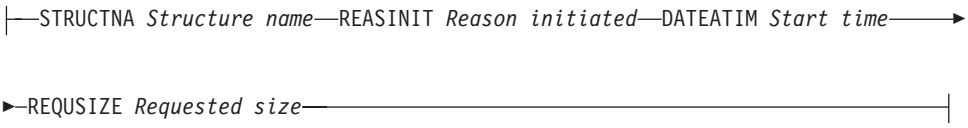


Global trace start details:

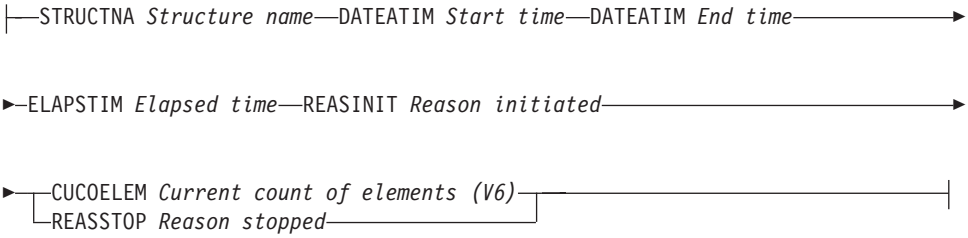
Thread commit in-doubt details:



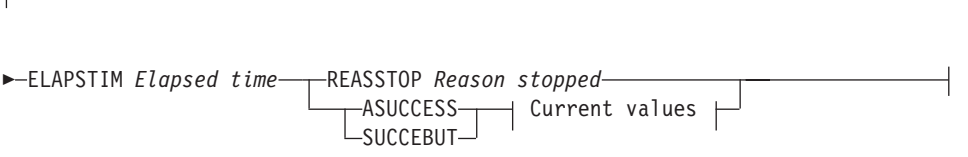
Coupling facility rebuild / alter start details:



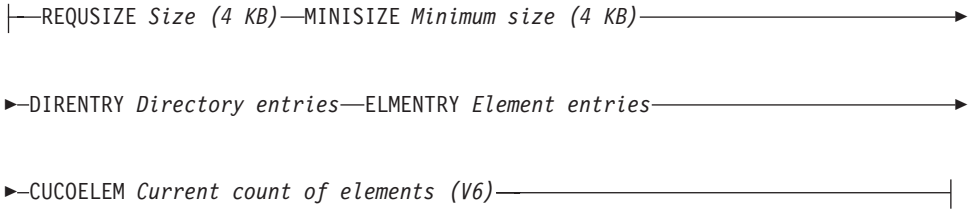
Coupling facility rebuild stop details:



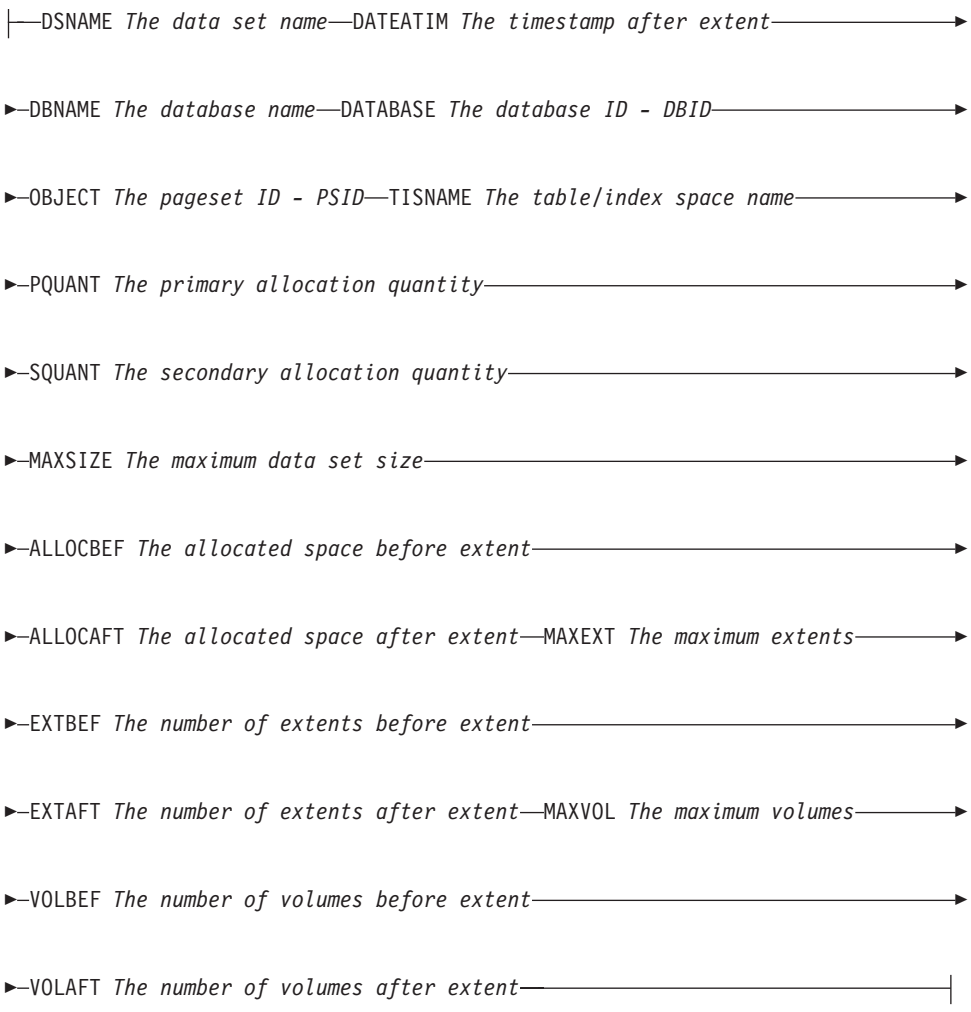
Coupling facility alter stop details:



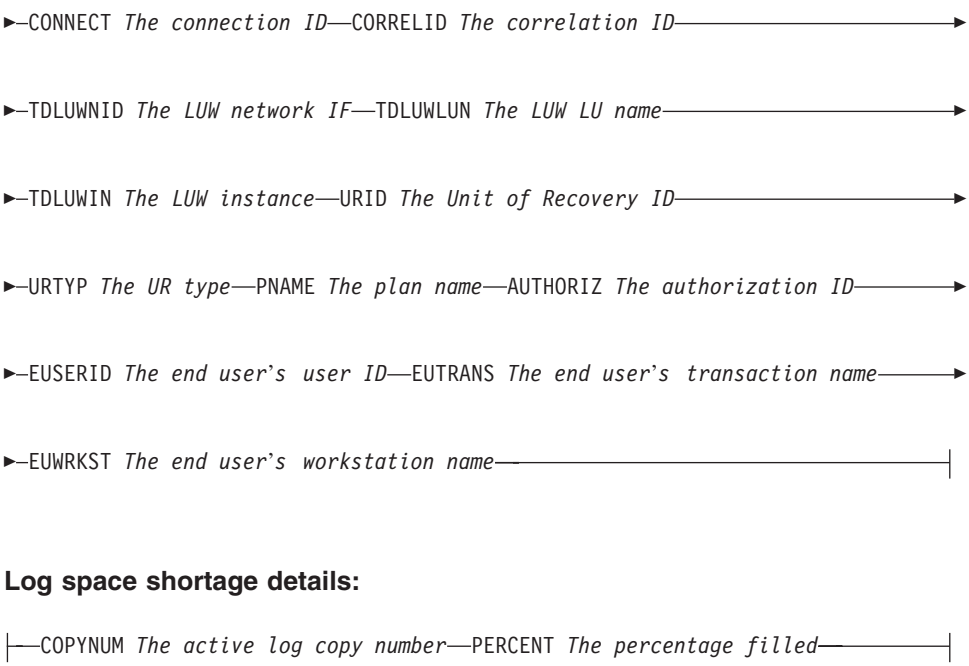
Current values:



Data set extension details:



Unit of recovery problem details:



Log space shortage details:

6. **length** (output)
   The length (in number of bytes) of the output data area.

Example

```
#include "pmExcpProc.h"
#include "pmParser.h"
#include "pmCounter.h"
#include "pmConnect.h"
#include "pmLogOnOff.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
...

pmHost         myHandle;
char           workProfile[] = "PMUSER          DB2PM           10.0.0.1:0001   ";
pmReturnCodes  error;
char*          data;
char*          helpPtr;
unsigned int   length,
               pos = 0,
               counterNo;
pmCounter      aCounter;
pmTOD          evTime;
char           evType[256];

// connect to data collector and log on
...

// start event exception processing
error = pmStartExceptionProc(&myHandle, workProfile, EVENT, 0, FALSE, NULL);

// fetch exception
error = pmFetchExceptions (...);

// parse it and store time and type of event exception
...
```

```
evTime = ...
evType = ... /* 'XEV_DEAD', 'XEV_AUTH', 'XEV_COMM', etc.             */
// ok, now get the details for this event
if(pmGetEventDetails(&myHandle, workProfile, evTime,
                     evType, &data, length).returnCode == 0)
{
  helpPtr = data + 4;    /* helpPtr points now to output data area    */
                         /* without 'total length of output' area     */
                         /* field.                                    */
  length -= 4;           /* output data area length without           */
                         /* 'total length of output' area field       */

  /* use parser functions to get values                               */
  ...

  /* do not forget to free output data area                           */
  pmFreeMem(data);
}

// log off and disconnect (stops also event exception processing)
...
```

Start Exception Processing

Function Call

```
pmReturnCodes pmStartExceptionProc (pmHost*         handle,
                                    char*           workProfile,
                                    pmExceptionType type,
                                    unsigned int    interval,
                                    pmBoolean       userExit,
                                    char*           thresholdDefs)
```

Header File

pmExcpProc.h

Description

This function requests the Data Collector to start event exception processing for a user specified by parameter **workProfile**.

This function is required before the **pmFetchExceptions()** function is used in the application program.

This function should not be called more than once for the same event type and for the same user. Use the **pmGetExceptionStatus()** with the same work profile to check in advance whether exception processing is already started for a specified user.

Parameters

1. **handle** (input)

   The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.

2. **workProfile** (input)

   A user's work profile for which this function is to be executed. The user was specified by the **pmLogon()** function.

3. **type** (input)

   The type of exception processing to start. Specify `EVENT`.

4. **interval** (input)

Reserved. Specify 0.

5. **userExit** (input)

   Reserved. Specify FALSE.

6. **thresholdDefs** (input)

   Reserved. Specify NULL.

Example

```
#include "pmExcpProc.h"
#include "pmConnect.h"
#include "pmLogOnOff.h"
...

pmHost         myHandle;
char           workProfile[] = "PMUSER          DB2PM           10.0.0.1:0001   ";
pmReturnCodes  error;

// connect to data collector and log on
...

// start event exception processing
error = pmStartExceptionProc(&myHandle, workProfile, EVENT, 0, FALSE, NULL);

// log off and disconnect
...
```

Get Exception Processing Status

Function Call

```
pmReturnCodes pmGetExceptionStatus (pmHost*         handle,
                                    char*           workProfile,
                                    pmExcpInfo*     info)
```

Header File

pmExcpProc.h

Description

This function requests the Data Collector to report the status of exception processing for a user specified by parameter **workProfile**. It returns information about:

- Whether exception processing is already started.
- Which exceptions are enabled in the Data Collector.

  The type of exception events (none, event exceptions only, periodic exceptions only, or both) is specified as Data Collector startup parameter.
- Which type of exception events occurred since exception processing was started.

  The **pmGetExceptionStatus()** function requests the Data Collector to scan the DB2 trace records for the existence of possible exception types to find out which type of exception events occurred.

Parameters

1. **handle** (input)

   The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.

2. **workProfile** (input)

   A user's work profile for which this function is to be executed. The user was specified by the **pmLogon()** function.

3. **info** (output)

   Pointer to the output data area. Some structure members are reserved for future use.

```
typedef struct _excpInfo
{
  pmExceptionType status;                 /* if EVENT, PERIODIC,     */
                                          /* BOTH or NONE are active */
  pmBoolean       userExitActive;         /* host user exit          */
  unsigned int    periodicExcpInterval;   /* interval in seconds     */
                                          /* when PM checks for      */
                                          /* periodic exceptions     */
  pmBoolean       perExcpHappened;        /* periodic excp happened  */
  pmBoolean       evExcpHappened;         /* event excp happened     */
                                          /* since logoff            */
  pmBoolean       deadlockActive,         /* indicates that the      */
                  timeoutActive,          /* corresponding events    */
                  EDMPoolActive,          /* are observed            */
                  authFailureActive,
                  traceActive,
                  datasetExtentActive,
                  unitOfRecoveryActive,
                  logspaceShortageActive,
                  commitActive,
                  rebuildActive;
  unsigned short  datasetExtentValue;     /* number of extents before*/
                                          /* datasetExtent event is  */
                                          /* triggered               */
} pmExcpInfo;
```

   - *status* indicates the type of exception processing that was started for this user. It contains:
     - `EVENT` for event exception processing
     - `PERIODIC` for periodic exception processing that is shown here if started from the DB2 PM Online Monitor.
     - `BOTH` for event exception and periodic exception processing
     - `NONE` if no exception processing was started.
   - *userExitActive* indicates whether a DB2 PM user exit routine is active. This can limit the access of the application to some data.
   - *periodicExcpInterval*. Reserved for future use.
   - *perExcpHappened*. Reserved for future use.
   - *evExcpHappened* indicates whether event exceptions occurred since the last **pmFetchExceptions()** function call for the user specified by parameter **workProfile**. If no **pmFetchExceptions()** function was called, it indicates whether event exceptions occurred since exception processing was started for this user.
   - *deadlockActive ... rebuildActive* indicate which events are observed.
   - *datasetExtentValue* shows the number of data set extents to occur before DB2 triggers a corresponding event. This number is a Data Collector startup parameter.

   Boolean variables are returned as either `TRUE` or `FALSE`.

Example

```
#include "pmExcpProc.h"
#include "pmConnect.h"
#include "pmLogOnOff.h"
...

pmHost        myHandle;
```

```
char           workProfile[] = "PMUSER          DB2PM           10.0.0.1:0001   ";
pmReturnCodes  error;
pmExcpInfo     info;

// connect to data collector and log on
...

// get exception processing information
error = pmGetExceptionStatus(&myHandle, workProfile, &info);

// log off and disconnect
...
```

Fetch Exceptions

Function Call

```
pmReturnCodes pmFetchExceptions (pmHost*           handle,
                                 char*             workProfile,
                                 unsigned int      exceptionNo,
                                 char**            data,
                                 unsigned int*     length)
```

Header File

pmExcpProc.h

Description

This function requests the Data Collector to post exception records to the application as they occur. Exception records are posted for a user (specified by parameter **workProfile**) for whom exception processing was started by the **pmStartExceptionProc()** function.

This function allows the application to fetch event exceptions immediately when they occur. Each **pmFetchExceptions()** function call returns those exception records that were not yet returned by a previous **pmFetchExceptions()** function call. If no exceptions occurred in the meantime, the function waits until the next event exception occurs.

You can use the **pmGetExceptionStatus()** function to check whether event exceptions occurred since the last time the **pmFetchExceptions()** function was used.

The **pmFetchExceptions()** function does not deliver an exception record twice to the user specified by **workProfile**. If the application needs access to the same exception record more than once, you need to retrieve the exception log from the Data Collector by the **pmGetEventExceptionLog()** function. Then, you can retrieve details about an individual exception log record by the **pmGetEventDetails()** function.

