

Enterprise Transport API

C Edition

3.8.0.L1

DACSLOCK LIBRARY

Document Version: 3.8.0
Date of issue: April 2024
Document ID: ETAC380REDAC.240



© **Refinitiv 2016 - 2024**. All rights reserved.

Republication or redistribution of Refinitiv content, including by framing or similar means, is prohibited without the prior written consent of Refinitiv. 'Refinitiv' and the Refinitiv logo are registered trademarks and trademarks of Refinitiv.

Any software, including but not limited to: the code, screen, structure, sequence, and organization thereof, and its documentation are protected by national copyright laws and international treaty provisions. This manual is subject to U.S. and other national export regulations.

Refinitiv, by publishing this document, does not guarantee that any information contained herein is and will remain accurate or that use of the information will ensure correct and faultless operation of the relevant service or equipment. Refinitiv, its agents, and its employees, shall not be held liable to or through any user for any loss or damage whatsoever resulting from reliance on the information contained herein.

Contents

1	Introduction	1
1.1	Product Description	1
1.2	IDN Versus the Refinitiv Data Platform	1
1.3	Audience	1
1.4	Organization of Manual	2
1.5	References	2
1.6	Conventions	2
1.6.1	<i>Typographic</i>	2
1.6.2	<i>Programming</i>	2
1.7	Glossary	2
2	Compliant Source Server Applications	5
2.1	Overview	5
2.2	Establish Subservice Names	5
2.3	Define Entitlement Codes	5
2.4	Create and Write Locks	5
2.5	Publish the Map	6
2.6	Read the Map and Supply it to a Data Access Control System Station	6
3	DACSLock API	7
3.1	DACSLock Operation	7
3.2	Forming a DACSLock	8
3.3	What a DACSLock Contains	9
3.4	Compression of DACSLocks	10
3.5	Compounding DACSLocks	11
3.6	Transport of DACSLocks	11
3.7	Compound DACSLock in Relation to Permissioning	12
4	DACSLock API Interface	13
4.1	DACS_CsLock() Function	13
4.1.1	<i>Synopsis</i>	13
4.1.2	<i>Function Arguments</i>	13
4.1.3	<i>Data Structure: COMB_LOCK_TYPE</i>	14
4.1.4	<i>Data Structure: DACS_ERROR_TYPE</i>	14
4.1.5	<i>Return Values</i>	14
4.2	DACS_GetLock() Function	15
4.2.1	<i>Synopsis</i>	15
4.2.2	<i>Function Arguments</i>	15
4.2.3	<i>Data Structure: PRODUCT_CODE_TYPE</i>	16
4.2.4	<i>Data Structure: DACS_ERROR_TYPE</i>	16
4.2.5	<i>Return Values</i>	16
4.3	DACS_CmpLock() Function	17
4.3.1	<i>Synopsis</i>	17
4.3.2	<i>Function Arguments</i>	17
4.3.3	<i>Return Values</i>	17
4.4	DACS_perror() Function	18
4.4.1	<i>Synopsis</i>	18
4.4.2	<i>Function Arguments</i>	18
4.4.3	<i>Return Values</i>	18

Appendix A Example Program..... 19

1 Introduction

1.1 Product Description

The goal of permissioning is to control access to data by users. Using an entitlement system such as Data Access Control System, permission profiles can be defined identifying what each user is allowed to access. Data Access Control System is the entitlement system for Refinitiv Real-Time Distribution System. Data Access Control System permission checks take place in the Market Data Client application. In order to perform a permission check for an item, Data Access Control System must have 'requirements' information for the item and a profile for the user (a.k.a. content based permissioning).

Data Access Control System requirements information is organized as numeric expressions. Each subservice of a service, e.g. all the data from an exchange, is assigned a (series of) numeric **entitlement codes (PEs)**. The numeric entitlement codes are transported with items as they move from source servers to Market Data Client applications on the Refinitiv Real-Time Distribution System. In order to accommodate the most general case in which information from multiple sources is combined to form new items (such as in compound servers), the **requirements** information for an item is a Boolean expression containing these entitlement codes. For an item obtained directly from a source, this expression is usually a single term. When a compound server combines two items to form a new (compound) item, it must also combine the requirements information to form a new (compound) requirement.

The name of the subservice associated with an entitlement code is maintained in tables within the Data Access Control System database and operational permission checking subsystem. The table that relates the entitlement codes for a service to the subservice names for that service is called the **map** for the service.

Requirements are transported on Refinitiv Real-Time Distribution System in protocol messages called **locks**. The DACSLock API provides functions to manipulate locks in a manner such that the source application need not know any of the details of the encoding scheme or message structure. For a source server to be Data Access Control System compliant, based on content, it must publish locks for the items it publishes; i.e., the source server application must produce lock events. Any item published without a lock or with a null lock is available to everybody permissioned for that service, even those without subservice permissions.

If a source server introduces (new) data to Refinitiv Real-Time Distribution System that originates outside the network on which the application is running, then the application developer is also responsible for providing the map information for the service.

In addition to source servers that publish new data directly from a vendor, there are also compound servers that gather market information from other sources, manipulate that information, and then re-publish it on Refinitiv Real-Time Distribution System. These servers must read locks from the Enterprise Transport API C Edition to combine them before publishing compound items. Again, the DACSLock API provides functions to assist with this process. Compound sources must also use a special form of user ID when connecting to the Refinitiv Real-Time Distribution System network.

NOTE: Subject-based sources do not require locks.

1.2 IDN Versus the Refinitiv Data Platform

Refinitiv's ultra-high speed network the Refinitiv Data Platform has replaced the older IDN network. All references herein are made to Refinitiv Data Platform. However, for historical reasons in the Data Access Control System administrative screens, this network is still referred to as IDN. For this reason, the terms Refinitiv Data Platform and IDN are interchangeable throughout this document.

1.3 Audience

This guide is intended for software programmers who wish to incorporate the DACSLocks into the development of their source applications.

1.4 Organization of Manual

The material presented in this guide is divided into the following sections:

CHAPTER	CONTENT / TOPIC
Chapter 1, Introduction	Document description.
Chapter 2, Compliant Source Server Applications	For subservice-level permissioning, a Data Access Control System-compliant source server must satisfy a series of requirements.
Chapter 3, DACSLock API	The data flow of a Data Access Control System Lock and its operation are depicted.
Chapter 4, DACSLock API Interface	Description and structural definitions of the DACSLock API functions.
Chapter A, Example Program	Lists an example program that was created using the DACSLock API.

