

[print](#) | [close](#)

## APIs by Example: A Validation List Entry's Life Cycle in CL Commands

[System iNetwork Programming Tips Newsletter](#)

[Carsten Flensburg](#)

Carsten Flensburg

Thu, 06/25/2009 (All day)

This week, I continue the coverage of validation lists and validation list entries, which I began in the June 11 issue of this newsletter. In that article, I discussed the basics of validation lists and the anatomy of validation list entries, and I also provided new validation list commands. I recommend reading the article, if you haven't already. A link is provided at the end of this article.

As I've demonstrated in my earlier contributions to this column, you'll often build the validation list APIs and functions directly into your applications. You can also find links to these articles at the end of this article. However, while developing and testing these applications, you'll often need to create, change, verify, and remove validation list entries to make sure that everything works the way you've designed and planned. This is where the Add Validation List Entry (ADDVLDLE), Verify Validation List Entry (VFYVLDLE), Change Validation List Entry (CHGVLDLE), and Remove Validation List Entry (RMVVLDLE) commands I present today can offer some assistance.

Each validation list entry command also constitutes an interface to, and example of, the equivalent validation list entry API and can be employed as a starting point for your own use of these APIs. I think it is fair to say that the validation list API documentation, and the data structures defined for the API input and output parameters, at times can be quite challenging. Hopefully, these examples will provide a shortcut, should you need to code these APIs yourself. Only the data structure enhancements added to RPG IV in recent releases have made it possible to define the data structures exactly as they were designed by IBM's API programmers.

To show you an example of this complexity and how it matches the RPG IV data structure facilities, I've included a walkthrough of one of the validation list entry API data structures from a previous article. Let's look at the Add Validation List Entry API parameter data structure that specifies the entry's attribute data, Qsy\_Attr\_Info\_T. Currently, this data structure primarily supports the ability to define whether a validation list entry's encrypted data is allowed to be stored in a decryptable form, a concept I explained last time.

By default, if you don't submit this data structure on the API call, this attribute is set to zero, which will prevent you from retrieving the encrypted data from the validation list. You're faced with the challenge of defining this data structure correctly and initializing this attribute to '1' in order to make the encrypted data retrievable. The data structure in question is defined by three subfields:

```
**- Validation list attribute data:
A-> D Qsy_Attr_Info_T...
    D                               Ds               Qualified
    D  Number_Attrs                 10i 0 Inz( 1 )
    D  Res_align                   12a
```

```

B->  D  Attr_Descr                                LikeDs(Qsy_Attr_Descr_T)
C->  D                                           Inz( *LikeDs )

```

The third subfield, `Attr_Descr`, is defined by the `LikeDs` keyword to have the same format as the template data structure called `Qsy_Attr_Descr_T` (B). The many subfield initializations in the template data structure are propagated to the `Attr_Descr` parameter with the `Inz( *LikeDs )` keyword (C). The following code snippet shows these initializations:

```

      D Qsy_Attr_Descr_T...
      D                               Ds                               Qualified
      D  Attr_Location            10i 0 Inz( QSY_IN_VLDL )
      D  Attr_Type                10i 0 Inz( QSY_SYSTEM_ATTR )
      D  Attr_Res                 8a   Inz( *Allx'00' )
F->  D  Attr_ID_p                 *
      D  Attr_Other_Descr...
      D                               32a   Inz( *Allx'00' )
      D  Attr_Data_Info...
      D                               96a
D->  D  Attr_VLDL                 LikeDs(Qsy_In_VLDL_T )
      D                               Overlay(Attr_Data_Info: 1)
E->  D                               Inz( *LikeDs )
      D  Attr_In_Other...
      D                               96a   Overlay(Attr_Data_Info:1)
      D                               64a   Overlay(Attr_In_Other:33)
      D                               Inz( *Allx'00' )
      D  Attr_Other_Data...
      D                               32a   Inz( *Allx'00' )