When the **pmFetchExceptions()** function has returned one or more exception records to the application, use the **pmGetEventDetails()** function to request details about individual records.

The **pmFetchExceptions()** function returns an unknown number of exception records to the application. The number of returned exception records depends on the elapsed time between two consecutive **pmFetchExceptions()** function calls and the number of events that occurred in the meantime. This function can return a maximum of 500 exception records. With parameter **exceptionNo** you can control the number of exception records to be returned. If more than the specified number of events occur between two consecutive function calls, only the most recent ones

are returned. You can also control that the **pmFetchExceptions()** function returns only the next exception record that occurs after the function is called. This causes the function to wait for the next event exception.

Parameters

1. **handle** (input)

   The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.

2. **workProfile** (input)

   A user's work profile for which this function is to be executed. The user was specified by the **pmLogon()** function.

3. **exceptionNo** (input)

   Set this parameter to either 0 or a positive number ≤ 500.

   If **exceptionNo** is set to 0, only the exception that occurs after this **pmFetchExceptions()** function call is received by the Data Collector is posted to the application. No previous exception records are posted.

   If **exceptionNo** is set to a number > 0, all exception records (up to the specified number) not yet delivered by a previous **pmFetchExceptions()** function call are immediately returned.

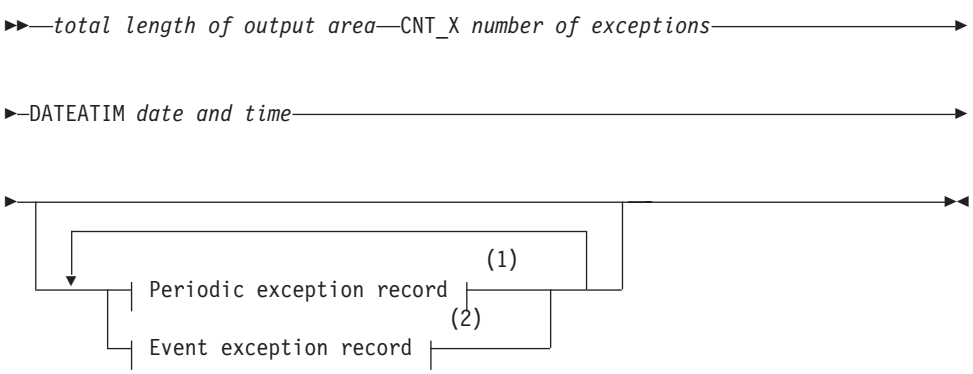
   If no exception record is to be returned, the function waits for the next exception.

   Note: For subsequent **pmFetchExceptions()** function calls you should set **exceptionNo** to the maximum of 500.

4. **data** (output)

   Pointer to the output data area. The returned data is in a DB2 PM specific format,[11] so use the parsing functions (described in "Parsing Data" on page 83) to extract data. The "Example" on page 59 shows how this can be done. The field table describes the exception IDs returned by this function in section "Exception IDs". Use the **pmFreeMem()** function to release the memory area, if the output data is no longer needed.

   ### Output Data Area for pmFetchExceptions()



11. This format is not affected by any workstation-specific character encoding. The parsing functions ensure the appropriate character conversion, based on the code page or character set in use.

Notes:

1. Periodic exceptions are currently not supported by the DB2 PM API.
2. Refer to the **pmGetEventExceptionLog()** function for a detailed description of the event exception record.

5. **length** (output)

   The length (in number of bytes) of the output data area.

Example

```
#include "pmExcpProc.h"
#include "pmParser.h"
#include "pmCounter.h"
#include "pmConnect.h"
#include "pmLogOnOff.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
...

pmHost         myHandle;
char           workProfile[] = "PMUSER          DB2PM           10.0.0.1:0001   ";
pmReturnCodes  error;
char*          data;
char*          helpPtr;
unsigned int   length,
               pos = 0;
pmCounter      aCounter;

// connect to data collector and log on
...

// start event exception processing
error = pmStartExceptionProc(&myHandle, workProfile, EVENT, 0, FALSE, NULL);

// ok, now fetch always the detected event exceptions
// until returnCode <> 0
while(!pmFetchExceptions (&myHandle, workProfile, 0,
                          &data, &length).returnCode)
{
  helpPtr = data + 4;    /* helpPtr points now to output data area    */
                         /* without 'total length of output' area     */
                         /* field.                                    */
  length -= 4;           /* output data area length without           */
                         /* 'total length of output' area field       */

  /* skip number of returned exceptions (CNT_X) because it is '1'     */
  if(skipToken(&myHandle, helpPtr, &pos, length, FALSE) == PM_FAILED) break;


  /* skip date and time of pmFetchExceptions() execution              */
  /* (first DATEATIM counter)                                          */
  if(skipToken(&myHandle, helpPtr, &pos, length, FALSE) == PM_FAILED) break;


  /* skip start of Event Exception Record (RC_XEVT), it is just       */
  /* a flag to indicate start                                         */
  if(skipToken(&myHandle, helpPtr, &pos, length, FALSE) == PM_FAILED) break;

  /* now we are at the start of the event information ->              */
  /* get type of event exception                                      */
  aCounter = nextToken(&myHandle, helpPtr, &pos, length, FALSE);
  if(aCounter.id == 0) break; /* counter invalid                      */

  /* handle all kinds of event exceptions                             */
```

```
  if(!strcmp(aCounter.name, "XEV_DEAD"))
    // deadlock event
    ...

  else if(!strcmp(aCounter.name, "XEV_TOUT"))
    // timeout event
    ...

  else if(!strcmp(aCounter.name, "XEV_EDM"))
    // EDM pool full event
    ...

  else if(!strcmp(aCounter.name, "XEV_AUTH"))
    // authentification failure event
    ...

  /* free internal used memory of counter                          */
  deleteCounter(aCounter);

  /* do not forget to free output data area                        */
  pmFreeMem(data);

  /* reset pos to beginning of data stream                         */
  pos = 0;
}

// log off and disconnect (stops also event exception processing)
...
```

Stop Exception Processing

Function Call

```
pmReturnCodes pmStopExceptionProc (pmHost*         handle,
                                   char*           workProfile,
                                   pmExceptionType type)
```

Header File

pmExcpProc.h

Description

This function requests the Data Collector to stop event exception processing for a user specified by parameter **workProfile**.

After successful completion of this function, the **pmFetchExceptions()** function cannot be used anymore for this user.

This function should be used only if event exception processing was started for this user, otherwise the Data Collector returns a warning.

Parameters

1. **handle** (input)

   The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.

2. **workProfile** (input)

   A user's work profile for which this function is to be executed. The user was specified by the **pmLogon()** function.

3. **type** (input)

   The type of exception processing to be stopped. Specify `EVENT`.

Example

```
#include "pmExcpProc.h"
#include "pmConnect.h"
#include "pmLogOnOff.h"
...

pmHost         myHandle;
char           workProfile[] = "PMUSER          DB2PM           10.0.0.1:0001   ";
pmReturnCodes  error;

// connect to data collector and log on
...

// stop event exception processing
error = pmStopExceptionProc(&myHandle, workProfile, EVENT);

// log off and disconnect
...
```

Executing DB2 Commands

DB2 commands can be executed by means of the DB2 PM API if the SAF user ID issuing the DB2 command has sufficient DB2 privileges. The Data Collector passes the DB2 command over to the instrumentation facility interface (IFI) for execution and returns the command response to the application.

Execute DB2 Command

Function Call

```
pmReturnCodes pmExecDB2CommandEx  (pmHost*      handle,
                                   char*        workProfile,
                                   int          db2CmdParmLevel,
                                   void*        db2CmdParms)
```

This function replaces the **pmExecDB2Command()** function.

Header File

pmExecDB2Command.h

Description

This function requests the Data Collector to execute a DB2 command and to return the command response for a user specified by parameter **workProfile**.

All DB2 commands are allowed, except you cannot request the Data Collector to start or to stop DB2.

The Data Collector might be set up so that it requires a PassTicket with each DB2 command that is to be executed by means of the API. If PassTickets are required, or if you want to apply PassTickets with DB2 commands even if it is not required, you must use the **pmGenPassticket()** function to generate a PassTicket. Then, you use this PassTicket as a parameter with the **pmExecDB2CommandEx()** function. Ensure that you specify the same **workProfile** for both function calls. Note that a PassTicket can be used only once, and only within 10 minutes after it is generated.

If your application works with a data sharing group, the **pmExecDB2CommandEx()** function can request the execution of DB2 commands for selected members of a data sharing group as well as for all members of a group. Ensure that the Data Collector is started with the `DATASHARINGGROUP=YES` startup parameter.

You can use the **pmGetInfo()** function to test whether the application program can successfully use API functions that request data from data sharing group members. Counter QR4DS works as a flag that indicates whether all prerequisites for data sharing group support are met.

To store the response from a DB2 command the API allocates an output data area of up to 1 MB. If the amount of data returned exceeds 1 MB, data is truncated, and the function returns a warning. The output data area holding the command response remains allocated until it is freed by the application.

This function requires that the output data area is initialized by the **clearHashTable()** function.

Parameters

1. **handle** (input)

   The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.

2. **workProfile** (input)

   A user's work profile for which this function is to be executed. The user was specified by the **pmLogon()** function.

3. **db2CmdParmLevel** (input)

   A keyword that specifies the parameter structure to be used in parameter **db2CmdParms**. The following keywords are allowed:

   - `PM_DB2CMD_PASSTICKET` specifies that parameter **db2CmdParms** uses a parameter structure for use in the currently attached DB2 subsystem (no data sharing group support).
   - `PM_DB2CMD_DSG` specifies that parameter **db2CmdParms** uses a parameter structure for use in a DB2 data sharing group.

4. **db2CmdParms** (input/output)

   Pointer to the parameter structure that contains the required parameters for this function call. Note that the same structure holds the returned data from the DB2 command execution after successful completion of this function call.

   - If you have specified `PM_DB2CMD_PASSTICKET` as parameter **db2CmdParmLevel**, use the following parameter structure:

     ```
     typedef struct _pmDB2CmdPassTicket {
       char          *command;       /* the DB2 command to be executed          */
       char          *passTicket;    /* PassTicket used to secure this          */
                                     /* transaction or NULL for unsecured       */
                                     /* transactions                            */
       char         **response;      /* pointer to area where the command       */
                                     /* response is stored                      */
       pmHashTable   *result;        /* output area (e.g. diagn. information)   */
     } pmDB2CmdPassTicket;
     ```

   - If you have specified `PM_DB2CMD_DSG` as parameter **db2CmdParmLevel**, use the following parameter structure:

     ```
     typedef struct _pmDB2CmdDSG {
       char          *command;       /* the DB2 command to be executed          */
       char          *passTicket;    /* PassTicket used to secure this          */
                                     /* transaction or NULL for unsecured       */
                                     /* transactions                            */
       char          *dsMember;      /* execute DB2 command on selected member  */
                                     /* or on connected member if set to NULL   */
       int            dsGlobal;      /* execute DB2 command with group scope if */
                                     /* set to PM_GROUP_SCOPE, or with member   */
                                     /* scope if set to PM_MEMBER_SCOPE         */
     ```

```
  char         **response;      /* pointer to area where the command      */
                                /* response is stored                     */
  pmHashTable   *result;        /* output area (e.g. diagn. information)  */
} pmDB2CmdDSG;
```

The structure members have the following meaning:

- *command* (input) specifies the DB2 command to execute. Precede the command by a hyphen "-". The command can be written in upper- or lowercase, or in mixed case.
- *passTicket* (input) specifies the PassTicket to secure this transaction, or `NULL` for unsecured transactions (if the Data Collector does not require PassTickets).
- *dsMember* (input) and *dsGlobal* (input) specify whether the DB2 command should be applied to a specified member of a data sharing group, or to all members of a data sharing group.

  dsMember (input) specifies the name of a data sharing group member, or `NULL`. If it contains a name, the name must be specified as an 8-character string, left aligned, and padded with blanks, if required.

  dsGlobal (input) specifies the scope of the DB2 command execution and contains one of the two keywords `PM_MEMBER_SCOPE` and `PM_GROUP_SCOPE`.

  The following combinations are allowed:

dsMember **set to ...**	*dsGlobal* **set to ...**	**Scope of DB2 command execution**
Specific name of a data sharing group member.	`PM_MEMBER_SCOPE`	DB2 command is applied to the specified member of the group.
`NULL`	`PM_MEMBER_SCOPE`	DB2 command is applied to the member of the group to which the application is currently connected.
`NULL`	`PM_GROUP_SCOPE`	DB2 command is applied to all members of the group.

- *response* (output) contains a pointer to the output data area in memory where the result of the DB2 command is stored. Use the **pmFreeMem()** function to release the memory area, if the output data is no longer needed.
- *result* (output) contains a pointer to the output data area that may contain diagnostic information about this function call. Use the **freeHashTable()** function to release the memory area, if the output data is no longer needed.

  If diagnostic information is returned, it is contained in the REPDIAG repeating block structure.

result	REPDIAG	Groups all diagnostic information.		
		REPDB2	Groups all DB2 diagnostic information (one block per data sharing group member).	
			DB2RC	DB2 return code
			DB2RS	DB2 reason code
			DB2MBR	Data sharing group member name that caused the DB2 return and reason code.

Currently, information is returned only for the data sharing group member with the highest return code.

Example

```
#include "pmExecDB2Command.h"
#include "pmConnect.h"
#include "pmLogOnOff.h"
#include <stdlib.h>
...
pmHost         myHandle;
char           workProfile[] = "PMUSER DB2PM 10.0.0.1:0001 ";
char*          response = NULL;
pmDB2CmdDSG    db2Parms;
pmReturnCodes  error;
pmHashTable    result;

// connect to data collector and log on
...

// execute command to archive logs with group scope
db2Parms.command = "-ARCHIVE LOG";
db2Parms.passTicket = NULL; /* host allows unrestricted DB2 command execution */
db2Parms.dsMember = NULL;
db2Parms.dsGlobal = PM_GROUP_SCOPE;
db2Parms.response = &response;
db2Parms.result = &result;

clearHashTable(&result);
error = pmExecDB2CommandEx(&myHandle, workProfile,
                           PM_DB2CMD_DSG, db2Parms);

// work with output
...
// don't forget to free memory for DB2 response
if(response) pmFreeMem(response);
freeHashTable(result);

// execute command to start trace on member MBR03
db2Parms.command = "-START TRACE(MON)";
db2Parms.dsMember = "MBR03   ";
db2Parms.dsGlobal = PM_MEMBER_SCOPE;
db2Parms.response = &response
db2Parms.result = &result;

clearHashTable(&result);
error = pmExecDB2CommandEx(&myHandle, workProfile,
                           PM_DB2CMD_DSG, db2Parms);

// work with output
...
// don't forget to free memory for DB2 response
if(response) pmFreeMem(response);
freeHashTable(result);

// log off and disconnect
  ...
```

Saving and Retrieving User Data

The Data Collector provides a central storage area of 1 MB for each user specified by a work profile. You can use this storage area to store and retrieve whatever data you want. The intention for this storage area is to hold user-specific data to support mobile users of the application. When mobile users disconnect from the Data Collector at one workstation and reconnect to it at another workstation without logging off, they need to have a workstation-independent storage to hold, for example, their current application settings.

A 1-MB central storage area has the following characteristics:

- It is created when a user specified by a work profile logs on to the Data Collector.
- It is initialized to X'00.
- It is released when:
  - The user specified by a work profile logs off from the Data Collector.
  - The Data Collector is stopped.
  - An SAF user or SAF group is purged by a DB2 PM operator command.
- It is divided into 256 chunks of 4-KB buffers. Buffers are addressed by buffer numbers of 1 to 256.