Table 1: Manual Overview

1.5 References

- *Enterprise Transport API C Edition Developers Guide*
- *DACSLock API Reference Manual*
- The [Refinitiv Professional Developer Community](#)

1.6 Conventions

1.6.1 Typographic

- C/C++ functions, arguments, and data structures are shown in **Courier New** font.
- Parameters, filenames, tools, utilities, and directories are shown in **Bold** font.
- Document titles and variable values are shown in *italics*.
- When initially introduced, concepts are shown in ***Bold, Italics***.
- Longer code examples (one or more lines of code) are shown in Courier New font against an orange background. Code comments are shown in green font color. For example:

```
/* calculate the length of the new lock */
int lock = dacsInterface.calculateLength(serviceId, operation,
    productEntityList, productEntityListLength, error);
```

1.6.2 Programming

Enterprise Transport API C Edition Standard conventions were followed.

1.7 Glossary

TERM	DESCRIPTION
API	Application Programming Interface
Application	A program that accesses data from and/or publishes data to the system.

Table 2: Glossary and Acronyms

TERM	DESCRIPTION
Compound Item	A data item prepared from data items retrieved from the system.
Concrete service	A set of real-time data items published by a source server. Each concrete service is identified on Data Access Control System by a unique name (known as a network).
Data Access Control System	A tool that controls the sets of data available to specified users on your RDMS.
Refinitiv Data Platform	Refinitiv's open, global, ultra-high-speed network and hosting environment, which allows users to access and share various types of content.
Entitlement Code	If a vendor service is permissioned down to the subservice level, an entitlement code must be provided with each item. Based on mapping tables provided by the vendor, Data Access Control System uses this code to determine the information provider and/or vendor product associated with an item.
Exchange	A commercial establishment at which or through which trading of financial instruments takes place. Exchanges are information providers.
Exchange Map Logic	The logic used to construct requirements when an item is supplied by more than one exchange. If OR logic is used, the user only needs permission to access one of the exchanges that supplies the item. If AND logic is used, the user must have permission to access all of the exchanges that supply the item.
Map Program	An application associated with a particular vendor service which requests permissioning data from the vendor host and then uses that data to construct various mapping tables required by the Data Access Control System.
Mapping Tables	These tables are used by the Data Access Control System to derive the requirement for an item. They map entitlement codes to vendor products and, if applicable, to information providers (exchanges and specialist services).
Network Service	See concrete service.
PE	Permissionable Entity. A number used to designate the permissioning basis of a data item on Refinitiv Data Platform (or Refinitiv Real-Time Distribution System). (Same as entitlement code.)
Permissioning	The control of access to and publication of data items by users.
Product	A subset of the data items delivered by an information vendor for which there is a single charge (based on vendor criteria).
Product Map Logic	The logic used to construct requirements when an item is supplied by more than one product. If OR logic is used, the user only needs permission to access one of the products that supplies the item. If AND logic is used, the user must have permission to access all of the products that supply the item.
Profile	Information, including a list of subservices, that is used during permission checking. There is a profile associated with each user.
Service	This term is used in two ways by the Data Access Control System in a market data system environment. See concrete service and vendor service.
Service ID	A unique numeric ID assigned to each network service. On RDMS, all valid service IDs for a particular system are listed in the global configuration file rmids.cnf .
Source	An application or server capable of supplying or transmitting information.
Source Server	An application program which provides a concrete service. More than one source server can provide the same concrete service.
Specialist Data Service	A set of data items provided by a third party (i.e. not from an exchange and not from the vendor delivering the data items to the site).

Table 2: Glossary and Acronyms (Continued)

TERM	DESCRIPTION
Subservice	A named set of items delivered by a vendor which are authorized as a group; e.g. all instruments traded on NYSE or all items that make up the product Securities 2000.
Subservice Type	There are three types of subservices: <ul style="list-style-type: none"> • Products • Exchanges • Specialist Service
User	A person with a unique, system-wide name.
Vendor Service	<p>A vendor may offer more than one type of data delivery service to a customer. For example, Refinitiv provides the IDN service. Each vendor service is identified on the Data Access Control System by a unique name.</p> <p>Each vendor service is associated with one or more concrete services. For example the IDN service may be published on the network by any combination of these concrete services: IDN Selectserver and Reuters Data Feed.</p>

Table 2: Glossary and Acronyms (Continued)

2 Compliant Source Server Applications

2.1 Overview

For permissioning at the subservice level to work, it is necessary to have information about the permissioning requirements for an item at locations other than just the source. For the Data Access Control System to permission a source service below the service level, the source must:

1. Contribute to the process by creating locks containing permissioning information.
2. Provide data mapping the entitlement codes in the locks to subservice names.

For subservice level permissioning, a the Data Access Control System-compliant source server must:

- Establish Subservice Names
- Define Entitlement Codes
- Create and Write Locks
- Publish the Map
- Read the Map and Supply it to a Data Access Control System Station

2.2 Establish Subservice Names

If a source server provides a service that is subdivided into subservices, the first requirement is that each subservice have a symbolic name. For Refinitiv Data Platform (i.e., IDN) such names are the names of Refinitiv products, exchanges, and specialist data services. These subservice names represent subsets of the service being provided by the vendor. An item may be in more than one subservice, and an item might not be in any named subservice. Whenever a source publishes an item on RDMS, the source must be able to associate with the item the identities of any subservices to which the item belongs. For example, an Refinitiv Data Platform source might identify that an item belongs to the subset of items from the New York Stock Exchange and is part of the Equities 2000 product.

The reason for the symbolic names is that these names are used by the system administrator to grant or deny access by users to items in the various subservices. The system administrator performing permissioning will not deal with arbitrary numeric encodings such as the Reuters PE (Permissionable Entity).

2.3 Define Entitlement Codes

The second requirement is that a number, called the entitlement code, must be associated with each subservice name. (A subservice can have one or more associated entitlement codes.) At the time an item is published, the source designates the subservice(s) to which the item belongs as a Boolean expression in entitlement codes. The reason for the entitlement codes is that it may be necessary to combine permissioning information from more than one source in order to permission compound items made up of constituents from more than one source.