```

The `Qsy_Attr_Descr_T` data structure again contains an embedded data structure, `Attr_VLDL`. This data structure is in turn defined by the template data structure `Qsy_In_VLDL_T` (D). The `Attr_VLDL` data structure is initialized with the values from the template data structure using the `Inz( *LikeDs )` keyword (E). The following code is where those initializations take place:

```

      D Qsy_In_VLDL_T    Ds                               Qualified
      D  Attr_CCSD       10i 0 Inz( -1 )
G->  D  Attr_Len         10i 0 Inz( 1 )
      D  Attr_Res_1      8a   Inz( *Allx'00' )
H->  D  Attr_Value_p     *

      **-- Qsy_Attr_Descr_T structure constants:
      D QSY_IN_VLDL      c                               0
      D QSY_SYSTEM_ATTR...
      D                  c                               0
      **-- Qsy_In_VLDL_T structure parameter:
      D Qsy_Vfy_Find     s                               1a   Inz( '1' )

      Reset  Qsy_Entry_ID_Info_T;

```

In the code, the complex data structures' subfields are referred to by specifying their qualified name:

```

A->  Qsy_Attr_Info_T
B->  |                               Attr_Descr

```

```

F->      |                      |                      Attr_ID_p
      |                      |                      |
      Qsy_Attr_Info_T.Attr_Descr.Attr_ID_p = %Alloc( 15 );

A->      Qsy_Attr_Info_T
B->      |                      Attr_Descr
F->      |                      |                      Attr_ID_p
      |                      |                      |
      %Str( Qsy_Attr_Info_T.Attr_Descr.Attr_ID_p: 15 )
      = 'QsyEncryptData';

A->      Qsy_Attr_Info_T
B->      |                      Attr_Descr
D->      |                      |                      Attr_VLDL
G->      |                      |                      |                      Attr_Len
      |                      |                      |                      |
      Qsy_Attr_Info_T.Attr_Descr.Attr_VLDL.Attr_Len
      = %Size( Qsy_Vfy_Find );

A->      Qsy_Attr_Info_T
B->      |                      Attr_Descr
D->      |                      |                      Attr_VLDL
H->      |                      |                      |                      Attr_Value_p
      |                      |                      |                      |
      Qsy_Attr_Info_T.Attr_Descr.Attr_VLDL.Attr_Value_p
      = %Addr( Qsy_Vfy_Find );

```

As you can imagine, it can easily take an hour or two to deduce and program such a data structure. It's often after much trial and error that I get such complex data structures working. At first, I often have only a vague idea of how the pieces fit together. Therefore, whenever time permits, I use the source debugger to step through the code and verify each element and subfield of the data structures. Using the display variable function against the name of the main data structure causes the source debugger to map out all segments and qualifications of the data structure and its subfields. This is also a great help in the trial-and-error process.

For now, let's turn our attention to the four CL commands being discussed today, the first one being the Add Validation List Entry (ADDVLDLE) command. Here's the ADDVLDLE command prompt:

Add Validation List Entry (ADDVLDLE)			
Type choices, press Enter.			
Validation list . . . . .		Name	
Library . . . . .	*LIBL	Name, *LIBL,	
*CURLIB			
Entry ID:			

Entry ID . . . . .		
Coded character set identifier	*DFT	1-65534, *DFT, *HEX
Encryption data:		
Encryption data . . . . .		
Coded character set identifier	*DFT	1-65534, *DFT, *HEX
Entry data:		
Entry data . . . . .		
Coded character set identifier	*DFT	1-65534, *DFT, *HEX
Encryption data option . . . . .	*VFYONLY	*VFYONLY, *VFYFIND
Entry ID hexadecimal . . . . .		
Encrypted data hexadecimal . . . . .		
Entry data hexadecimal . . . . .		