The API provides two functions to use these buffers:

- The **pmSaveUserData()** function saves a block of data up to 4 KB in a buffer addressed by a buffer number of 1 to 256.
- The **pmGetUserData()** function retrieves a block of data from a buffer addressed by a buffer number of 1 to 256. Alternatively you can use this function to retrieve the content of all 256 buffers by specifying a buffer number of 0.

Buffers are bound to a work profile. Both functions must be assigned to a work profile when they are called.

Data is saved in a buffer as is. No conversion takes place. However, if less than 4 KB of data is saved, returned data is filled with 0x00 up to a size of 4 KB.

When a **pmGetUserData()** function is called, the API allocates an output data area of 4 KB (or 1 MB, if the contents of all buffers are retrieved) in the workstation's memory to store the returned data. The output data area holding the returned data remains allocated until it is freed by the application.

Save User Data

Function Call

```
pmReturnCodes pmSaveUserData(pmHost*         handle,
                             char*           workProfile,
                             char*           data,
                             unsigned int    length,
                             unsigned short  bufferNo)
```

Header File

pmUserData.h

Description

This function requests the Data Collector to save user-specific application data in a buffer in the Data Collector addressed by buffer number **bufferNo**.

Parameters

1. **handle** (input)

   The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.

2. **workProfile** (input)

   A user's work profile for which this function is to be executed. The user was specified by the **pmLogon()** function.

3. **data** (input)

Pointer to the data in the workstation's memory to be saved.

4. **length** (input)

   The length (in number of bytes) of the data area to save (up to 4 KB). If you save UCS-2 encoded data, the length must also be specified as the number of bytes.

5. **bufferNo** (input)

   Number of the Data Collector buffer. This must be a number from 1 to 256.

Example

```
#include "pmUserData.h"
#include "pmConnect.h"
#include "pmLogOnOff.h"
...

pmHost         myHandle;
char           workProfile[] = "PMUSER          DB2PM           10.0.0.1:0001   ";
char           *userData = ...;
pmReturnCodes  error;

// connect to data collector and log on
...

// store data in buffer
error = pmSaveUserData(&myHandle, workProfile, userData, strlen(userData), 1);

// log off and disconnect
...
```

Get User Data

Function Call

```
pmReturnCodes pmGetUserData(pmHost*          handle,
                            char*            workProfile,
                            unsigned short   bufferNo,
                            char**           data)
```

Header File

pmUserData.h

Description

This function requests the Data Collector to retrieve user-specific application data from the Data Collector that was previously saved by the **pmSaveUserData()** function.

Either the content of a buffer addressed by buffer number **bufferNo** is retrieved, or the contents of all 256 buffers is retrieved from the Data Collector.

Parameters

1. **handle** (input)

   The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.

2. **workProfile** (input)

   A user's work profile for which this function is to be executed. The user was specified by the **pmLogon()** function.

3. **bufferNo** (input)

The number of a 4-KB buffer from which data is to be retrieved. If **bufferNo** is set to 0, the data of all 256 buffers is retrieved.

4. **data** (output)

   Pointer to the output data area in memory. The memory (4 KB or 1 MB) is allocated by the DB2 PM API. Note that each buffer content is filled with 0x00, if less than 4 KB of data was saved. Use the **pmFreeMem()** function to release the memory area, if the output data is no longer needed.

Example

```
#include "pmUserData.h"
#include "pmConnect.h"
#include "pmLogOnOff.h"
...

pmHost         myHandle;
char           workProfile[] = "PMUSER          DB2PM           10.0.0.1:0001   ";
char           *storedUserData;
pmReturnCodes  error;

// connect to data collector and log on
...

// store data in buffer 1
...

// work

...

// get stored data
error = pmGetUserData (&myHandle, workProfile, 1, &storedUserData);

// work with received data
...

// do not forget to free memory if no longer used
pmFreeMem(storedUserData);

// log off and disconnect
...
```

Parsing Data

Introduction to Parsing

The following DB2 PM API functions, described in "Processing DB2 Exception Events" on page 55, return data streams with complex and varying data structures:

- **pmGetEventExceptionLog()**
- **pmGetEventDetails()**
- **pmFetchExceptions()**

These structures vary with the type of event requested and the number of exceptions being monitored.

A data stream consists of a pattern of fixed and variable information units that are called tokens. For example, the **pmGetEventExceptionLog()** function returns a data stream that starts as following:

Output Data Area for pmGetEventExceptionLog()

Here, CNT_X *number* and DATEATIM *date and time* represent two tokens.

The Parsing Functions

The API provides a set of functions that lets you parse a data stream on a token basis after it is stored in memory:

- **nextToken()** returns a counter represented by the current token in the data stream and increments a pointer to point to the next token.
- **nextTokenValue()** tests the current token in the data stream for a specified counter name, extracts and stores the counter value, and increments a pointer to the next token.
- **skipToken()** tests whether the current token is valid and increments a pointer to the next token.
- **testToken()** tests whether the current token represents a specified counter name. (This function does not increment a pointer.)
- **deleteCounter()** is used together with the **nextToken()** function and frees the memory area where the counter structure for a specified counter is stored.

These functions have some common characteristics:

- The **nextToken()**, **nextTokenValue()**, and **skipToken()** functions require the specification of a handle. You should specify the handle that was used with the function that returned the data stream (**pmGetEventExceptionLog()**, **pmGetEventDetails()**, **pmFetchExceptions()**). This ensures that the correct code page is used by these functions when they perform their tests.
- You do not need to specify a work profile because these functions do not communicate with the Data Collector.
- The data stream in the output data area starts with a 4-byte length field, which shows the length (in number of bytes) of the output data area without the preceding 4-byte length field. The parsing functions require that you set a pointer (parameter **tokenBlock**) *after* the 4-byte length field. If you use the pointer of the function that returned the data as a reference, add four bytes to that pointer value to skip the length field.
- All parsing functions (except the **testToken()** function) increment a position pointer to point to the next token in a data stream before a function finishes. A subsequent parsing function uses this position pointer as input parameter to address the adjacent token. To start parsing a data stream you should set the position pointer to 0.

  The **testToken()** function only tests for a specified counter name and returns a *true* or *false* condition. The position pointer is not incremented to allow one of the other parsing functions to work with this token.

The Counter Structure

A counter in the data stream is represented as a structure identical to the one shown in "The Counter Structure" on page 36. Note that the parsing functions do not use structure member *attribute* to validate a counter value. The **nextToken()** function uses the content of the *counterID* member to test for a valid counter. The other parsing functions return with a return code of PM_OK or PM_FAILED.

Working with Parsing Functions

Proper use of the parsing functions ensures that an application using these functions also handles unexpected tokens in the returned data streams.

Unexpected tokens in a data stream can occur, for example, if users of an existing application install API enhancements (new versions, releases, or PTFs) that return additional tokens in the data stream.

You should consider the following coding sequences to ensure that an application continues to work even if the data stream changes to accommodate new functions.

If you expect a unique counter name in one of the tokens in the data stream, the following sequence will let you identify the token and retrieve the counter value:

1. Use the **testToken()** function and specify as parameter the counter name you are expecting in the data stream.

   This function does not increment the position pointer.
2. If the token does not contain the counter name you have specified, the **testToken()** function returns `PM_FAILED`. Use the **skipToken()** function to increment the position pointer to the next token in the data stream. Continue with step 1 to test the next token for the specified name (until the end of the data stream is reached).
3. If the token contains the counter name you have specified, the **testToken()** function returns `PM_OK`. Use the **nextTokenValue()** function to retrieve the counter value from the token. Continue as required by the application's logic.

However, if you have no unique expectation about a counter name but need to react on a variety of counters, apply the following program structure. The **testToken()** function checks the current token for one of the specified counter names; the **nextTokenValue()** function retrieves the corresponding counter value. If the current token does not contain one of the expected counter names, the **skipToken()** function increments the position pointer to the next token.

```
Do While not end of data stream
   {IF testToken() = counter_name_1         /* Does not increment position pointer */
       nextTokenValue() for counter_name_1  /* Retrieve counter value 1 from token */
          ... program logic for counter 1 ...
    ELSE
   {IF testToken() = counter_name_2         /* Does not increment position pointer */
       nextTokenValue() for counter_name_2  /* Retrieve counter value 2 from token */
          ... program logic for counter 2 ...
    ELSE
   {IF testToken() = counter_name_n         /* Does not increment position pointer */
       nextTokenValue() for counter_name_n  /* Retrieve counter value n from token */
          ... program logic for counter n ...
    ELSE
    skipToken()                             /* Increments position pointer         */
                                            /* Continue with next token            */
End of Do While
```

Do not use **nextToken()** together with **skipToken()** as an alternative. Both functions increment the position pointer upon return, and you would miss every other token.

Get Token

Function Call

```
pmCounter nextToken (pmHost*       handle,
                     char*         tokenBlock,
                     unsigned int* pos,
                     unsigned int  length,
                     pmBoolean     attr)
```

Header File

pmParser.h

Description

This function returns a counter represented by the current token in the data stream and increments a pointer to point to the next token.

A counter is valid if member *counterID* in the counter structure contains a value greater than 0.

Use the **deleteCounter()** function to release the memory that was allocated by a **nextToken()** function call.

Note the following with regard to the workstation's character representation, respectively code page in use:

Notes:

1. The counter name in counter structure *pmCounter* is always returned in the appropriate character representation.
2. Counter values of type PMSTRING and PMVARCHAR are always returned in the appropriate character representation. All other types are left unchanged.

Parameters

1. **handle** (input)

   Specify the handle that was used by the function that created the returned data stream.

2. **tokenBlock** (input)

   Pointer to the output data area (following the *total length of output area* field).

3. **pos** (input/output)

   The actual position pointer of the parser in the data stream. This position pointer must be initialized with `0` when you start parsing a data stream. After successful completion of this function, **pos** points to the next token.

4. **length** (input)

   The length (in number of bytes) of the output data area (not including the *total length of output area* field).

5. **attr** (input)

   Reserved. Specify `FALSE`.

Get Token Value

Function Call

```
pmBoolean nextTokenValue(pmHost*       handle,
                         char*         tokenBlock,
                         unsigned int* pos,
```

```
                                unsigned int  length,
                                char*         name,
                                char*         storage)
```

Header File
pmParser.h

Description
This function tests the current token in the data stream for a counter name specified by parameter **name**, stores the counter value in memory location **storage**, and increments a pointer to the next token.

Upon successful execution, the function returns `PM_OK`.

If the function returns `PM_FAILED`, the data stream is not valid, or the specified counter is not found at this position.

Note that counters of type PMVARCHAR and PMPARSEDREPBLOCK are not supported by this function.[12]

Parameters
1. **handle** (input)

   Specify the handle that was used by the function that created the returned data stream.
2. **tokenBlock** (input)

   Pointer to the output data area (following the *total length of output area* field).
3. **pos** (input/output)

   The actual position pointer of the parser in the data stream. This position pointer must be initialized with `0` when you start parsing a data stream. After successful completion of this function, **pos** points to the next token.
4. **length** (input)

   The length (in number of bytes) of the output data area (not including the *total length of output area* field).
5. **name** (input)

   The counter name expected at this position in the data stream.
6. **storage** (output)

   Pointer to the memory location where the counter value is stored. The memory area must be already allocated. Note that two bytes per character are required for UCS-2 encoded strings. Member *length* in the counter structure shows the actual length of the counter value.

Skip Token

Function Call
```
pmBoolean skipToken (pmHost*       handle,
                     char*         tokenBlock,
                     unsigned int* pos,
                     unsigned int  length,
                     pmBoolean     attr)
```

12. This is because the maximum size of the returned counter length is not known in advance. The application would have to allocate memory in size of two times the maximum expected counter length (for UCS-2 encoded strings). If you have a need to retrieve counter values of type PMVARCHAR, you may want to experiment with the **nextToken()** function.

Header File
pmParser.h

Description
This function tests whether the current token is valid and increments a pointer to the next token.

This function skips unconditionally to the next token. Use this function to skip a token in the data stream you are not interested in, or to detect the end of a data stream. As a side effect, this function tests the current token for a valid data stream.

If the data stream is valid, the function returns `PM_OK`, else it returns `PM_FAILED`.

Parameters
1. **handle** (input)

   Specify the handle that was used by the function that created the returned data stream.
2. **tokenBlock** (input)

   Pointer to the output data area (following the *total length of output area* field).
3. **pos** (input/output)

   The actual position pointer of the parser in the data stream. This position pointer must be initialized with `0` when you start parsing a data stream. After successful completion of this function, **pos** points to the next token.
4. **length** (input)

   The length (in number of bytes) of the output data area (not including the *total length of output area* field).
5. **attr** (input)

   Reserved. Specify `FALSE`.

Test Token

Function Call
```
pmBoolean testToken (pmHost*       handle,
                     char*         counterName,
                     char*         tokenBlock,
                     unsigned int  pos,
                     unsigned int  length)
```

Header File
pmParser.h

Description
This function tests whether the current token represents a specified counter name. (This function does not increment a pointer.)

If the specified counter name is found, the function returns `PM_OK`, else it returns `PM_FAILED`.

Parameters
1. **handle** (input)

   Specify the handle that was used by the function that created the returned data stream.
2. **counterName** (input)

The counter name expected at this position in the data stream. The name is specified as a character string or UCS-2 encoded string.

3. **tokenBlock** (input)

   Pointer to the output data area (following the *total length of output area* field).

4. **pos** (input)

   The actual position pointer of the parser in the data stream. This position pointer must be initialized with `0` when you start parsing a data stream.

   After successful completion of this function, **pos** *does not* point to the next token. The position pointer remains unchanged.

5. **length** (input)

   The length (in number of bytes) of the output data area (not including the *total length of output area* field).

Delete Counter

Function Call

```
void deleteCounter (pmCounter counter)
```

Header File

pmCounter.h

Description

This function frees the memory area that was allocated by a **nextToken()** function call.

The output data area where the returned data stream is stored in memory is not freed.

Parameters

1. **counter** (input)

   The counter to be freed.

Converting and Adjusting Dates and Times

Introduction to Date and Time Functions

The Data Collector returns date and time values in a format that must be converted before these values can be processed in a C language program. Further, adjustments to dates and times may be required because the Data Collector returns this data as provided by DB2. DB2 stores dates and times as universal times.[6]

If the application program operates in another time zone than the Data Collector, further date and time adjustment is required.

Format Conversions

Several API functions return counter values from the Data Collector to the application program that represent date and time information. Also, several API functions require input parameters to be sent to the Data Collector that represent date and time information.

A date is usually a time stamp that consists of a calendar date and time (for example, a time stamp of when a history snapshot was taken). A time is usually an

information expressed in hours, minutes, and seconds, with no reference to a calendar date (for example, a counter that reflects an accumulated waiting time of a DB2 process).

Dates and times are presented in different formats:

- The Data Collector represents dates and times in the Store Clock format. More exact, counters of type "Date" or "Time" are presented in Store Clock format. See *ESA/390™ Principles of Operation* about the Store Clock format, if necessary.

  The Data Collector returns affected counters as data type pmTOD, which is defined as:

  ```
  typedef char pmTOD[8];
  ```

- The application program, if written in the C programming language, uses a format that conforms to the ANSI-C standard (**time_t** data type).

The API provides two functions that convert counter values from one format to the other:

- The **tod2time()** function converts a counter value from Store Clock format to *time_t* format.