The PE used for permissioning on IDN (FID 1, PROD_PERM) is an example of an entitlement code.

The list of subservice names and associated entitlement codes is called the *map* for the source.

2.4 Create and Write Locks

At the time a source server opens a data stream, it must write a lock containing the entitlement code expression associated with the item. The source server must use an existing Refinitiv API to publish permissionable data on the Refinitiv Real-Time Distribution System.

One of the capabilities of such an API is to create the lock. The arguments to the API function include a list of entitlement codes and the Boolean operator (AND or OR) that indicates how they are to be logically combined. After the lock is created, it needs to be posted to the Refinitiv Real-Time Distribution System.

A source application can post a revised lock at any time.

2.5 Publish the Map

The fourth requirement is that the source must publish as data items the map of its entitlement codes and subservice names (or use **map_generic** to load map through the use of a file). As an example, the map for the Refinitiv Data Platform service is published as a series of RICs referred to as the Reuters Product Definition Pages.¹

Within the map data there must be a readily available data item that contains a date-time stamp. The value of this date-time stamp must be the date and time when the last change was made to the map. The objective is to permit the application that reads the map to read a single item and determine if the rest of the data has changed and thus needs to be reread.

The map items (records or pages) must be available to a user and application that have no subservice permissions. This is necessary so that the map can be retrieved at a new site that does not yet have permissions distributed.

NOTE: For Refinitiv Data Platform services, template files containing preliminary mapping information are provided with the Data Access Control System software so that subservice permissioning can be set up before the latest map is retrieved from the source. If a template file is not provided with a third-party source application, it is important to not require subservice permissioning so that the map can be retrieved from the source.

2.6 Read the Map and Supply it to a Data Access Control System Station

There are two ways to load map data:

- Use the Generic map collect program (**map_generic**).
- Download the map.

In order for the Data Access Control System administrator to be able to assign permissions to the subservices for a service and for the Data Access Control System operational subsystem to perform permission checks, the database must contain the map information for the source. As indicated previously, it is expected that the source will publish this map. Further it is expected that the source application developer will provide an application (known as the map collection program) to:

- Request the map items from the source.
- Determine if there were any changes since the previous map was received.
- Convert any revised information into a file that can be loaded into the Data Access Control System database.

The source application developer may also want to provide a map monitor program which can be scheduled to run periodically to retrieve the latest map and see if any changes have occurred since the last map collection (by checking the date-time stamp). Based on the status reported by the monitor program, the administrator knows when the map collection program needs to be run.

The Data Access Control System does not care about the format of the map published by the source as long as the map collection program for that source produces an appropriately formatted permission map file.

For further details on the map collect and proper permission map file formatting, refer to the *DACSLock API Reference Manual* specific to the version of the Data Access Control System that you run.



TIP: The Data Access Control System software package comes with a map collection program for the Refinitiv Data Platform service.

1. Refer to the *Refinitiv Product Definition Pages User Guide* to see how Refinitiv communicates permissioning information for its products.

3 DACSLock API

3.1 DACSLock Operation

The DACSLock contains requirements for an item that a vendor source deems necessary. On the Market Data Client LAN, users are entitled to specific capabilities. Therefore, when a Lock, which contains requirements, is tested against the capabilities of the user, enough information is available to permission the following:

PERMISSIONING OPERATION
USER -> APPLICATION
USER -> SERVICE
USER -> SUB-SERVICE (entitlement codes)

Table 3: Data Access Control System Permissioning Capabilities

DACSLocks are critical to the operation of the Data Access Control System for content-based sources. Subject-based sources do not require locks. The DACSLock contains the requirements for the requested item. The data flow of a DACSLock and its operation are depicted in the remaining sections of this chapter.

3.2 Forming a DACSLock

The Source Server is responsible for creating a DACSLock. What information is encoded within the DACSLock is vendor specific. Two examples are presented to clarify the formation of DACSLocks with respect to a Source Server. Figure 1, "Forming a DACSLock for Refinitiv Real Time Direct," demonstrates the input requirements, the DACSLock API to be called, and the transport mechanism of the DACSLock from the Source Server to the Market Data Client application.