  You use this function if you want to further process date and time information in your C language program, for example, to display, print, or change a date or time.

- The **time2tod()** function converts a counter value from *time_t* format to Store Clock format.

  You use this function, for example, to edit date and time information as an input parameter for an API function.

Local Time Adjustment

Dates and times in DB2 instrumentation data is stored internally as universal time (UT)[6], no matter in which geography a DB2 system operates. Whenever a program retrieves DB2 instrumentation data, and requires date and time information to be expressed in local time, an adjustment is required.

For example, when a DB2 system at New York, U.S.A., writes an event record at 11:00 a.m. local time, the corresponding entry is 16:00 universal time, because the time difference between New York and Greenwich is −5 hours. If a program operating at New York retrieves this 16:00 universal time entry, it must adjust the time by −5 hours to show again the correct time of the event.

If the application program retrieves date- and time-based counters from the Data Collector, these counters are also expressed in universal time. The Data Collector does not perform a conversion to local times. If the application program requires to show or use a date or time in local time, it must adjust the universal time.

The following API functions ease the calculation of local times:

- The **pmGetInfo()** function provides the time difference (counter QR4TID) between the local time of the host processor (where DB2 and the Data Collector operate) and universal time.
- The **pmAddTOD()** and **pmSubTOD()** functions add and subtract two time values.

Note: If you use the ANSI-C **time()** function to create time stamps for use as input parameters:

- First, adjust the local workstation time to universal time.

- Second, convert the adjusted universal time to the Store Clock format.

Time Zone Adjustment

The Data Collector and the application program may operate in different time zones. For example, suppose that the application operates at Berlin, Germany (UT +1 hour), and monitors a DB2 subsystem at New York, U.S.A. (UT –5 hours). A DB2 event record written at 11:00 a.m. New York local time gets a time stamp of 16:00 universal time. To obtain the time of this event in Berlin local time:

1. Adjust the time stamp (UT) of the event by –5 hours to obtain New York local time (16:00–05:00=11:00).

   Use the information returned by the **pmGetInfo()** function, counter QR4TID, for this adjustment.
2. Adjust New York local time to Berlin local time (11:00+06:00=17:00).

   This adjustment (+6 hours) is, of course, specific to this example.

For further information about time zones, see the documentation for function **tzset()** or **_tzset()** in your compiler run-time library reference.

Convert Store Clock Format to time_t Format

Function Call

```
void tod2time (pmTOD   aTOD,
               time_t* aTime)
```

Header File

pmTOD.h

Description

This function converts a time from Store Clock format to **time_t** format. The **time_t** format contains the seconds since 01-01-1970 00:00:00. Milliseconds from the Store Clock format are not supported and truncated.

Parameters

1. **aTOD** (input)

   The time value to convert, in Store Clock format.
2. **aTime** (output)

   The converted time value, in **time_t** format.

Example

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <time.h>
#include "pmConnect.h"
#include "pmGetInfo.h"
#include "pmTOD.h"

int main(void)
{
  pmHost        myHandle;
  pmReturnCodes error;
  char          *workprofile = "PMUSER  GROUPID PROFILEID       TERMINALID      ";
  pmHashTable   result;
  pmCounter*    aCounter;
  pmCursor      aCursor;
  pmHashTable*  DCInfo;
  pmTOD         timeDifference;
  pmTOD         hostTime;
```

```
time_t        time;
struct tm*    timeStruct;

/* connect to data collector */
error = pmConnect("10.0.0.1", "4711", 850, &myHandle);
if(error.returnCode)
{
  printf("pmConnect - Error [%d/%d]\n", error.returnCode, error.reasonCode);
  exit(-1);
}

/* initialize timezone setting on workstation */
tzset();

/* prepare result area */
clearHashTable(&result);

/* get time difference and current time from DC */
error = pmGetInfo(&myHandle, workprofile, result);
if(error.returnCode)
{
  printf("pmGetInfo - Error [%d/%d]\n", error.returnCode, error.reasonCode);
  exit(-1);
}
else
{
  /* first locate repeating block for DC info */
  aCounter = pmGetCounter(result, "REPDCINF");
  if(aCounter != NULL)
  {
    /* access general info */
    aCursor = initCursor(*aCounter);
    DCInfo = getRepBlockItem(aCursor);

    /* get time difference between local time and GMT on host */
    aCounter = pmGetCounter(*DCInfo, "QR4TID");
    memcpy(timeDifference, aCounter->value, sizeof(pmTOD));

    /* get current time (GMT) on host */
    aCounter = pmGetCounter(*DCInfo, "QR4TIME");
    memcpy(hostTime, aCounter->value, sizeof(pmTOD));

    /* adjust host time to local time by adding time difference */
    pmAddTOD(hostTime, timeDifference, hostTime);

    /* convert Store Clock Format into time_t format */
    tod2time(hostTime, &time);

    /* convert time_t format into tm format (without time zone adjustment) */
    timeStruct = gmtime(&time);

    /* year is relative to 1900 and month starts with 0 -> adjust */
    printf("Local time on host: %2.2d:%2.2d:%2.2d on %4.4d/%2.2d/%2.2d\n",
           timeStruct->tm_hour, timeStruct->tm_min, timeStruct->tm_sec,
           timeStruct->tm_year + 1900, timeStruct->tm_mon + 1,
           timeStruct->tm_mday);
  }

  /* free memory for DC info */
  freeHashTable(result);
}

/* disconnect */
error = pmDisconnect(&myHandle);
if(error.returnCode)
{
  printf("pmDisconnect - Error [%d/%d]\n", error.returnCode, error.reasonCode);
```

```
    exit(-1);
  }

  return(0);
}
```

Convert time_t Format to Store Clock Format

Function Call

```
void time2tod (time_t aTime,
               pmTOD   aTOD)
```

Header File

pmTOD.h

Description

This function converts a time from **time_t** format to Store Clock format. The **time_t** format contains the seconds since 01-01-1970 00:00:00.

Parameters

1. **aTime** (input)

   The time value to convert, in **time_t** format.

2. **aTOD** (output)

   The converted time value, in Store Clock format.

Example

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <time.h>
#include "pmConnect.h"
#include "pmGetInfo.h"
#include "pmTOD.h"

int main(void)
{
  pmHost        myHandle;
  pmReturnCodes error;
  char          *workprofile = "PMUSER  GROUPID PROFILEID       TERMINALID      ";
  pmHashTable   result;
  pmCounter*    aCounter;
  pmCursor      aCursor;
  pmHashTable*  DCInfo;
  pmTOD         timeDifference;
  pmTOD         hostTime;
  time_t        time;
  struct tm*    timeStruct;
  struct tm     newTime;

  /* connect to data collector */
  error = pmConnect("10.0.0.1", "4711", 850, &myHandle);
  if(error.returnCode)
  {
    printf("pmConnect - Error [%d/%d]\n", error.returnCode, error.reasonCode);
    exit(-1);
  }

  /* initialize timezone setting on workstation */
  tzset();

  /* prepare result area */
  clearHashTable(&result);
```

```
  /* get time difference and current time from DC */
  error = pmGetInfo(&myHandle, workprofile, result);
  if(error.returnCode)
  {
    printf("pmGetInfo - Error [%d/%d]\n", error.returnCode, error.reasonCode);
    exit(-1);
  }
  else
  {
    /* first locate repeating block for DC info */
    aCounter = pmGetCounter(result, "REPDCINF");
    if(aCounter != NULL)
    {
      /* access general info */
      aCursor = initCursor(*aCounter);
      DCInfo = getRepBlockItem(aCursor);

      /* get time difference between local time and GMT on host */
      aCounter = pmGetCounter(*DCInfo, "QR4TID");
      memcpy(timeDifference, aCounter->value, sizeof(pmTOD));

      /* convert local time stamp into Store Clock Format             */
      /* preset input data: 15:23:01 2000/05/23 (HH:MM:SS YYYY/MM/DD) */
      newTime.tm_hour = 15;
      newTime.tm_min  = 23;
      newTime.tm_sec  = 01;
      newTime.tm_year = 2000 - 1900;  /* adjust year relative to 1900 */
      newTime.tm_mon  = 5 - 1;        /* adjust month to start at 0   */
      newTime.tm_mday = 23;

      /* convert tm structure into time_t structure using time zone   */
      /* setting on workstation (result will be in GMT depending on   */
      /* implementation of mktime() function)                         */
      time = mktime(&newTime);

      /* convert time_t structure to Store Clock Format */
      time2tod(time, hostTime);

      /* if mktime() did not adjust the time to GMT the Store Clock   */
      /* format needs to be adjusted using pmSubTOD:                  */
      /*   pmSubTOD(hostTime, timeDifference, hostTime);              */

      /* use converted time to request history data, etc. */
      /* ... */
    }

    /* free memory for DC info */
    freeHashTable(result);
  }

  /* disconnect */
  error = pmDisconnect(&myHandle);
  if(error.returnCode)
  {
    printf("pmDisconnect - Error [%d/%d]\n", error.returnCode, error.reasonCode);
    exit(-1);
  }

  return(0);
}
```

Add and Subtract in Store Clock Format

Function Call

```
void pmAddTOD(pmTOD tod1, pmTOD tod2, pmTOD result)
void pmSubTOD(pmTOD tod1, pmTOD tod2, pmTOD result)
```

Header File

pmTOD.h

Description

These functions add and subtract two time values that are given in Store Clock format.

Parameters

1. **tod1** (input)

   tod2 (input)

   Time values to add or subtract, in Store Clock format.

2. **result** (output)

   For **pmAddTOD()**: **tod1** + **tod2**, in Store Clock format.

   For **pmSubTOD()**: **tod1** − **tod2**, in Store Clock format.

Extracting Counters from Hash Tables

Several functions return counters as repeating blocks, which have complex structures and are of varying size. Repeating blocks are described in "Working with Returned Data" on page 36. These counters are stored in hash tables upon return. Hash tables must be initialized before data can be stored in them. The following functions are useful to initialize hash tables, to retrieve individual counter data, and to release the workstation's memory when hash tables are no longer required.

Initialize a Hash Table

Function Call

```
void clearHashTable (pmHashTable *table)
```

Header File

pmHashTable.h

Description

This function initializes a hash table in workstation memory to store counters that contain repeating blocks. Use this function before you call a function that returns data with repeating blocks.

Parameters

1. **table** (input)

   Pointer to the hash table to be initialized (output data area).

Example

```
pmHashTable   result;

clearHashTable(&result);
// call API function that fills result, for example, pmGetInfo()
...
```

Free a Hash Table

Function Call

```
void  freeHashTable (pmHashTable table)
```

Header File

pmHashTable.h

Description

This function frees the memory area that was allocated for a hash table by the **clearHashTable()** function. The counters in the hash table are no longer available.

Parameters

1. **table** (input)

   Hash table to be freed (output data area).

Example

```
pmHashTable   result;

// call API function that fills result, for example, pmGetInfo()
freeHashTable(result);

...
```

Get a Counter from Hash Table

Function Call

```
pmCounter  *pmGetCounter    (pmHashTable   table,
                             char         *name)
```

Header File

pmHashTable.h

Description

This function locates a named counter in a named hash table and returns the counter value. If the counter is not found, the function returns `NULL`.

This function will only search the specified hash table. For hierarchical repeating blocks you need to locate counters in the appropriate hash table determined by the repeating block identifier. See the functions **initCursor()**, **getRepBlockItem()**, **setToNext()**, and **endOfBlock()** for more information about how to navigate through repeating block structures.

Parameters

1. **table** (input)

   Hash table to be searched in.

2. **name** (input)

   Name of counter to be searched for.

Example

This example shows how the hash table named *threadCounters* is searched for a counter named *TXSELECT*. Variable *myCounter* contains a pointer to the result.

```
pmHashTable   threadCounters;
pmCounter    *myCounter;

myCounter = pmGetCounter(threadCounters, "TXSELECT");
```

Initialize a Cursor for a Repeating Block

Function Call

```
pmCursor initCursor (pmCounter aCounter)
```

Header File

pmHashTableList.h

Description

This function returns a cursor which is used to enumerate all items of a repeating block. Use this function after you have located a valid repeating block counter with the **pmGetCounter()** function. You can use the **setToNext()** function to increment the cursor to the next counter in a repeating block.

Parameters

1. **aCounter** (input)

   The counter containing the repeating block counter (for example, REPTHRD) to be enumerated.

Example

In this example the repeating block counter REPTHRD in hash table *result* is located and kept in variable *myCounter*. Next, the cursor for REPTHRD is initialized and maintained in variable *myCursor*. The **getRepBlockItem()** function extracts individual repeating block items from REPTHRD. Function **setToNext()** increments the cursor to the next repeating block item. Function **endOfBlock()** identifies whether REPTHRD contains more repeating block items.

```
pmHashTable   result,
             *threadCounters;
pmCursor      myCursor;
pmCounter    *myCounter;

/* get repeating block counter */
myCounter = pmGetCounter(result, "REPTHRD");
if(myCounter)
{
 myCursor = initCursor(*myCounter);

 /* enumerate all items in repeating block REPTHRD */
 while(!endOfBlock(myCursor))
 {
  threadCounters = getRepBlockItem(myCursor);

  // work with counters of this thread
  ...

  setToNext(&myCursor);
 }
}
```

Get a Repeating Block Item

Function Call

```
pmHashTable   *getRepBlockItem (pmCursor aCursor)
```

Header File

pmHashTableList.h

Description

This function returns a pointer into the hash table where the repeating block item, pointed to by parameter **aCursor**, is located. The cursor **aCursor** was initialized by

the **initCursor()** function, and it needs to be incremented by the **setToNext()** function until the end of a repeating block is encountered by the **endOfBlock()** function.

Parameters

1. **aCursor** (input)

   The enumerative cursor, which points to the current repeating block item.

Example

See the example for "Initialize a Cursor for a Repeating Block" on page 97.

Identify the End of a Repeating Block

Function Call

```
int endOfBlock (pmCursor aCursor)
```

Header File

pmHashTableList.h

Description

This function checks whether the end of the repeating block, pointed to by parameter **aCursor**, is reached.

- If no more items are available, the function returns a nonzero value.
- If one or more items are available, the function returns `0`.

Use this function after you have initialized the cursor for a specified repeating block counter, and before you retrieve individual repeating block items with the **getRepBlockItem()** function.

Parameters

1. **aCursor** (input)

   The enumerative cursor, which points to the current repeating block item, or beyond it, if the end of the block is reached.

Example

See the example for "Initialize a Cursor for a Repeating Block" on page 97.

Increment a Cursor in a Repeating Block

Function Call

```
void setToNext (pmCursor* aCursor)
```

Header File

pmHashTableList.h

Description

This function increments the cursor position for a repeating block counter to the next item in the repeating block. Use this function to step through all items in a repeating block until the **endOfBlock()** function can encounter the end of the repeating block.

Parameters

1. **aCursor** (input)

   The enumerative cursor, which points to the current repeating block item, or beyond it, if the end of the block is reached.

Example
See the example for “Initialize a Cursor for a Repeating Block” on page 97.

Miscellaneous API Functions

This section describes API functions that are used in several examples. They might be helpful to reduce your programming effort.

Free a Memory Area

Function Call
```
void  pmFreeMem (void* ptr)
```

Header File
pmTypes.h

Description
This function frees memory that was allocated by an API function.

Parameters
1. **ptr** (input)

   Pointer to the memory area to be freed.