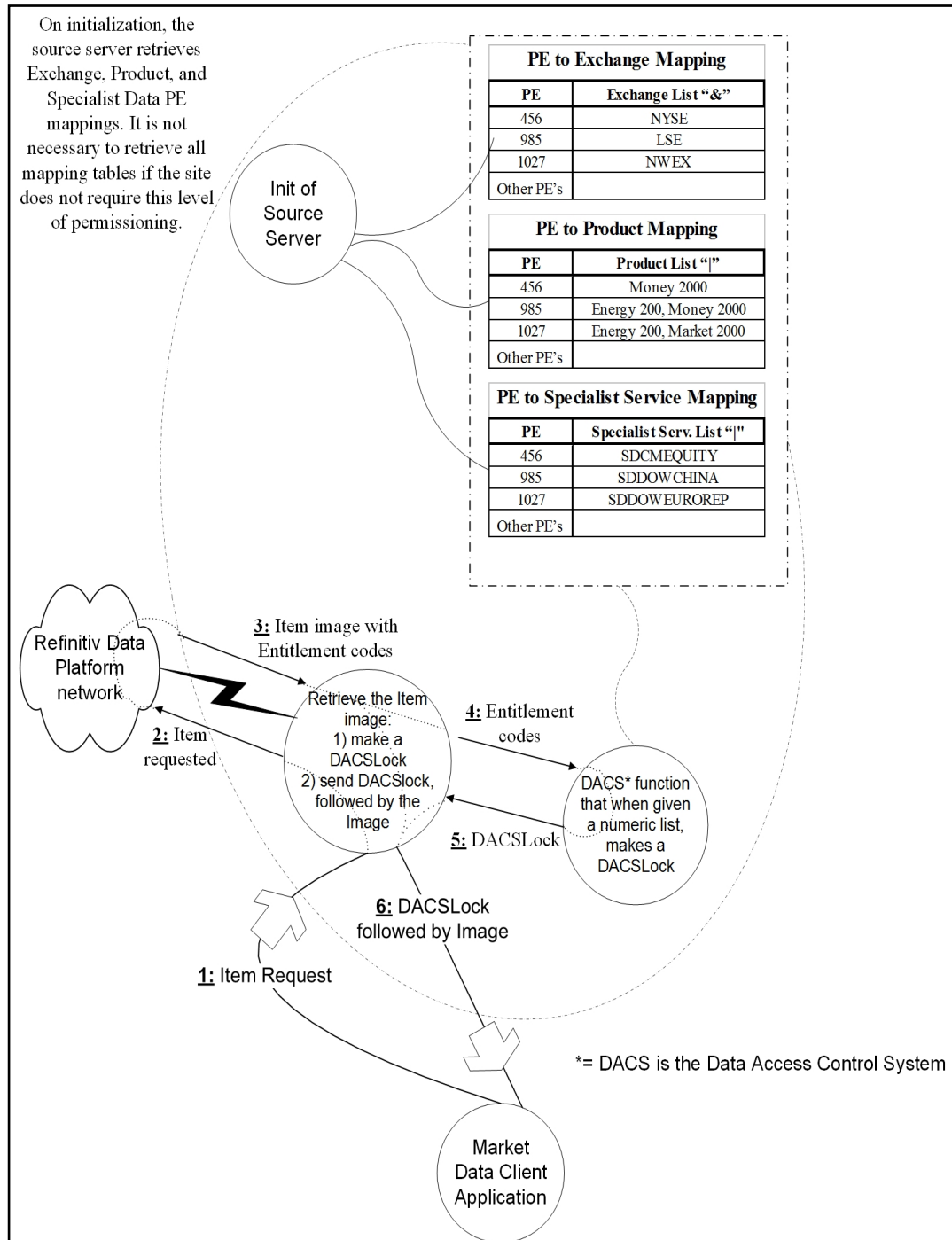


Figure 1. Forming a DACSLock for Refinitiv Real Time Direct

Figure 1 presents the following information:

- The Data Access Control System station via use of its map collect utility is responsible for retrieving the Exchange, Product, and Subservice mappings. For this mapping to be dynamic, it is necessary for the mapping tables to be retrieved from the datafeed line by the Refinitiv Data Platform Server.
- It is up to the Refinitiv Data Platform Server if it retrieves all the mapping tables based on the customer site requirements and on the capabilities of the server's datafeed line.
- The DACSLock API function that makes the DACSLock is supplied the list of entitlement codes that corresponds to the Item. For most Items this will be only one PE. However, in the case of a NEWS2000 Item, the PE list could be up to 256 entitlement codes. A second parameter to this DACSLock API function is a single operator to be assigned to this entitlement codes list. This is required in the case of NEWS2000 so that the PE list can be assigned an **OR** operator. The other possible operator that can be assigned to the entitlement codes list is the **AND** operator that may be required by other Source Servers.
- The DACSLock must be sent before the Image. This is required so that the Market Data Client application (via the Enterprise Transport API) can permission the Item as soon as the Image arrives, instead of having to hold the Image and await the Lock.

3.3 What a DACSLock Contains

The DACSLock needs to contain that information relevant to the requirements for the Item requested. Since a DACSLock needs to be the minimal size it can to minimize communication costs, a source server encodes the requirements into a single numeric number. In Figure 1, a mapping table is created by the Source Server mapping the textual requirements to a numeric value. By supplying a operator to the DACSLock API function that creates DACSLocks, complex requirements can be created by the source server. Referring to Figure 1, and examining the mapping tables, the following requirements can be formulated:

ENTITLEMENT CODES	REQUIREMENTS
456	NYSE & Money 2000 & (NY LONDON)
985	LSE & (Energy 2000 Money 2000) & NY
1027	NWEX & (Energy 2000 Markets 2000) & LONDON

Table 4: Item Requirement Formulations

The item contains the PE and therefore, by using the mapping tables, also contains permissioning requirements. A source server can also add requirements to the DACSLock for an item by adding extra entitlement codes to the PE list supplied to the appropriate DACSLock API class. For example, if the source server imposes that only a **Page_Call** application can use this item, then the source server makes a new PE with that requirement and **AND's** it to the PE for that item.

ENTITLEMENT CODES	REQUIREMENTS
5000	Page_Call

Table 5: PE Example that can Add Extra Requirements to an Item

So if an Item has a PE = 456 and the source server requires only **Page_Call** access, then the source server supplies entitlement codes 456 and 5000 as the parameters passed to the appropriate DACSLock API function that builds DACSLocks from PE lists.

3.4 Compression of DACSLocks

To minimize the physical size of a DACSLock it is compressed. The type of compression is based on Binary Coded Decimal (BCD) algorithm. For example, the DACSLock API is required to make a lock that has the following PE requirements:

REFINITIV DATA PLATFORM	BRIDGE
1027 & (456 985) & 5000	1 & 2

Table 6: PE Requirements for a Compound Item

What the compression algorithm performs is an interpretation of operator functions and the BCD conversion of the numeric PE values using the following table:

OPERATION	REPLACED NIBBLE VALUE
numeric 0 – 9	0 - 9
"&" and operation	A
" " or operation	B
"EOF" End of DACSLock	C
"EOS" End of Source Server PE List	D

Table 7: Operator and BCD Interpretation Table

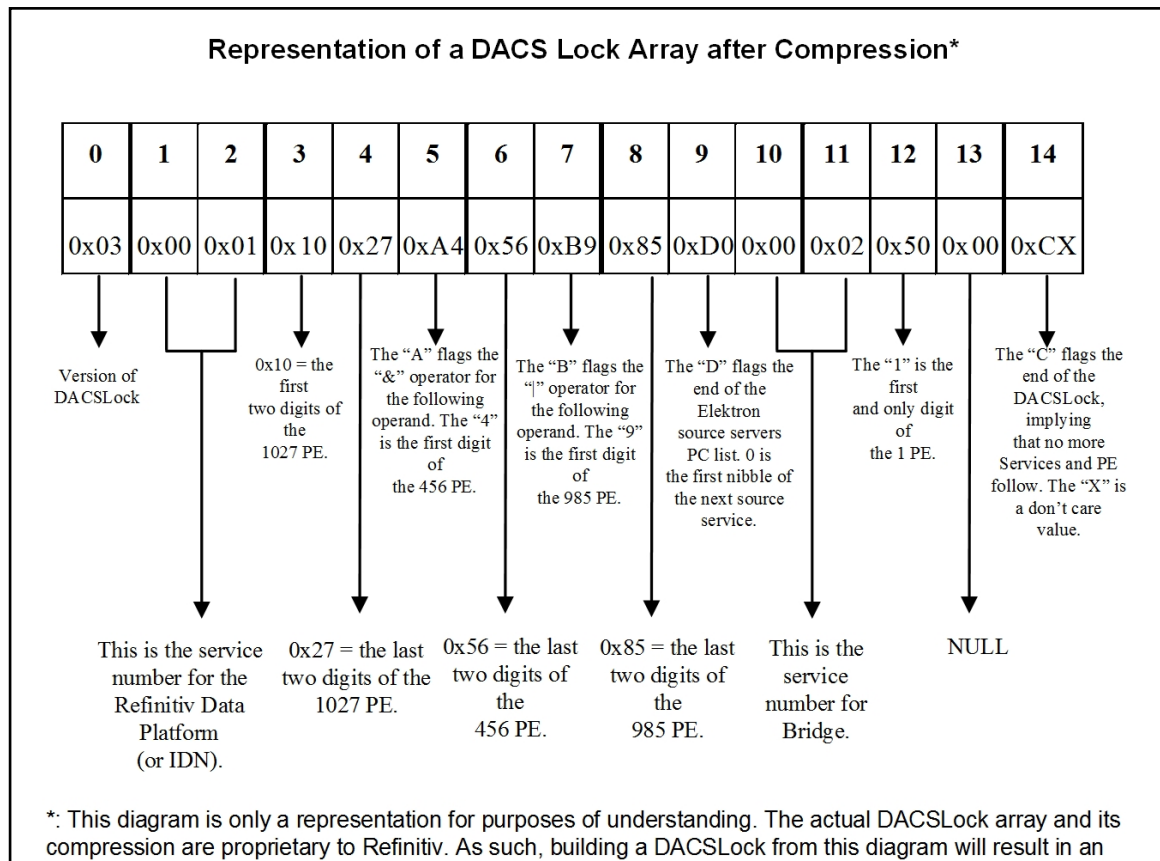


Figure 2. DACSLock Array after Compression

By using the Operator and BCD Interpretation Table, the above unsigned character array is created. This array shows some of the advantages of this compression. First, it makes no stipulation to the maximum value of a PE, thus not imposing a limitation of PE values, other than a possible machine maximum that can be manipulated easily with the software language used, i.e. $(2^{32})-1$ for an unsigned long. The second advantage is the compression is simple and not computational intensive for fast compression and decompression. Third, the compressed DACSLock is smaller than the actual ASCII string if it were sent.

3.5 Compounding DACSLocks

The previous example, in Figure 2, had two source server PE lists within the DACSLock. This situation is possible when a compound server combines the Items from more than one source server type. To compound a DACSLock, the compound server calls the DACSLock API function that, when given the constituent Item DACSLocks, will create a DACSLock that contains all of the constituent Item requirements. To minimize the size of a DACSLock, the DACSLock API uses Boolean algebra rules to minimize the entitlement codes within the PE list wherever possible. Some of the rules are:

```
(A | B) & A = A
A & A & B = A & B
A | A | B = A | B
```

The compound server stores all constituent Item DACSLocks until the Item is no longer required. This functionality is required in the event a new DACSLock is received by the compound server for an Item that it has open, thus requiring the compound server to update the compound item DACSLock of which the received DACSLock is part. Then the compound server is required to forward its new compound Item DACSLock to those servers that have the compound item open.

3.6 Transport of DACSLocks

DACSLocks for compound servers might grow larger than the maximum size of a single message, in which case the server splits the DACSLock across multiple messages.

3.7 Compound DACSLock in Relation to Permissioning

The previous sections explain the rules for constructing and transporting the DACSLocks. Figure 3 shows what the data flow functionality of a DACSLock is in an operating environment with two Source Servers, one Compound Server, and a Market Data Client application. In this example the Market Data Client application is requesting an item from a Compound Server. This item is made from the constituent items from the Refinitiv Real Time Direct (represented as **RDF Server** in the following diagram) and Bridge Servers.