Example
```
/* do not forget to free output data area       */
/* if the returned data is no longer required.  */
pmFreeMem(data);
```

Format as Hex String

Function Call
```
char*     asHexString     (char            *data,
                           unsigned int     length);
```

Header File
pmTrace.h

Description
This function formats a byte stream as a hexadecimal dump. Use the **pmFreeMem()** function to release the memory area, if the output data is no longer needed.

Parameters
1. **data** (input)

   Pointer to byte stream.
2. **length** (input)

   Number of bytes to include in hexadecimal dump.

Example
```
char *testString = "Hello";
char *asHex;

asHex = asHexString(testString, 5);
printf("Hex dump: %s\n", asHex);  /* will print 'Hex dump: 48656C6C6F' */
                                  /* on ASCII systems.                 */

pmFreeMem(asHex);
```

Format as Hex String (UCS-2)

Function Call

```
unsigned short    *asHexUCS2String   (char           *data,
                                      unsigned int     length);
```

Header File

pmTrace.h

Description

This function formats a byte stream as a hexadecimal dump. The resulting string will be returned in UCS-2 format (two byte Unicode). Use the **pmFreeMem()** function to release the memory area, if the output data is no longer needed.

Parameters

1. **data** (input)

   Pointer to byte stream.

2. **length** (input)

   Number of bytes to include in hexadecimal dump.

Example

```
char *testString = "Hello";
unsigned short *asHex;

asHex = asHexUCS2String(testString, 5);
/* do something with the resulting hex dump */

pmFreeMem(asHex);
```

Initialize a Qualifier List

Function Call

```
void initQualifierList (pmQualifierList* list);
```

Header File

pmGetStatThread.h

Description

This function initializes a qualifier list. Use this function before you add qualifiers to the list with the **addQualifier()** function.

Parameters

1. **list** (input)

   Name of the qualifier list.

Example

See the example in "How Qualifying Works" on page 33.

Add Qualifier List Entries

Function Call

```
pmReturnCodes addQualifier (pmHost*          handle,
                            pmQualifierList* list,
                            char*            counterName,
                            char*            counterValue);
```

Header File

pmGetStatThread.h

Description

This function adds qualifiers to a qualifier list that was created by the **initQualifierList()** function.

Parameters

1. **handle** (input)

   Specify the same handle as the function that uses the qualifier list.

2. **list** (input)

   Specify the name of a qualifier list. The name is identical with the one used to initialize the qualifier list.

3. **counterName** (input)

   Specify a qualification ID (the name of the counter to add to the list).

4. **counterValue** (input)

   Specify a counter value for the qualification ID.

Example

See the example in “How Qualifying Works” on page 33.

Appendix A. Return Codes and Reason Codes

Introduction

Return codes and reason codes indicate the success of an API function call and allow an application program to take appropriate action depending on the result.

- Return codes show the success of a function call, or show the major reason and severity of a failure.
- Reason codes show a more detailed cause of a failure.

The DB2 PM API knows the following return codes:

Table 2. Common DB2 PM API Return Codes

Return Code (Hex)	Description
0x0000	The function completed successfully.
0x0004	The Data Collector returned a warning. Depending on the reason code, output data *might* have been returned.
0x0008	The Data Collector returned an error. The function did not complete successfully. No output data is returned.
PM _ CONNECTION ERROR (0x000C)	The DB2 PM API detected a connection error. The function did not complete successfully. Depending on the reason code, the application *may* still be connected to the Data Collector.
PM _ API ERROR (0x0010)	The DB2 PM API detected an error. The function did not complete successfully.
PM _ API WARNING (0x0014)	The DB2 PM API detected a warning. The function completed.

Return codes and reason codes are represented as hexadecimal values. In addition, several codes are also mapped to more descriptive string constants. Note that these string constants have no intervening space characters. The underscore characters (_) are part of the constants. If the constants appear as broken strings on some output media, this is because of the limited presentation space available.

Network-specific error conditions are indicated as `PM_CONNECTIONERROR` return code. To build appropriate reason codes the API uses platform-specific information. Therefore, several reason codes do not provide a hexadecimal equivalent. You should use the constants instead.[13]

Return Code and Reason Code Descriptions

The following table lists all possible combinations and gives a description of each failure. Where appropriate, function-specific hints are given.

13. If you really need to use the hexadecimal representation of network-specific reason codes, you can identify them (for the operating platform in use) at the beginning of a connection trace (see "Using the DB2 PM API Trace Facility" on page 14 and "Sample Connection Trace" on page 125 for details).

Use this table if you want to understand the reason of a problem.

Table 3. DB2 PM API Return Code and Reason Code Descriptions

Return Code (Hex)	Reason Code (Hex)	Description
0x0004	0x0001	No data returned by DB2.
0x0004	0x0002	Command response contains more than 1 MB of data. Data is truncated.
0x0004	0x0003	Request resulted in a DB2 abend. See the OS/390 system log for more information.
0x0004	0x0004	Logoff already running for the user specified by the work profile.
0x0004	0x0005	Information: Logon is successful. The SAF user is already logged on to the Data Collector through a different connection.
0x0004	0x0006	Authorization exit in the Data Collector is active and returned no data.
0x0004	0x0007	Severe Error in authorization exit. Standard authorization checks are used.
0x0004	0x000D	DB2 command failed. Possible reasons are: • Command authorization failure • Command processor abend • Command syntax error • Command output limit being exceeded Response might be truncated.
0x0004	0x000E	Dynamic statement cache error. You qualified a special statement in the statement cache. The requested statement was not found in the cache.
0x0004	0x000F	No data available in snapshot store. You have requested to view previously saved data, or to build the delta/interval, and at least one buffer is empty. Fill buffer and retry.
0x0004	0x0016	No data returned from data sharing group member. Qualification criteria are not met, or cache is empty for IFCID 316. Correct qualifier.
0x0004	0x0017	No data returned from data sharing group or group member. Verify whether a specified member is a member of the data sharing group.
0x0004	0x0018	No data returned from data sharing group member. Request contains IFCID 254, but no group buffer pools are connected.
0x0004	0x1043	Exception processing not active.
0x0004	0x1045	Event detail no longer available.
0x0008	0x0001	Internal Data Collector error.
0x0008	0x0002	Authorization verification failed. Check the OS/390 system log for messages from the Data Collector.

Table 3. DB2 PM API Return Code and Reason Code Descriptions (continued)

Return Code (Hex)	Reason Code (Hex)	Description
0x0008	0x0007	DB2 BIND outstanding. The Data Collector needs BIND against DB2. See *IBM DB2 Performance Monitor for OS/390 Version 6 Installation and Customization* for how to execute PM BIND.
0x0008	0x0008	DB2 request failed, no data is returned. See the console log for more information.
0x0008	0x0011	Internal Data Collector error.
0x0008	0x0016	Internal Data Collector error.
0x0008	0x0017	Maximum number of 500 users reached.
0x0008	0x001B	Internal Data Collector error.
0x0008	0x001C	Internal Data Collector error.
0x0008	0x001F	The client application has called a DB2 PM API function without being logged on to the Data Collector. User not logged on, or SAF user ID or SAF group ID not found.
0x0008	0x0022	Internal Data Collector error.
0x0008	0x0024	Returned amount of data too large. Request canceled. Use qualification or change some input parameters to reduce the amount of data.
0x0008	0x0025	Internal Data Collector error.
0x0008	0x002B	Internal Data Collector error.
0x0008	0x0032	Internal Data Collector error.
0x0008	0x0035	Internal Data Collector error.
0x0008	0x0036	Request rejected. The history data set is empty. No snapshot is gathered yet.
0x0008	0x0037	Request rejected. You requested a snapshot with parameter **dir** set to `TO` for a time that is older than the oldest snapshot in the history data set. Specify a **requestTime** that is younger than the oldest snapshot in the history data set, or locate the oldest snapshot with **requestTime** set to `TOD_FIRST` and **dir** set to `FORWARD`.
0x0008	0x0038	Request rejected. You requested a snapshot with parameter **dir** set to `BACK` that is already the oldest snapshot in the history data set.

Table 3. DB2 PM API Return Code and Reason Code Descriptions (continued)

Return Code (Hex)	Reason Code (Hex)	Description
0x0008	0x0039	Request rejected. The specified snapshot in the history data set could not be found. • For **pmGetHistory()**: You specified a snapshot with parameter **dir** set to `TO` for a time that is younger than the youngest snapshot in the history data set. Specify a **requestTime** that is older than the youngest snapshot in the history data set, or locate the youngest snapshot with **requestTime** set to `TOD_LAST` and **dir** set to `BACK`. • For **pmGetSnapshotEx()**: You specified `GET_HISTORY` and a time stamp of a snapshot in *historyTime* that could not be found in the history data set. Use the **pmGetHistoryContents()** function to retrieve valid snapshot time stamps and try again.
0x0008	0x003A	Request rejected. You requested a snapshot with parameter **dir** set to `FORWARD` for a snapshot that is currently the youngest in the history data set. Wait until the next snapshot is gathered by the Data Collector and retry the operation.
0x0008	0x003B	Internal Data Collector error.
0x0008	0x003C	Internal Data Collector error.
0x0008	0x003D	Internal Data Collector error.
0x0008	0x003E	Internal Data Collector error.
0x0008	0x0040	Data Collector Authorization exit returned error. Irrecoverable error.
0x0008	0x0042	The function is not available. A DB2 PM license is required for this function.
0x0008	0x0043	The DB2 subsystem is not started, or it cannot be communicated with.
0x0008	0x0044	Logon request failed. CAF Connect error. Verify your DB2 permissions. Check the OS/390 system log for more information.
0x0008	0x0045	Logon request failed. CAF Open error. Verify your DB2 permissions. Check the OS/390 system log for more information.
0x0008	0x0049	Session limit exceeded. The maximum number of TCP/IP sessions, which was specified as Data Collector startup parameter, was exceeded.
0x0008	0x004D	You requested dynamic statement cache counters, but dynamic statement cache is not enabled. Try without these counters.
0x0008	0x0051	Internal Data Collector error.

Table 3. DB2 PM API Return Code and Reason Code Descriptions (continued)

Return Code (Hex)	Reason Code (Hex)	Description
0x0008	0x0052	History request rejected. Snapshot not in history data set or overwritten. Use **pmGetHistoryContents()** to retrieve valid snapshot timestamps and retry.
0x0008	0x0053	You specified `GET_INTERVAL` or `GET_DELTA`, but the time stamp of the snapshot in buffer **Latest** is older or equal to the time stamp in buffer **Stored**. The interval or delta calculation was not possible. Refresh buffer **Latest** with a more recent snapshot.
0x0008	0x0054	Data sharing group support not available. See counter QR4DS (item 3j on page 23) for details.
0x0008	0x0055	No data received from data sharing group member. DB2 might be temporarily short on memory space. Try again.
0x0008	0x0056	A counter is qualified for which no data is returned from a data sharing group member. Correct the qualifier list.
0x0008	0x0057	No data returned from data sharing group member. Monitor Trace Class 1 not active. Start trace for this member.
0x0008	0x0058	No data returned from data sharing group member. The request to the Data Collector contains an incorrect or unsupported IFCID. Verify that the latest DB2 PM PTF is installed. If you cannot solve the problem, call for IBM support.
0x0008	0x0059	Insufficient user authority for the specified fields.
0x0008	0x005A	Invalid combination of input values for members *dsMember* and *dsGlobal*. You cannot specify `PM_GROUP_SCOPE` together with a specific name of a data sharing group member.
0x0008	0x1000	Internal Data Collector error.
0x0008	0x1001	Internal Data Collector error.
0x0008	0x1002	Internal Data Collector error.
0x0008	0x154C	Internal Data Collector error. Request rejected.
0x0008	0x154D	Internal Data Collector error. Request rejected.
0x0008	0x1551	After successful client logon, the Data Collector received a second logon request.

Table 3. DB2 PM API Return Code and Reason Code Descriptions (continued)

Return Code (Hex)	Reason Code (Hex)	Description
0x0008	0x1552	Buffer shortage. The Data Collector has received the maximum number of API function calls it can serve simultaneously. Try again later, or check the Data Collector startup parameters: • Increase value for parameter `TCPIPSESS` (number of parallel TCP/IP sessions). • Check parameters `TCPMULTIUSER` and `TCPMUSB` (multi-user terminal TCP/IP session support).
0x0008	0x1554	Internal Data Collector error.
0x0008	0x155A	Incorrect snapshot store ID specified.
0x0008	0x155B	Internal Data Collector error.
0x0008	0x155C	Internal Data Collector error.
0x0008	0x155D	Incorrect snapshot store ID specified, or snapshot store no longer available.
0x0008	0x155E	Profile ID of user not found or no longer valid. • For **pmLogoff()**: Subuser not logged on.
0x0008	0x1560	Internal Data Collector error.
0x0008	0x1561	Internal Data Collector error. Storage allocation for buffers failed.
0x0008	0x1562	At least one field ID was requested that exceeds the basic scope of collected IFCIDs, as specified by the **pmInitializeStoreEx()** function.
0x0008	0x1563	Internal Data Collector error.
0x0008	0x156C	You specified `GET_DELTA` or `VIEW_DELTA`, but the time stamp of the snapshot in buffer **Latest** is older or equal to the time stamp in buffer **Stored**. The delta calculation was not possible. Refresh buffer **Latest** with a more recent snapshot.
PM _ CONNECTION ERROR (0x000C)	PM _ PORTNUMBER _ TOOHIGH (0x0006)	The specified port number is out of range. It must be between 1 024 and 65 535.
PM _ CONNECTION ERROR (0x000C)	PM _ SERVICE _ UNKNOWN (0x0007)	The specified service name could not be found in the local *services* file.
PM _ CONNECTION ERROR (0x000C)	PM _ HOST _ NOTFOUND (0x0008)	The specified host name could not be resolved. It is not known by your Domain Name System server and your local *hosts* file.
PM _ CONNECTION ERROR (0x000C)	PM _ UNEXPECTED _ EOF (0x0016)	The Data Collector sent incomplete data. Check the OS/390 system log for messages from the Data Collector. The connection may not be usable.
PM _ CONNECTION ERROR (0x000C)	PM _ CONNECTION _ ABORTED	The Data Collector is no longer reachable. It may be stopped, or the connection to it may be terminated.

Table 3. DB2 PM API Return Code and Reason Code Descriptions (continued)

Return Code (Hex)	Reason Code (Hex)	Description
PM _ CONNECTION ERROR (0x000C)	PM _ DCNOT AVAILABLE	The Data Collector is not started or the specified host port is wrong.
PM _ CONNECTION ERROR (0x000C)	PM _ HOSTDOWN	The specified host is down. Try later.
PM _ CONNECTION ERROR (0x000C)	PM _ HOST UNREACHABLE	The specified host is unreachable. Check the specified IP address and the network route.
PM _ CONNECTION ERROR (0x000C)	PM _ NETWORK DOWN	The network is down. Check the network connection.
PM _ CONNECTION ERROR (0x000C)	PM _ SOCKET DESCR _ INVALID	The specified socket descriptor is not valid.
PM _ CONNECTION ERROR (0x000C)	PM _ SOCKET NOT CONNECTED	The specified socket is disconnected. You have to connect first. (Not applicable for Sun Solaris, Linux, and OS/390)
PM _ CONNECTION ERROR (0x000C)	PM _ TIMEOUT	The network operation was canceled because of a timeout. The network traffic is probably slow, or the host IP address is wrong.
PM _ CONNECTION ERROR (0x000C)	PM _ WINSOCK _ NOT READY	Only Windows NT: The network service is not started. Check the network settings.
PM _ CONNECTION ERROR (0x000C)	PM _ WINSOCK _ VERNOTFOUND1	Only Windows NT: The required Winsock version could not be found. Install Winsock V. 1.1 or higher.
PM _ CONNECTION ERROR (0x000C)	PM _ WINSOCK _ VERNOTFOUND2	Only Windows NT: The required Winsock version could not be found. Install Winsock V. 1.1 or higher.
PM _ API ERROR (0x0010)	PM _ DATASTREAM _ INVALID (0x0005)	Received data stream not valid. Counter not found, or end of data reached before expected. • For **pmInitializeStoreEx()**: Verify that the latest DB2 PM PTF is installed. If you cannot solve the problem, call for IBM support.