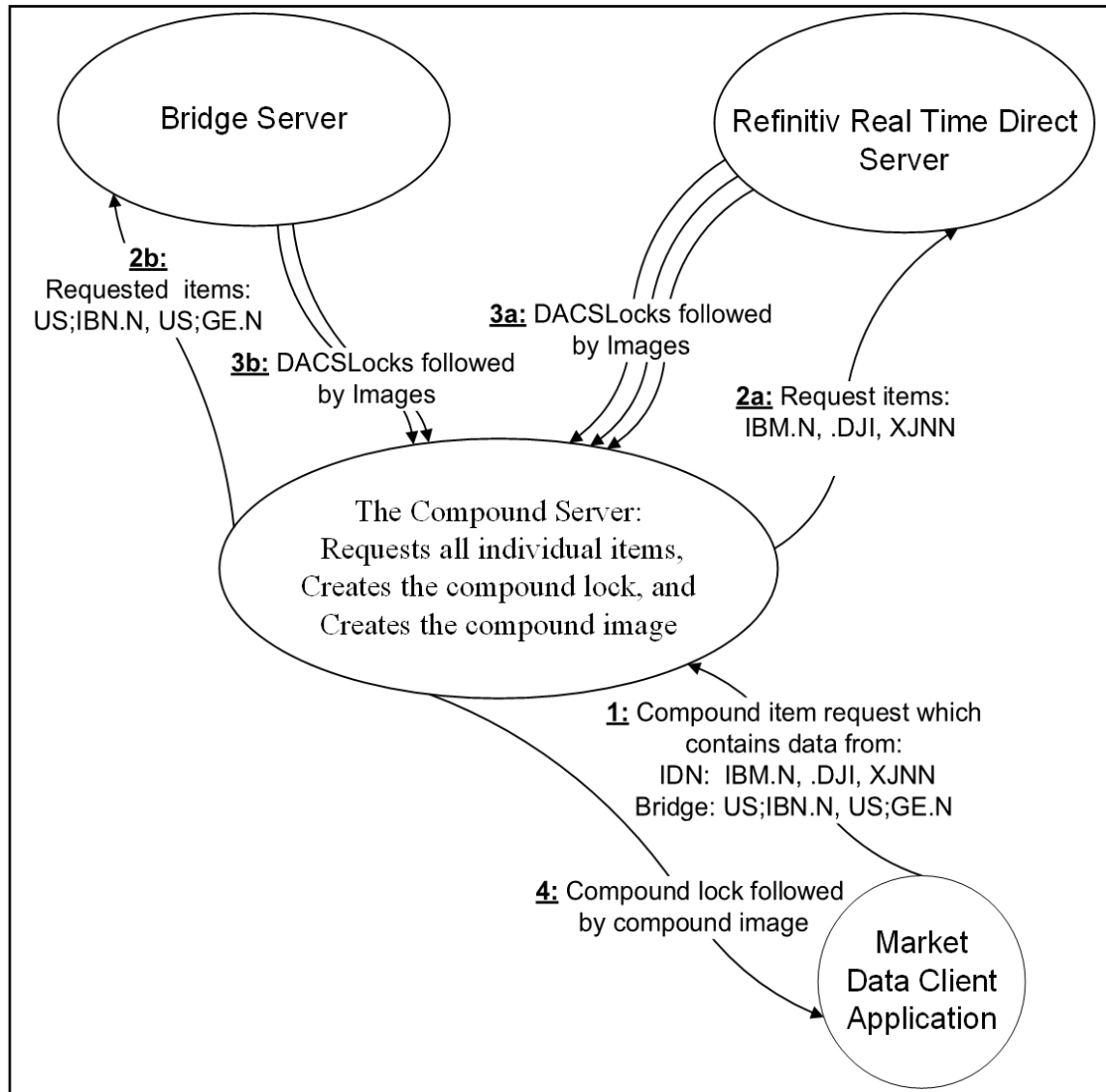


Figure 3. Compound Locks

4 DACSLock API Interface

This chapter provides descriptions and structural definitions of the following functions:

- `DACS_CsLock()` Function
- `DACS_GetLock()` Function
- `DACS_CmpLock()` Function
- `DACS_perror()` Function

4.1 DACS_CsLock() Function

4.1.1 Synopsis

The **DACS_GetLock()** function combines a list of access locks into a single composite Data Access Control System access lock. You use this function to form complex access locks from constituent access locks that were created by a previous call to the **DACS_CsLock()** or **DACS_GetLock()** functions.

```
o
DACS_CsLock (nm_channel, item_name, newLock, newLockLen, lockList, Dacs_Error)
    int          nm_channel
    char         *item_name
    unsigned char **newLock
    int          *newLockLen;
    COMB_LOCK_TYPE lockList[];
    DACS_ERROR_TYPE *Dacs_Error
```

4.1.2 Function Arguments


ARGUMENT	DESCRIPTION
nm_channel	nm_channel is no longer used. It  WARNING! For backward compatibility, nm_channel must be set to 0 (zero).
item_name	For backward compatibility, item_name must be an empty string ("").
newLock	newLock is a pointer to an unsigned char pointer that shall point to the generated access lock for the "item" built by the source/compound server. If the pointer is NULL on entry to the DACS_CsLock() function, space for the new access lock will be dynamically allocated using malloc() . It is the responsibility for the caller of DACS_CsLock() to free() the memory allocated when the newly generated access lock is no longer required.
newLockLen	newLockLen reflects the length of the newly generated lock. If the newLock parameter does not point to a NULL pointer, then the source/compound server application must supply the *newLockLen with the maximum size of the user-supplied access lock pointer. If the *newLockLen parameter supplied by the source/compound server application is less than the length required to fit the access lock, a PERM_FAILURE error is returned.
lockList	lockList is a pointer to an array of pointers to the access locks of associated component items to be combined by the source/compound server. A NULL pointer terminates the list.
Dacs_Error	Dacs_Error points to a DACS_ERROR_TYPE data structure in which returned errors will be placed.

Table 8: DACS_CsLock() Functional Arguments

4.1.3 Data Structure: COMB_LOCK_TYPE

The **COMB_LOCK_TYPE** data structure is defined to be as follows:

```
typedef struct {
    int          server_type;
    char         *item_name;
    unsigned char *access_lock;
    int          lockLen;
} COMB_LOCK_TYPE;
```

Where:

- **server_type** must be set to 0 (zero).
- **item_name** is a NULL pointer (for backward compatibility).
- **access_lock** is a pointer to the user-supplied access lock that is to be used as one of the constituent item access locks.
- **lockLen** is the length of the access lock. This value was previously returned from a **DACS_GetLock()** function.

4.1.4 Data Structure: DACS_ERROR_TYPE

The **DACS_ERROR_TYPE** data structure is defined to be as follows:

```
typedef struct {
    short dacs_error;
    short dacs_suberr;
} DACS_ERROR_TYPE;
```

4.1.5 Return Values

DACS_GetLock() returns:

- **DACS_SUCCESS** if the function did not encounter a fatal error, and the **newLock** and **newLockLen** parameters shall be populated.
- **DACS_FAILURE** is returned if a fatal, unrecoverable error was encountered. An ASCII explanation of the error can be further determined by passing the **Dacs_Error** data structure to the **DACS_perror()** function.

4.2 DACS_GetLock() Function

4.2.1 Synopsis

The **DACS_GetLock()** function creates a Data Access Control System access lock from a list of entitlement codes (i.e., codes specified by the data vendor on which all permissioning is based).

```
DACS_GetLock      (service_type, ProductCodeList, LockPtr, LockLen, Dacs_Error)
    int            service_type;
    PRODUCT_CODE_TYPE *ProductCodeList;
    unsigned       char **LockPtr;
    int            *LockLen;
    DACS_ERROR_TYPE *Dacs_Error;
```

4.2.2 Function Arguments

ARGUMENT	DESCRIPTION
service_type	service_type passes the service type of the server that is to be encoded into the Data Access Control System access lock. This value is the unique numeric ID assigned to a service by its serviceld parameter in the global configuration file used by Refinitiv Real-Time Distribution System's core infrastructure components.
ProductCodeList	ProductCodeList parameter passes the list of entitlement codes which shall be encoded into the Data Access Control System access lock.
LockPtr	<p>The LockPtr parameter is a pointer to an unsigned char pointer that shall point to the generated access lock that represents the entitlement code list in DACSLock format. If the pointer is NULL on entry to the DACS_GetLock() function, space for the new access lock will be dynamically allocated using malloc(). It is the responsibility for the caller of DACS_GetLock() to free() the memory allocated when the newly generated access lock is no longer required.</p> <p>NOTE: If the DACS_GetLock() function returns an error, then no space for the access lock will have been allocated.</p>
LockLen	The LockLen parameter is a pointer to an int. This parameter is updated to reflect the length of the newly generated access lock. If the LockPtr parameter does not point to a NULL pointer on entry, then the user must supply the LockLen with the maximum size of the user-supplied access lock buffer. If the user-supplied LockLen is less than the length required to fit the access lock, then a DACS_FAILURE error is returned.
Dacs_Error	The Dacs_Error parameter points to a DACS_ERROR_TYPE data structure in which returned errors will be placed.