Table 3. DB2 PM API Return Code and Reason Code Descriptions (continued)

Return Code (Hex)	Reason Code (Hex)	Description
PM _ API ERROR (0x0010)	PM _ INCORRECT _ PARAMETER (0x000B)	One of the specified parameters is incorrect. Check for correct length of the parameters. • For **pmFetchEexceptions()**: Parameter **exceptionNo** is greater than 500. • For **pmInitializeStoreEx()**: Value of **initStoreLevel** is incorrect. • For **pmExecDB2CommandEx()**: Value of **db2CmdParmLevel** is incorrect. • For **pmSaveUserData()**: Value of **length** or **bufferNo** is incorrect. • For **pmGetHistory()**: - **counterNo** must be larger than 0. - **fields** must contain at least one valid counter name. - **dir** must be `BACK`, `FORWARD`, or `TO`. • For **pmGetSnapshotEx()**: Incorrect *mode* in **snapshotParms**, or too many fields are requested (maximum is 2048). • For **pmGetEventDetails()**: Unknown event exception type specified. • For **pmGetEventExceptionLog()**: Start time is equal or greater than end time. • For **pmStopExceptionProc()**: **type** must be `EVENT`.
PM _ API ERROR (0x0010)	PM _ FIELDTABLE _ ERROR (0x0012)	The list of supported counters could not be loaded from the Data Collector. See the trace data and the Data Collector system log for more details. It is not possible to execute DB2 PM functions before the **pmConnect()** function succeeds. Retry.
PM _ API ERROR (0x0010)	PM _ MEMORY _ ERROR (0x0017)	Insufficient workstation memory available. An API function fails because it cannot allocate sufficient memory. Try to stop other workstation applications, or increase the size of the workstation's virtual memory paging file.
PM _ API ERROR (0x0010)	PM _ UNSUPPORTED _ CHARACTER (0x0018)	An input parameter contains a character that cannot be converted to an equivalent EBCDIC character. Check and correct any string parameters for the function that causes this problem.
PM _ API ERROR (0x0010)	PM _ CODEPAGE _ ERROR	The specified code page could not be found or is not valid.

Table 3. DB2 PM API Return Code and Reason Code Descriptions (continued)

Return Code (Hex)	Reason Code (Hex)	Description
PM _ API ERROR (0x0010)	PM _ INPUTAREA _ OVERFLOW	Request input area overflow. • For **pmInitializeStoreEx()**: You requested too many counters, or too many counters as qualifiers. Divide the snapshot store into two or more stores. • For **pmGetHistoryContents()**: Too many IFCIDs requested. Request input area overflow. Reduce the number of requested IFCIDs.
PM _ API WARNING (0x0014)	PM _ USING _ DEFAULT _ CODEPAGE (0x0013)	The specified code page could not be loaded from the Data Collector. Using default code page ASCII 850.
PM _ API WARNING (0x0014)	PM _ OLD _ DC (0x0014)	The Data Collector is not up to date. The Data Collector interface changed meanwhile. Therefore some API functions might fail. Install the latest Data Collector PTF.
PM _ API WARNING (0x0014)	PM _ FIELDTABLE _ UNEXPECTEDEOF (0x0015)	The Data Collector returned an incorrect or incomplete list of supported DB2 PM counters. Some DB2 PM counters may be missing and cause some functions to fail. Check the trace for more details.

Return Codes and Reason Codes by Function

The following table lists all functions and all return codes and reason codes issued by it.

Use this table during your programming work to identify possible combinations returned by a function call.

Table 4. DB2 PM API Return Codes and Reason Codes by Function

Function	Return Code (Hex)	Reason Code (Hex)
All functions	0x0008	0x001F
	0x0008	0x0043
	0x0008	0x1552
	PM _ CONNECTION ERROR (0x000C)	PM _ UNEXPECTED _ EOF (0x0016)
	PM _ CONNECTION ERROR (0x000C)	PM _ CONNECTION _ ABORTED
	PM _ CONNECTION ERROR (0x000C)	PM _ HOSTDOWN
	PM _ CONNECTION ERROR (0x000C)	PM _ HOST UNREACHABLE
	PM _ CONNECTION ERROR (0x000C)	PM _ NETWORK DOWN
	PM _ CONNECTION ERROR (0x000C)	PM _ SOCKET DESCR _ INVALID
	PM _ CONNECTION ERROR (0x000C)	PM _ SOCKET NOT CONNECTED
	PM _ CONNECTION ERROR (0x000C)	PM _ TIMEOUT
	PM _ CONNECTION ERROR (0x000C)	PM _ WINSOCK _ NOT READY
	0x0010	0x0017
	0x0010	0x0018
pmConnect()	0x008	0x007
	PM _ CONNECTION ERROR (0x000C)	PM _ PORTNUMBER _ TOOHIGH (0x0006)
	PM _ CONNECTION ERROR (0x000C)	PM _ SERVICE _ UNKNOWN (0x0007)
	PM _ CONNECTION ERROR (0x000C)	PM _ HOST _ NOTFOUND (0x0008)
	PM _ CONNECTION ERROR (0x000C)	PM _ DCNOT AVAILABLE
	PM _ CONNECTION ERROR (0x000C)	PM _ WINSOCK _ VERNOTFOUND1
	PM _ CONNECTION ERROR (0x000C)	PM _ WINSOCK _ VERNOTFOUND2
	PM _ API ERROR (0x0010)	PM _ FIELDTABLE _ ERROR (0x0012)
	PM _ API ERROR (0x0010)	PM _ CODEPAGE _ ERROR
	PM _ API WARNING (0x0014)	PM _ USING _ DEFAULT _ CODEPAGE (0x0013)
	PM _ API WARNING (0x0014)	PM _ FIELDTABLE _ UNEXPECTEDEOF (0x0015)

Table 4. DB2 PM API Return Codes and Reason Codes by Function (continued)

Function	Return Code (Hex)	Reason Code (Hex)
pmExecDB2CommandEx()	0x0004	0x0001
	0x0004	0x0002
	0x0004	0x0003
	0x0004	0x000D
	0x0004	0x0017
	0x0008	0x0001
	0x0008	0x0008
	0x0008	0x0011
	0x0008	0x0022
	0x0008	0x0054
	0x0008	0x0055
	0x0008	0x005A
	PM _ API ERROR (0x0010)	PM _ INCORRECT _ PARAMETER (0x000B)
pmFetchExceptions()	0x0004	0x0006
	0x0004	0x0007
	0x0004	0x1043
	0x0008	0x0001
	0x0008	0x0008
	0x0008	0x0042
	0x0008	0x1000
	PM _ API ERROR (0x0010)	PM _ INCORRECT _ PARAMETER (0x000B)
pmGenPassticket()	0x0008	0x0001
	0x0008	0x0011
	0x0008	0x001B
	0x0008	0x0022
	PM _ API ERROR (0x0010)	PM _ DATASTREAM _ INVALID (0x0005)
	PM _ API ERROR (0x0010)	PM _ INCORRECT _ PARAMETER (0x000B)
	PM _ API WARNING (0x0014)	PM _ OLD _ DC (0x0014)
pmGetEventDetails()	0x0004	0x1045
	0x0008	0x0001
	0x0008	0x1000
	0x0008	0x1001
	PM _ API ERROR (0x0010)	PM _ INCORRECT _ PARAMETER (0x000B)

Table 4. DB2 PM API Return Codes and Reason Codes by Function (continued)

Function	Return Code (Hex)	Reason Code (Hex)
pmGetEventExceptionLog()	0x0008	0x0001
	0x0008	0x0042
	0x0008	0x1000
	0x0008	0x1001
	PM _ API ERROR (0x0010)	PM _ INCORRECT _ PARAMETER (0x000B)
pmGetExceptionStatus()	0x0008	0x0001
	0x0008	0x0042
	0x0008	0x1000
	PM _ API ERROR (0x0010)	PM _ DATASTREAM _ INVALID (0x0005)
pmGetHistory()	0x0008	0x0001
	0x0008	0x0022
	0x0008	0x0032
	0x0008	0x0035
	0x0008	0x0036
	0x0008	0x0037
	0x0008	0x0038
	0x0008	0x0039
	0x0008	0x003A
	0x0008	0x003B
	0x0008	0x003C
	0x0008	0x003D
	0x0008	0x003E
	0x0008	0x0040
	0x0008	0x0042
	PM _ API ERROR (0x0010)	PM _ INCORRECT _ PARAMETER (0x000B)
	PM _ API ERROR (0x0010)	PM _ INPUTAREA _ OVERFLOW
pmGetHistoryContents()	0x0008	0x0024
	0x0008	0x0022
	0x0008	0x0051
	PM _ API ERROR (0x0010)	PM _ INPUTAREA _ OVERFLOW

Table 4. DB2 PM API Return Codes and Reason Codes by Function (continued)

Function	Return Code (Hex)	Reason Code (Hex)
pmGetInfo()	0x0008	0x0001
	0x0008	0x0011
	0x0008	0x001B
	0x0008	0x0022
	PM _ API ERROR (0x0010)	PM _ DATASTREAM _ INVALID (0x0005)
	PM _ API WARNING (0x0014)	PM _ OLD _ DC (0x0014)

Table 4. DB2 PM API Return Codes and Reason Codes by Function (continued)

Function	Return Code (Hex)	Reason Code (Hex)
pmGetSnapshotEx()	0x0004	0x0001
	0x0004	0x0002
	0x0004	0x0003
	0x0004	0x0006
	0x0004	0x0007
	0x0004	0x000E
	0x0004	0x000F
	0x0008	0x0001
	0x0008	0x0008
	0x0008	0x0022
	0x0008	0x0024
	0x0008	0x0025
	0x0008	0x002B
	0x0008	0x0035
	0x0008	0x0036
	0x0008	0x0039
	0x0008	0x003B
	0x0008	0x003C
	0x0008	0x003D
	0x0008	0x003E
	0x0008	0x0040
	0x0008	0x0042
	0x0008	0x004D
	0x0008	0x0053
	0x0008	0x1554
	0x0008	0x155A
	0x0008	0x155D
	0x0008	0x155E
	0x0008	0x1560
	0x0008	0x1561
	0x0008	0x1562
	0x0008	0x1563
	0x0008	0x156C
	PM _ API ERROR (0x0010)	PM _ INCORRECT _ PARAMETER (0x000B)

Table 4. DB2 PM API Return Codes and Reason Codes by Function (continued)

Function	Return Code (Hex)	Reason Code (Hex)
pmGetUserData()	0x0008	0x0001
	0x0008	0x0022
	0x0008	0x0042
	0x0008	0x154D
	PM _ API ERROR (0x0010)	PM _ INCORRECT _ PARAMETER (0x000B)
pmInitializeStoreEx()	0x0004	0x0016
	0x0004	0x0017
	0x0004	0x0018
	0x0008	0x0001
	0x0008	0x0022
	0x0008	0x0024
	0x0008	0x0025
	0x0008	0x0042
	0x0008	0x0054
	0x0008	0x0055
	0x0008	0x0056
	0x0008	0x0057
	0x0008	0x0058
	0x0008	0x0059
	0x0008	0x005A
	0x0008	0x1554
	0x0008	0x155B
	0x0008	0x155C
	PM _ API ERROR (0x0010)	PM _ DATASTREAM _ INVALID (0x0005)
	PM _ API ERROR (0x0010)	PM _ INCORRECT _ PARAMETER (0x000B)
	PM _ API ERROR (0x0010)	PM _ INPUTAREA _ OVERFLOW
pmLogoff()	0x0004	0x0004
	0x0008	0x0001
	0x0008	0x0011
	0x0008	0x001F
	0x0008	0x0022
	0x0008	0x155E

Table 4. DB2 PM API Return Codes and Reason Codes by Function (continued)

Function	Return Code (Hex)	Reason Code (Hex)
pmLogon()	0x0004	0x0005
	0x0008	0x0001
	0x0008	0x0002
	0x0008	0x0016
	0x0008	0x0017
	0x0008	0x001C
	0x0008	0x0022
	0x0008	0x0044
	0x0008	0x0045
	0x0008	0x0049
	0x0008	0x1551
pmQueryStores()	0x0008	0x0001
	0x0008	0x0022
	0x0008	0x0025
	0x0008	0x0042
	0x0008	0x1554
pmReleaseStore()	0x0008	0x0001
	0x0008	0x0022
	0x0008	0x0025
	0x0008	0x0042
	0x0008	0x1554
	0x0008	0x155A
	0x0008	0x155D
	0x0008	0x155E

Table 4. DB2 PM API Return Codes and Reason Codes by Function (continued)

Function	Return Code (Hex)	Reason Code (Hex)
pmResetEx()	0x0004	0x0001
	0x0004	0x0002
	0x0004	0x0003
	0x0004	0x0006
	0x0004	0x0007
	0x0004	0x000E
	0x0004	0x000F
	0x0008	0x0001
	0x0008	0x0008
	0x0008	0x0022
	0x0008	0x0024
	0x0008	0x0025
	0x0008	0x002B
	0x0008	0x0035
	0x0008	0x0036
	0x0008	0x0039
	0x0008	0x003B
	0x0008	0x003C
	0x0008	0x003D
	0x0008	0x003E
	0x0008	0x0040
	0x0008	0x0042
	0x0008	0x004D
	0x0008	0x1554
	0x0008	0x155A
	0x0008	0x155D
	0x0008	0x155E
	0x0008	0x1560
	0x0008	0x1561
	0x0008	0x1562
	0x0008	0x1563
pmSaveUserData()	0x0008	0x0001
	0x0008	0x0022
	0x0008	0x0042
	0x0008	0x154C
	PM _ API ERROR (0x0010)	PM _ INCORRECT _ PARAMETER (0x000B)

Table 4. DB2 PM API Return Codes and Reason Codes by Function (continued)

Function	Return Code (Hex)	Reason Code (Hex)
pmStartExceptionProc()	0x0008	0x0001
	0x0008	0x0042
	0x0008	0x1000
	0x0008	0x1001
	0x0008	0x1002
pmStopExceptionProc()	0x0004	0x1043
	0x0008	0x0001
	0x0008	0x1000
	PM _ API ERROR (0x0010)	PM _ INCORRECT _ PARAMETER (0x000B)

Appendix B. Field Table Summary

The field table contains a list of all data fields that are accessible through DB2 PM API functions. Because of the number of fields the information about it is not shown in this guide, but is provided as a text file that accompanies the DB2 PM API. The file resides in data set SDGOWS01 as member DGOKFLDS.

A short abstract of the field table (section General IDs) is shown next:

Field ID	Field Name	Field Type	Field Length	Description
0001	LNGTHID	Char	0002	Length of current block. In contrast to LNGTHID4, this is a 2-byte field. The value includes the length of LNGTHID itself.
0002	EYECAT	Char	0004	Eye catcher of current block.
0003	LNGTHID4	Int	0004	Length of current block. In contrast to LNGTHID, this is a 4-byte field. The value includes the length of LNGTHID4 itself.
0010	REPSTAT	Rep	0002	Start of statistics record. Contains the number of repetitions as value.
0011	REPSTDDF	Rep	0002	Start of DDF repeating block in statistics record. Contains the number of repetitions as value.
0012	REPSTBUF	Rep	0002	Start of buffer pool repeating block in statistics record. Contains the number of repetitions as value.
0013	REPSTGBF	Rep	0002	Start of group buffer pool repeating block in statistics record. Contains the number of repetitions as value.
0014	REPSTGLB	Rep	0002	Start of CF cache data repeating block in statistics record. Contains the number of repetitions as value.
⋮	⋮	⋮	⋮	⋮
0050	SNAPTIME	Date	0008	Time stamp of current snapshot.
⋮	⋮	⋮	⋮	⋮
0103	HISTINTV	Smint	0002	Number of history snapshots.
0104	HISTOFF	Smint	0002	Offset for history interval.

Column
Meaning

Field ID
Unique identifier number of a counter. The field table is sorted by identifier number. Ranges of numbers might be reserved for future use.

Field Name
Symbolic name of a counter. These counter names are used in an application program to specify counters.

Field Type
Data type of the named counter.

Field Length
Length in bytes of the named counter.

Description
Short description of the purpose of the counter.

The entire field table is sorted by Field ID (column 1). Fields that describe a common topic are grouped together. You will find the following groups in the field table. Notice that fields or groups marked "for internal purposes only" should not be used.

- General IDs

  General field IDs that describe the structure of the provided or returned data.
- Qualification IDs

  Field IDs that qualify data. See *DB2 for OS/390 Administration Guide*, the appendix about IFI programming about qualifying.
- TOP/Sort IDs
- Statistics data:
  - Address space data
  - Instrumentation destination data
  - Instrumentation data
  - Subsystem services data
  - Command data
  - IFC checkpoint data
  - Log manager data
  - Distributed Data Facility (DDF) data
  - Distributed Data Facility (DDF) system data
  - SQL statement data
  - Bind data
  - Buffer manager data
  - Data manager control data
  - Lock usage data
  - EDM pool usage data
  - Group buffer pool usage data
  - Global locking data
  - CF cache structure data
- Collect report data IDs

  Fields concerning Collect Report Data.
- Thread data:
  - Exception IDs
  - Thread Distributed Detail Data
- Exception IDs
- Statistics data:
  - Statement cache
  - Statement text
  - System parameters block - SYSP

  - Log initialization parameters block
  - Archive initialization parameters block
  - System parameters - SPRM
  - List of VSAM catalog qualifiers
  - Database start flags
  - List of all databases
  - Distributed Data Facility (DDF) start control information
  - Group initialization parameters block
  - DSNHDECP CSECT
  - Buffer Manager group buffer pool attributes
  - Buffer Manager dynamic pool attributes
- Thread data:
  - Current statement data
  - Current statement text
  - Current statement data
  - Current statement data
  - Current statement data
  - Instrumentation data
  - Buffer pool usage data
  - SQL statement data
  - Buffer pool usage data
  - Lock usage data
  - Correlation header data
  - Agent status data
  - Distributed header data
  - Distributed data
  - Distributed accounting data
  - Account code and Distributed Data Facility (DDF) data
  - IFI accounting data
  - Package data
  - Global locking data
  - Group buffer pool data
  - Locked resources
  - Locked resources
  - Locked resources
  - Locked resources
  - Locked resources - details

A cross-reference list follows the field table that maps field names to field IDs.

Appendix C. Sample Traces

This section shows three samples of trace data. "Using the DB2 PM API Trace Facility" on page 14 describes how to produce traces.

Sample Connection Trace

```
(CONNECTION) 10:30:13, PID 92 : +-----------------------------------------+
(CONNECTION) 10:30:13, PID 92 : | PM_NETWORKDOWN          | 10050 | 0x2742 |
(CONNECTION) 10:30:13, PID 92 : +-----------------------------------------+
(CONNECTION) 10:30:13, PID 92 : | PM_TIMEOUT              | 10053 | 0x2745 |
(CONNECTION) 10:30:13, PID 92 : +-----------------------------------------+
(CONNECTION) 10:30:13, PID 92 : | PM_HOSTUNREACHABLE      | 10060 | 0x274c |
(CONNECTION) 10:30:13, PID 92 : +-----------------------------------------+
(CONNECTION) 10:30:13, PID 92 : | PM_HOSTDOWN             | 10064 | 0x2750 |
(CONNECTION) 10:30:13, PID 92 : +-----------------------------------------+
(CONNECTION) 10:30:13, PID 92 : | PM_DCNOTAVAILABLE       | 10061 | 0x274d |
(CONNECTION) 10:30:13, PID 92 : +-----------------------------------------+
(CONNECTION) 10:30:13, PID 92 : | PM_SOCKETDESCR_INVALID  | 10038 | 0x2736 |
(CONNECTION) 10:30:13, PID 92 : +-----------------------------------------+
(CONNECTION) 10:30:13, PID 92 : | PM_SOCKET_NOTCONNECTED  | 10038 | 0x2736 |
(CONNECTION) 10:30:13, PID 92 : +-----------------------------------------+
(CONNECTION) 10:30:13, PID 92 : | PM_CONNECTION_ABORTED   | 10058 | 0x274a |
(CONNECTION) 10:30:13, PID 92 : +-----------------------------------------+
(CONNECTION) 10:30:13, PID 92 : sendRequest: Request input area data for request on socket 0x00C4 :
(CONNECTION) 10:30:13, PID 92 : receiveRequest :
(CONNECTION) 10:30:13, PID 92 : receiveRequest : Received request header
(CONNECTION) 10:30:13, PID 92 : receiveRequest : Receiving data for request...
(CONNECTION) 10:30:13, PID 92 : receiveRequest : Request on socket 0x00C4 received!
(CONNECTION) 10:30:13, PID 92 : sendRequest: Request input area data for request on socket 0x00C4 :
(CONNECTION) 10:30:13, PID 92 : receiveRequest :
(CONNECTION) 10:30:13, PID 92 : receiveRequest : Received request header
(CONNECTION) 10:30:13, PID 92 : receiveRequest : Receiving data for request...
(CONNECTION) 10:30:13, PID 92 : receiveRequest : Request on socket 0x00C4 received!
(CONNECTION) 10:30:13, PID 92 : sendRequest: Request input area data for request on socket 0x00C4 :
(CONNECTION) 10:30:13, PID 92 : receiveRequest :
(CONNECTION) 10:30:13, PID 92 : receiveRequest : Received request header
(CONNECTION) 10:30:13, PID 92 : receiveRequest : Request on socket 0x00C4 received!
(CONNECTION) 10:30:13, PID 92 : sendRequest: Request input area data for request on socket 0x00C4 :
(CONNECTION) 10:30:13, PID 92 : receiveRequest :
(CONNECTION) 10:30:13, PID 92 : receiveRequest : Received request header
(CONNECTION) 10:30:13, PID 92 : receiveRequest : Receiving data for request...
(CONNECTION) 10:30:13, PID 92 : receiveRequest : Request on socket 0x00C4 received!
(CONNECTION) 10:30:14, PID 92 : sendRequest: Request input area data for request on socket 0x00C4 :
(CONNECTION) 10:30:14, PID 92 : receiveRequest :
(CONNECTION) 10:30:14, PID 92 : receiveRequest : Received request header
(CONNECTION) 10:30:14, PID 92 : receiveRequest : Receiving data for request...
(CONNECTION) 10:30:14, PID 92 : receiveRequest : Request on socket 0x00C4 received!
⋮
```

Sample Command Trace

```
(COMMAND) 10:30:13, PID 92 :
(COMMAND) 10:30:13, PID 92 : ********************************************************************************
(COMMAND) 10:30:13, PID 92 : *                                                                              *
(COMMAND) 10:30:13, PID 92 : *  Application Programming Interface                                           *
(COMMAND) 10:30:13, PID 92 : *  ---------------------------------                                           *
(COMMAND) 10:30:13, PID 92 : *                                                                              *
(COMMAND) 10:30:13, PID 92 : *  IBM  DB2  UDB  Performance  Monitor  for OS/390  V6                         *
(COMMAND) 10:30:13, PID 92 : *                                                                              *
(COMMAND) 10:30:13, PID 92 : *  COPYRIGHT :    5645-DB2  (C) Copyright IBM Corp. 1999, 2000                 *
(COMMAND) 10:30:13, PID 92 : *                 LICENSED MATERIALS - PROPERTY OF IBM                         *
(COMMAND) 10:30:13, PID 92 : *                 SEE COPYRIGHT INSTRUCTIONS, G120 - 2083                      *
(COMMAND) 10:30:13, PID 92 : *                                                                              *
(COMMAND) 10:30:13, PID 92 : ********************************************************************************
(COMMAND) 10:30:13, PID 92 : API Version:        : V6
(COMMAND) 10:30:13, PID 92 : API Release:        : R6
(COMMAND) 10:30:13, PID 92 : API Internal Version: 6
(COMMAND) 10:30:13, PID 92 :
(COMMAND) 10:30:13, PID 92 : pmConnect() : Connecting to 9.164.172.191:6653 ...
(COMMAND) 10:30:13, PID 92 : pmConnect() : initializing network ...
(COMMAND) 10:30:13, PID 92 : pmConnect() : Network initialized.
(COMMAND) 10:30:13, PID 92 : pmConnect() : port specified as integer ...
(COMMAND) 10:30:13, PID 92 : pmConnect() : hostname specified as dotted decimal integer (ip addr) ...
(COMMAND) 10:30:13, PID 92 : pmConnect() : Creating socket ...
(COMMAND) 10:30:13, PID 92 : pmConnect() : Socket created.
(COMMAND) 10:30:13, PID 92 : pmConnect() : Connecting ...
(COMMAND) 10:30:13, PID 92 : pmConnect() : Connect successfull.
(COMMAND) 10:30:13, PID 92 : pmConnect() : Loading code page from host ...
(COMMAND) 10:30:13, PID 92 : createRequestheader : workProfile - NULL
(COMMAND) 10:30:13, PID 92 : loadCodePage() : Receiving return area successfull !
(COMMAND) 10:30:13, PID 92 : loadCodePage() : Now parsing response ...
(COMMAND) 10:30:13, PID 92 : loadCodePage() : Host uses EBCDIC code page 500.
(COMMAND) 10:30:13, PID 92 : 00010203372D2E2F1605250B0C0D0E0F101112133C3D322618191C27071D1E1F404F7...
⋮
(COMMAND) 10:30:13, PID 92 : pmConnect() : Loading field table from dc ...
(COMMAND) 10:30:13, PID 92 : createRequestheader : workProfile - NULL
(COMMAND) 10:30:13, PID 92 : loadFieldTableFromDC() : Receiving return area successfull !
(COMMAND) 10:30:13, PID 92 : loadFieldTableFromDC() : Now parsing response ...
(COMMAND) 10:30:13, PID 92 : loadFieldTableFromDC() : 1921 counters loaded.
(COMMAND) 10:30:13, PID 92 : loadFieldTableFromDC() : field table load complete.
(COMMAND) 10:30:13, PID 92 : pmLogon() : Starting logon ...
```

```
(COMMAND) 10:30:13, PID 92 : createRequestheader : workProfile - chm            DB2PMTEST1      AAA001
(COMMAND) 10:30:13, PID 92 : pmLogon() : Sending request ...
(COMMAND) 10:30:13, PID 92 : pmLogon() : Logon failed! RC/RS [0x0004/0x0005]
```

Sample Data Trace

```
(DATA) 10:30:13, PID 92 :
(DATA) 10:30:13, PID 92 : sendRequest: Request input area data for request on socket 0x00C4 :
(DATA) 10:30:13, PID 92 : C4C7D6C9000000A4F0F2004B0000000000000000000000000000000000000000000000010000000000000...
(DATA) 10:30:13, PID 92 :
(DATA) 10:30:13, PID 92 :
(DATA) 10:30:13, PID 92 : receiveRequest: Result header data for request on socket 0x00C4 :
(DATA) 10:30:13, PID 92 : C4C7D6C9000000A4F0F2004B0000000000000000000000000000000000000000000000010000000010000...
(DATA) 10:30:13, PID 92 :
(DATA) 10:30:13, PID 92 :
(DATA) 10:30:13, PID 92 : receiveRequest: Result data for request on socket 0x00C4 :
(DATA) 10:30:13, PID 92 : 3051000003523052000001F43053000003330001013530540000000030550000035230560000...
(DATA) 10:30:13, PID 92 :
(DATA) 10:30:13, PID 92 :
(DATA) 10:30:13, PID 92 : sendRequest: Request input area data for request on socket 0x00C4 :
(DATA) 10:30:13, PID 92 : C4C7D6C9000000A4F0F2004C0000000000000000000000000000000000000000000000010000000000000...
(DATA) 10:30:13, PID 92 :
(DATA) 10:30:13, PID 92 :
(DATA) 10:30:13, PID 92 : receiveRequest: Result header data for request on socket 0x00C4 :
(DATA) 10:30:13, PID 92 : C4C7D6C9000000A4F0F2004C0000000000000000000000000000000000000000000000010000000010000...
(DATA) 10:30:13, PID 92 :
(DATA) 10:30:13, PID 92 :
(DATA) 10:30:13, PID 92 : receiveRequest: Result data for request on socket 0x00C4 :
(DATA) 10:30:13, PID 92 : D3D5C7E3C8C9C44000010002E240C5E8C5C3C1E3404000020004C340D9C5D7E2E3C1E3400010...
(DATA) 10:30:13, PID 92 :
(DATA) 10:30:13, PID 92 :
:
```

Appendix D. Outdated Functions

The following functions were replaced over time, but are further supported by the API. Their descriptions are kept here for reference to ensure maintainability of application programs that used them in the past. For return and reason codes refer to the replacement functions in "Appendix A. Return Codes and Reason Codes" on page 103.

Alphabetical List

pmExecDB2Command()

This function is replaced by the **pmExecDB2CommandEx()** function.

Function Call

```
pmReturnCodes pmExecDB2Command  (pmHost*      handle,
                                 char*        workProfile,
                                 char*        command,
                                 char**       response)
```

Header File

pmExecDB2Command.h

Description

This function requests the Data Collector to execute a DB2 command and to return the command response for a user specified by parameter **workProfile**.

All DB2 commands are allowed, except you cannot request the Data Collector to start or to stop DB2.

The API allocates an output data area of up to 1 MB to store the command response. If the amount of data returned exceeds 1 MB, data is truncated, and the function returns a warning. The output data area holding the command response remains allocated until it is freed by the application.

Parameters

1. **handle** (input)

   The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.

2. **workProfile** (input)

   A user's work profile for which this function is to be executed. The user was specified by the **pmLogon()** function.

3. **command** (input)

   The DB2 command to execute. Precede the command by a hyphen "-". The command can be written in upper- or lowercase, or in mixed case.

4. **response** (output)

   Pointer to the output data area in memory where the result of the DB2 command is stored. Use the **pmFreeMem()** function to release the memory area, if the output data is no longer needed.