Table 9: DACS_GetLock() Functional Arguments

4.2.3 Data Structure: **PRODUCT_CODE_TYPE**

The **PRODUCT_CODE_TYPE** data structure is defined to be as follows:

```
typedef struct {
    char                operator;
    unsigned short      pc_listLen;
    unsigned long       pc_list[/* pc_listLen */];
} PRODUCT_CODE_TYPE;
```

where:

- **operator**: is the user-supplied operation that shall be imposed on the entitlement code list that is supplied. The operator field must have one of the following formats:
 - The ampersand character ‘&’ (for an “AND” list): The implication is that the user of the item must be permissioned for all the entitlement codes contained within the access lock.
 - The pipe character ‘|’ (for an “OR” list): The implication is that the user of the item only needs access to any one of the entitlement codes contained within the access lock.
- **pc_listLen**: is used to flag the number of entries that are contained within the **PRODUCT_CODE_TYPE** array.
- **pc_list**: is a numerically ascending sorted list of **unsigned long** values that shall be interpreted as the entitlement codes to be encoded into a Data Access Control System access lock.

4.2.4 Data Structure: **DACS_ERROR_TYPE**

The **DACS_ERROR_TYPE** data structure is defined to be as follows:

```
typedef struct {
    short dacs_error;
    short dacs_suberr;
} DACS_ERROR_TYPE;
```

4.2.5 Return Values

DACS_GetLock() returns:

- **DACS_SUCCESS** if the function does not encounter a fatal error.
- **DACS_FAILURE** if a fatal, unrecoverable error was encountered. An ASCII explanation of the error can be further determined by passing the **Dacs_Error** data structure to the **DACS_perror()** function.

4.3 DACS_CmpLock() Function

4.3.1 Synopsis

The **DACS_CmpLock ()** function compares two access locks for equality. You can use this function to verify whether a new access lock is different from a previously-generated access lock, thus reducing the possible overhead incurred in redistribution of an unchanged access lock.

```
DACS_CmpLock (Lock1Ptr, Lock1Len, Lock2Ptr, Lock2Len, Dacs_Error)
    unsigned char    *Lock1Ptr;
    int              Lock1Len;
    unsigned char    *Lock2Ptr;
    int              Lock2Len;
    DACS_ERROR_TYPE  *Dacs_Error;
```

4.3.2 Function Arguments

ARGUMENT	DESCRIPTION
Lock1Ptr	The Lock1Ptr parameter is a pointer to the first access lock that is to be compared.
Lock1Len	The Lock1Len parameter is an integer. This parameter is the length of the first access lock that is to be compared.
Lock2Ptr	The Lock2Ptr parameter is a pointer to the second access lock that is to be compared.
Lock2Len	The Lock2Len parameter is an integer. This parameter is the length of the second access lock that is to be compared.

Table 10: DACS_CmpLock() Functional Arguments

4.3.3 Return Values

DACS_CmpLock () returns:

- **DACS_SUCCESS** if the function did not encounter a fatal error and the access locks were logically identical.
- **DACS_DIFF** if a fatal error was not encountered but the two access locks were logically different.
- **DACS_FAILURE** if a fatal, unrecoverable error was encountered. An ASCII explanation of the error can be further determined by passing the **Dacs_Error** data structure to the **DACS_perror ()** function.

4.4 DACS_perror() Function

4.4.1 Synopsis

The **DACS_perror()** function creates a textual message describing the last error generated by a Data Access Control System Library call. This message is saved to the supplied buffer. The error number is taken from the location **Dacs_Error** -> **dacs_errno** variable pointed to by the user application, when a library function returns a **DACS_FAILURE**.

```
DACS_perror (err_buffer, buffer_len, text, Dacs_Error);
    unsigned char    *err_buffer;
    int              buffer_len;
    unsigned char    *text;
    DACS_ERROR_TYPE  *Dacs_Error;
```

4.4.2 Function Arguments

ARGUMENT	DESCRIPTION
err_buffer	err_buffer contains the destination address for the generated error string. If the supplied buffer is smaller than the generated error string, DACS_FAILURE will be returned.
buffer_len	buffer_len is a variable that indicates the maximum size of the err_buffer .
text	text is a pointer to a null-terminated character string that is placed into the buffer before the Data Access Control System error message. This string and the error message are separated by a colon and a blank space. If an empty character string is specified, only the Data Access Control System Library error message is placed into the err_buffer .
Dacs_Error	Dacs_Error points to a DACS_ERROR_TYPE data structure into which any returned errors are placed.

Table 11: DACS_perror() Functional Arguments

4.4.3 Return Values

DACS_perror() returns:

- **DACS_SUCCESS** if the function did not encounter a fatal error.
- **DACS_FAILURE** if a fatal unrecoverable error was encountered.