Example

```
#include "pmExecDB2Command.h"
#include "pmConnect.h"
#include "pmLogOnOff.h"
#include <stdlib.h>
...

pmHost         myHandle;
char           workProfile[] = "PMUSER          DB2PM           10.0.0.1:0001   ";
char*          response = NULL;
pmReturnCodes  error;

// connect to data collector and log on
...

// execute command to display bufferpools
error = pmExecDB2Command(&myHandle, workProfile,
                         "-DISPLAY BUFFERPOOL(*)", &response);

// work with output
...

// don't forget to free memory for DB2 response
if(response) pmFreeMem(response);

// log off and disconnect
...
```

pmGetSnapshot()

This function is replaced by the **pmGetSnapshotEx()** function.

Function Call

```
pmReturnCodes pmGetSnapshot  (pmHost*           handle,
                              char*             workProfile,
                              pmSnapshotMode    mode,
                              unsigned int      id,
                              char**            fields,
                              unsigned int      counterNo,
                              pmTOD             timestampLatest,
                              pmTOD             timestampStored,
                              pmHashTable       result)
```

Header File

pmGetStatThread.h

Description

This function requests the Data Collector to return counter data from a specified snapshot store for a user identified by parameter **workProfile**.

Parameter **mode** controls the buffer calculations and the results to be returned. It also controls whether actual DB2 performance data or data from the history data set is to be used. If the latter is to be used, the Data Collector must have been started with history recording active.

Parameter **fields** controls which counters from the specified counter store are to be returned.

This function requires that the output data area is initialized with the **clearHashTable()** function.

Parameters

1. **handle** (input)

   The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.

2. **workProfile** (input)

   A user's work profile for which this function is to be executed. The user was specified by the **pmLogon()** function.

3. **mode** (input)

   One or more keywords that describes how the counters in the identified snapshot store are treated before being returned. The following table summarizes the possible modes and its major characteristics. For a detailed description see "Possible Modes to Control the Snapshot Store Operation" on page 30.

   You need to specify one of the keywords shown in the left column of the table. If you want to specify additional keywords, provide them as shown in the following example. Internally, the keywords are declared as numeric constants, which need to be bitwise ORed to control the program flow.

   ```
   GET_LATEST | GET_HISTORY | GET_LOCKEDRESOURCES
   ```

Mode	Copies "Latest" to "Stored"	Refreshes "Latest"	Returns ...	Remark
GET_LATEST	No	Yes	"Latest"	Optionally, together with GET_HISTORY, or GET_LOCKEDRESOURCES, or both. Alternatively, together with GET_HISTORY, or GET_SUMMARY, or both.
VIEW_LATEST	No	No	"Latest"	Optionally, together with GET_LOCKEDRESOURCES. Alternatively, together with GET_SUMMARY.
GET_INTERVAL	No	Yes	"Latest"–"Stored"	Preset by **pmReset()**. Optionally, together with GET_HISTORY, or GET_SUMMARY, or both.
VIEW_INTERVAL	No	No	"Latest"–"Stored"	Optionally, together with GET_SUMMARY.
GET_DELTA	Yes	Yes	"Latest"–"Stored"	Optionally, together with GET_HISTORY, or GET_SUMMARY, or both.
VIEW_DELTA	No	No	"Latest"–"Stored"	Optionally, together with GET_SUMMARY.

   - The GET_HISTORY keyword selects the history data set (instead of actual DB2 performance data) as the data source to refresh the counters in a snapshot store before their values are returned. It can be used together with one of the GET modes, which refreshes buffer **Latest**.

The snapshot in the history data set is identified by parameter **timestampLatest**. See "History Processing - Get History Contents" on page 52 about how to retrieve a list of all snapshots in the history data set and how to identify a snapshot by its time stamp.

- If you specify GET_INTERVAL, you must first use the **pmReset()** function to ensure that the reference values for interval processing are set. See "Snapshot Processing - Reset Interval Data" on page 49 for more details.

  If you specify GET_INTERVAL together with GET_HISTORY, ensure that you have selected an appropriate data source also for the **pmReset()** function.
- The GET_SUMMARY keyword causes the Data Collector to summarize all thread-related counters. You can combine this keyword with other mode keywords, except GET_LOCKEDRESOURCES.

4. **id** (input)

   The identification of the snapshot store for which this function is to be executed. The identification was set by the **pmInitializeStore()** function.

5. **fields** (input)

   An array of counter names for which counter data is requested.

   If you set **fields** to NULL, counter data is requested for all counters in this snapshot store.

   If you specify a subset of counter names, counter data is requested only for the specified counters. A subset is any number of counters that was specified by the **pmInitializeStore()** function for this snapshot store **id**.

   To specify a subset of counters insert the counter names as follows. Counter names are not case-sensitive. Invalid counters are ignored. Counters not available in the stored snapshot record in a history data set are ignored. No warning is returned.

   ```
   char *fields[] = {
                       "QISEDBD",  "QTMAXDS", "QTAUCHK",  "QTAUSUC",  "QBSTMAX",
                       "QBSTPID",  "QBSTGET", "QBSTWFT",  "QBSTWFD",  "QBSTWKPD",
                       "QBSTWKPD", "QBSTMAX", "QBSTWKPD", "QBSTWDRP", "QBSTWBVQ",
                       "QBSTWFR",  "QBSTWFF", "QBSTWFD",  "QISEDSI",  "QISEDSG",
                       "QISEDSC",  "QXSTFND", "QXSTNFND", "QBSTWDRP", "QBSTWBVQ"
                     };
   ```

   If you specify also thread counters in the array, and GET_SUMMARY as **mode** parameter, all thread counters in the array are summarized before being returned. See the field table, section "Thread Data", for valid thread counter names.

6. **counterNo** (input)

   The number of counters specified with parameter **fields**. If parameter **fields** is NULL, set **counterNo** to 0.

7. **timestampLatest** (output/input)

   This parameter returns a time stamp, which shows when the buffer **Latest** of all requested counters in this store was changed.

   If the **pmGetSnapshot()** function is used with **mode** parameter GET_HISTORY, specify a valid time stamp of a snapshot in the history data set. See parameter **mode** for more details. Input data is overwritten upon return.

8. **timestampStored** (output)

   This parameter returns a time stamp, which shows when the buffer **Stored** of all requested counters in this store was changed.

   The time stamp holds a valid time only if parameter **mode** is one of the following:

- GET_INTERVAL
- VIEW_INTERVAL
- GET_DELTA
- VIEW_DELTA

9. **result** (output)

   Pointer to the output data area. See "Working with Returned Data" on page 36 for how to retrieve individual counter values. Use the **freeHashTable()** function to release the memory area, if the output data is no longer needed.

   If counters about locked resources are requested (with **mode** set to GET_LOCKEDRESOURCES), the output data area does not contain the REPTHLCK repeating block under the REPTHRD repeating block. Instead, the following additional information is returned in the output data area:

result	REPTHLCK	Groups all locked resources.	
		counter*	Resource-related IFCID 150 counters.
		REPLOTHR	Thread-related IFCID 150 counters. Contains one repeating block per thread.

pmInitializeStore()

This function is replaced by the **pmInitializeStoreEx()** function.

Function Call

```
pmReturnCodes pmInitializeStore (pmHost*          handle,
                                 char*            workProfile,
                                 char**           fields,
                                 unsigned int     counterNo,
                                 pmQualifierList* qualifier,
                                 char*            userData,
                                 unsigned int*    id)
```

Header File

pmGetStatThread.h

Description

This function requests the Data Collector to create a snapshot store, to initialize it, and to return a snapshot store identifier.

The snapshot store **id** is used during subsequent requests to identify the store. Optionally, you can assign a 32-character name to the store.

The snapshot store remains active in the Data Collector. It is released by the **pmReleaseStore()** function, or when the user identified by parameter **workProfile** is logged off.

Parameters

1. **handle** (input)

   The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.

2. **workProfile** (input)

   A user's work profile for which this function is to be executed. The user was specified by the **pmLogon()** function.

3. **fields** (input)

An array of DB2 counter names that the snapshot store should include. You can include statistics counter names and thread counter names that are listed in column "Field Name" of the field table.

Insert the counter names as follows. Note that counter names are not case-sensitive. Invalid counters are ignored.

```
char *fields[] = {
                   "QISEDBD",  "QTMAXDS", "QTAUCHK",  "QTAUSUC",  "QBSTMAX",
                   "QBSTPID",  "QBSTGET", "QBSTWFT",  "QBSTWFD",  "QBSTWKPD",
                   "QBSTWKPD", "QBSTMAX", "QBSTWKPD", "QBSTWDRP", "QBSTWBVQ",
                   "QBSTWFR",  "QBSTWFF", "QBSTWFD",  "QISEDSI",  "QISEDSG",
                   "QISEDSC",  "QXSTFND", "QXSTNFND", "QBSTWDRP", "QBSTWBVQ"
                 };
```

4. **counterNo** (input)

   The number of counters specified with parameter **fields**.

5. **qualifier** (input)

   The name of a qualifier list. The list specifies the criteria which data the Data Collector should return. See "How Qualifying Works" on page 33, if required.

6. **userData** (input)

   An optional user- or application-specific 32-character string that is assigned to this snapshot store. This string is returned by the **pmQueryStores()** function. If **userData** is not used, this parameter should be set to NULL.

7. **id** (output)

   A snapshot store identifier generated by the Data Collector. This identifier is required to identify this store in subsequent snapshot function calls.

Example

The following example shows how qualifiers are specified with a **pmInitializeStore()** function call, if you want the **pmGetSnapshot()** function to return only locked resources with suspensions.

```
pmHost          myHandle;
char           *workprofile = "PMUSER  GROUPID PROFILEID       TERMINALID      ";
pmReturnCodes   error;
char           *fields[] = { "QISEDBD", "QTMAXDS", "QTAUCHK" };
unsigned int    counterNo = 3;
pmQualifierList qualifier;  // the qualifier list
char           *userData = "DB2 PM snapshot store";
unsigned int    store_id;

// initialize qualifier list
initQualifierList(&qualifier);

// disable display of single and multiple held locks
// to show only locked resources with suspensions
error = addQualifier(&myHandle, &qualifier, "TLRSINGL", " ");
error = addQualifier(&myHandle, &qualifier, "TLRMULT", " ");

// initialize snapshot store with qualification
error = pmInitializeStore(&myHandle, workprofile, fields, counterNo,
                          &qualifier, userData, &store_id);
```

pmReset()

This function is replaced by the **pmResetEx()** function.

Function Call

```
pmReturnCodes pmReset (pmHost*          handle,
                       char*            workProfile,
                       pmSnapshotMode   mode
                       unsigned int     id,
                       pmTOD            snapshotTime)
```

Header File

pmGetStatThread.h

Description

This function requests the Data Collector to preset reference values for an identified snapshot store for subsequent interval processing.

Use the **pmReset()** function before you use the **pmGetSnapshot()** function in mode `GET_INTERVAL`. **pmReset()** makes the buffer **Stored** a fixed reference for interval processing.

1. Buffer **Stored** is filled with latest performance data, or data from a snapshot of the history data set.
2. Buffer **Latest** is released.

Subsequent **pmGetSnapshot()** function calls in mode `GET_INTERVAL` now use the content of buffer **Stored** to calculate the difference to buffer **Latest**. The value in buffer **Stored** is not changed by subsequent **pmGetSnapshot()** function calls in mode `GET_INTERVAL`.

The **pmReset()** function is a companion to the **pmGetSnapshot()** function, which can use actual DB2 performance data, or snapshot data from the history data set, to refresh the content of buffer **Latest**. Consequently, you also need to specify the data source for the **pmReset()** function. Use parameter **mode** to specify either data source:

- If you specify `GET_DB2`, the **pmReset()** function uses actual DB2 performance data to fill buffer **Stored**.
- If you specify `GET_HISTORY`, the **pmReset()** function uses data from a selected snapshot of the history data set to fill buffer **Stored**. The snapshot in the history data set is specified by its time stamp, in the same manner as with the **pmGetSnapshot()** function.

Parameters

1. **handle** (input)

   The platform-independent handle that identifies the TCP/IP connection to be used to transfer this request to the Data Collector. The handle was set by the **pmConnect()** function.
2. **workProfile** (input)

   A user's work profile for which this function is to be executed. The user was specified by the **pmLogon()** function.
3. **mode** (input)

   A keyword that controls the data source to use:

   - `GET_DB2` uses actual DB2 performance data.
   - `GET_HISTORY` uses snapshot data from the history data set. Specify the snapshot with parameter **snapshotTime**. See "History Processing - Get History Contents" on page 52 about how to retrieve a list of all snapshots in the history data set and how to identify a snapshot by its time stamp.
4. **id** (input)

The identification of the snapshot store for which this function is to be executed. The identification was set by the **pmInitializeStore()** function.

5. **snapshotTime** (output/input)

   If used with parameter **mode** set to `GET_DB2`, this parameter returns a time stamp, which shows when the buffer **Stored** was filled with latest DB2 performance data.

   If used with parameter **mode** set to `GET_HISTORY`, specify a valid time stamp of a snapshot in the history data set that you want to use.

   The time stamp is in Store Clock format. If you need a conversion to **time_t** format, see "Convert Store Clock Format to time_t Format" on page 91 for details.

Appendix E. Notices

This information was developed for products and services offered in the U.S.A.IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Deutschland
Informationssysteme GmbH
Department 3982
Pascalstrasse 100

70569 Stuttgart
Germany

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States, or other countries, or both:

AIX
CICS
DATABASE 2
DB2
IBM

MVS
OS/390
RACF
VisualAge

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

Bibliography

IBM DB2 Performance Monitor for OS/390 Version 6 Online Monitor User's Guide, SC26-9168

IBM DB2 Performance Monitor for OS/390 Version 6 Batch User's Guide, SC26-9167

IBM DB2 Performance Monitor for OS/390 Version 6 Command Reference, SC26-9166

IBM DB2 Performance Monitor for OS/390 Version 6 Messages, SC26-9169

IBM DB2 Performance Monitor for OS/390 Version 6 Using the Workstation Online Monitor, SC26-9170

IBM DB2 Performance Monitor for OS/390 Version 6 Installation and Customization, SC26-9171

IBM DB2 Performance Monitor for OS/390 Version 6 General Information, GC26-9172

Program Directory for IBM DB2 UDB Server for OS/390 DB2 Performance Monitor DB2 Workstation Analysis and Tuning Version 6, GI10-8183

IBM DB2 Universal Database Server for OS/390 Version 6 Administration Guide, SC26-9003

IBM DB2 Universal Database Server for OS/390 Version 6 Command Reference, SC26-9006

IBM DB2 Universal Database Server for OS/390 Version 6 Application Programming and SQL Guide, SC26-9004

IBM DB2 Universal Database Server for OS/390 Version 6 Messages and Codes, GC26-9011

OS/390 Security Server (RACF) - Introduction, GC28-1912

OS/390 MVS System Codes, GC28-1780

ESA/390 Principles of Operation, SA22-7201

TCP/IP for MVS: Application Programming Interface Reference, SC31-7187

TCP/IP Tutorial and Technical Overview, GG24-3376

Index

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

W

Readers' Comments — We'd Like to Hear from You

DB2 Performance Monitor for OS/390
Data Collector Application Programming Interface Guide
Version 6

Publication No. SC26-9173-01

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	☐	☐	☐	☐	☐

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Name

Address

Company or Organization

Phone No.

Readers' Comments — We'd Like to Hear from You
SC26-9173-01

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

PLACE
POSTAGE
STAMP
HERE

IBM Deutschland Entwicklung GmbH
Information Development, Dept. 0446
Schoenaicher Strasse 220
71032 Boeblingen
Germany

Fold and Tape **Please do not staple** Fold and Tape

SC26-9173-01

Cut or Fold
Along Line

Program Number: 5645-DB2

Printed in the United States of America

SC26-9173-01