Appendix A Example Program

The following is some example code created using the DACSLock API:

```
/*
 * This source code is provided under the Apache 2.0 license and is provided
 * AS IS with no warranty or guarantee of fit for purpose. See the project's
 * LICENSE.md for details.
 * Copyright (C) 2019,2024 Refinitiv. All rights reserved.
 */

/*
 * This is the main file for the rsslAuthLock application. Its
 * purpose is to demonstrate the functionality of the DACS library.
 */

#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include "dacs_lib.h"

#define MAX_PRODUCT_CODES 256

typedef struct {
    char            _operator;
    unsigned short  pc_listLen;
    unsigned long   pc_list[MAX_PRODUCT_CODES];
} PC_DATA;

static void rsslAuthLock();

int main(int argc, char **argv)
{
    rsslAuthLock();
}

static void rsslAuthLock()
{
    PC_DATA pcData;
    PRODUCT_CODE_TYPE* pcTypePtr = (PRODUCT_CODE_TYPE *)&pcData;
    COMB_LOCK_TYPE combineLockList[5];
    COMB_LOCK_TYPE* combineLockListPtr = (COMB_LOCK_TYPE *)&combineLockList[0];
    unsigned char* lockPtr = NULL;
    unsigned char lockData[32];
    unsigned char* lock1Ptr = NULL;
    unsigned char lock1Data[32];
    unsigned char* lock2Ptr = NULL;
    unsigned char lock2Data[32];
}
```

```

unsigned char* combineLockPtr = NULL;
unsigned char* combineLock1Ptr = NULL;
unsigned char* combineLock2Ptr = NULL;
int lockLen = 0;
int lock1Len = 0;
int lock2Len = 0;
int combineLockLen = 0;
int combineLock1Len = 0;
int combineLock2Len = 0;
DACS_ERROR_TYPE dacsError;
unsigned char dacsErrorBuffer[128];

printf("\nCreate lock\n");
pcData._operator = OR_PRODUCT_CODES;
pcData.pc_listLen = 1;
pcData.pc_list[0] = 62;

if (DACS_GetLock(5000, pcTypePtr, &lockPtr, &lockLen, &dacsError) == DACS_FAILURE)
{
    if (DACS_perror(dacsErrorBuffer, sizeof(dacsErrorBuffer), (unsigned char *)"DACS_GetLock()
failed with error", &dacsError) == DACS_SUCCESS)
    {
        printf("%s\n", dacsErrorBuffer);
    }
    else
    {
        printf("DACS_GetLock() failed\n");
    }
    return;
}
printf("DACS_GetLock() - Success\n");

printf("\nCreate lock1\n");
pcData.pc_list[1] = 144;
pcData.pc_listLen += 1;
if (DACS_GetLock(5000, pcTypePtr, &lock1Ptr, &lock1Len, &dacsError) == DACS_FAILURE)
{
    if (DACS_perror(dacsErrorBuffer, sizeof(dacsErrorBuffer), (unsigned char *)"DACS_GetLock()
failed with error", &dacsError) == DACS_SUCCESS)
    {
        printf("%s\n", dacsErrorBuffer);
    }
    else
    {
        printf("DACS_GetLock() failed\n");
    }
    return;
}
printf("DACS_GetLock() - Success\n");

printf("\nCreate lock2\n");

```



```

pcData.pc_list[1] = 62;
pcData.pc_list[2] = 144;
pcData.pc_listLen += 1;
if (DACS_GetLock(5000, pcTypePtr, &lock2Ptr, &lock2Len, &dacsError) == DACS_FAILURE)
{
    if (DACS_perror(dacsErrorBuffer, sizeof(dacsErrorBuffer), (unsigned char *)"DACS_GetLock()
failed with error", &dacsError) == DACS_SUCCESS)
    {
        printf("%s\n", dacsErrorBuffer);
    }
    else
    {
        printf("DACS_GetLock() failed\n");
    }
    return;
}
printf("DACS_GetLock() - Success\n");

printf("\nChange Service id in lock1\n");
/* free existing lock1 */
free(lock1Ptr);
lock1Ptr = NULL;
pcData.pc_list[0] = 62;
pcData.pc_list[1] = 144;
pcData.pc_listLen = 2;
// Service id changed from 5000 to 30
if (DACS_GetLock(30, pcTypePtr, &lock1Ptr, &lock1Len, &dacsError) == DACS_FAILURE)
{
    if (DACS_perror(dacsErrorBuffer, sizeof(dacsErrorBuffer), (unsigned char *)"DACS_GetLock()
failed with error", &dacsError) == DACS_SUCCESS)
    {
        printf("%s\n", dacsErrorBuffer);
    }
    else
    {
        printf("DACS_GetLock() failed\n");
    }
    return;
}
printf("DACS_GetLock() - Success\n");

printf("\nCompare lock1 and lock2\n");
if (DACS_CmpLock(lock1Ptr, lock1Len, lock2Ptr, lock2Len, &dacsError) == DACS_DIFF) // identical
{
    printf("DACS_CmpLock() - Two locks are different\n");
}
else
{
    printf("DACS_CmpLock() - Two locks are identical\n");
}

```

```

printf("\nCombine lock, lock1, and lock2 into combineLock\n");
combineLockList[0].server_type = 0;
combineLockList[0].item_name = NULL;
combineLockList[0].lockLen = lockLen;
memcpy(lockData, lockPtr, lockLen);
combineLockList[0].access_lock = &lockData[0];
combineLockList[1].server_type = 0;
combineLockList[1].item_name = NULL;
combineLockList[1].lockLen = lock1Len;
memcpy(lock1Data, lock1Ptr, lock1Len);
combineLockList[1].access_lock = &lock1Data[0];
combineLockList[2].server_type = 0;
combineLockList[2].item_name = NULL;
combineLockList[2].lockLen = lock2Len;
memcpy(lock2Data, lock2Ptr, lock2Len);
combineLockList[2].access_lock = &lock2Data[0];
combineLockList[3].server_type = 0;
combineLockList[3].item_name = NULL;
combineLockList[3].access_lock = NULL;
combineLockList[3].lockLen = 0;
if (DACS_CsLock(0, (char *)"", &combineLockPtr, &combineLockLen, combineLockListPtr, &dacsError) ==
DACS_FAILURE)
{
    if (DACS_perror(dacsErrorBuffer, sizeof(dacsErrorBuffer), (unsigned char *)"DACS_CsLock() failed
with error", &dacsError) == DACS_SUCCESS)
    {
        printf("%s\n", dacsErrorBuffer);
    }
    else
    {
        printf("DACS_CsLock() failed\n");
    }
    return;
}
printf("DACS_CsLock() - Success\n");

/* free locks */
free(lockPtr);
free(lock1Ptr);
free(lock2Ptr);
free(combineLockPtr);
free(combineLock1Ptr);
free(combineLock2Ptr);
}

```

© 2016 - 2024 Refinitiv. All rights reserved.

Republication or redistribution of Refinitiv content, including by framing or similar means, is prohibited without the prior written consent of Refinitiv. 'Refinitiv' and the Refinitiv logo are registered trademarks and trademarks of Refinitiv.

Any third party names or marks are the trademarks or registered trademarks of the relevant third party.

Document ID: ETAC380REDAC.240

Date of issue: April 2024