The command and all its parameters are documented in detail in the accompanying help text panel group and also match the validation list entry parts explained earlier. *Entry ID* defines the value that identifies the individual validation list entry. *Encryption data* is where you'd store a password or other confidential data to be either verified or retrieved at a later point. *Entry data* provides an opportunity to include and store other related information with the validation list entry.

For all three parameters, you also specify the coded character set identifier (CCSID) for each value. Depending on the value specified, the command will convert the value accordingly before storing it in the validation list entry.

Note that both the encryption data and entry data parts are optional. It is possible to create a validation list entry without either of the two. You do so by specifying \*NONE for either of them when creating the validation list entry. Please see the help text for all details. The *Encryption data* option is where you specify whether the encryption data is stored in a one-way or two-way encryption format, as explained earlier. Note that by default, the stored encryption data is only verifiable, not retrievable.

Because these commands are intended to be used in development and testing scenarios, I've also included an option to specify the command's three main parameters in hexadecimal format. To ensure that no CCSID conversion issues are at play, these parameters allow you to specify the entry ID, encryption data, and entry data in hexadecimal notation, i.e. hex nibble values 0-9 and A-F. The help text explains this option in more detail. The CHGVLDLE and RMVVDLE commands both present a subset of the above interface. Further documentation is in the help text panel groups.

The Verify Validation List Entry (VFYVLDLE) command also displays some of the above parameters, as you'll see below:

```

Verify Validation List Entry (VFYVLDLE)

Type choices, press Enter.

Validation list . . . . . Name
Library . . . . . *LIBL Name, *LIBL,
*CURLIB
Entry ID:

Entry ID . . . . .

Coded character set identifier *DFT 1-65534, *DFT, *HEX

Encryption data:

Encryption data . . . . .

Coded character set identifier *DFT 1-65534, *DFT, *HEX

Entry ID hexadecimal . . . . .

Encrypted data hexadecimal . . .

```

The main difference is that the outcome of the validation list entry, in case of the verification process leading to a failure, is communicated in the form of the exception message CBX0201 being returned to the command caller. If run from a command line, this has no further implications. However, if you're running the command in a program, you'll want to monitor for the CBX0201 message in order to catch the event of verification failure. So far I've offered the CVTVLDL, DSPVLDLE, ADDVLDLE, VFYVLDLE, CHGVLDLE, and RMVLDLE CL commands. Next time, I'll complete the collection of validation list commands, so if this has caught your interest so far, remember to check out the next APIs by Example.

**This APIs by Example includes the following sources:**

```

CBX2051  -- RPGLE  -- Add Validation List Entry - CPP
CBX2051V -- RPGLE  -- Add Validation List Entry - VCP
CBX2051H -- PNLGRP -- Add Validation List Entry - Help
CBX2051X -- CMD    -- Add Validation List Entry

```

```

CBX2052  -- RPGLE  -- Verify Validation List Entry - CPP
CBX2052V -- RPGLE  -- Verify Validation List Entry - VCP
CBX2052H -- PNLGRP -- Verify Validation List Entry - Help
CBX2052X -- CMD    -- Verify Validation List Entry

CBX2053  -- RPGLE  -- Change Validation List Entry - CPP
CBX2053V -- RPGLE  -- Change Validation List Entry - VCP
CBX2053H -- PNLGRP -- Change Validation List Entry - Help
CBX2053X -- CMD    -- Change Validation List Entry

CBX2054  -- RPGLE  -- Remove Validation List Entry - CPP
CBX2054V -- RPGLE  -- Remove Validation List Entry - VCP
CBX2054H -- PNLGRP -- Remove Validation List Entry - Help
CBX2054X -- CMD    -- Remove Validation List Entry

CBX205   -- RPGLE  -- Validation List Entry Commands - Services
CBX205B  -- SRVSRC -- Validation List Entry Commands - Binder source

CBX205M  -- CLP    -- Validation List Entry Commands - Build commands

```

To create these Validation List Entry command objects, compile and run CBX205M, following the instructions in the source header. As always, you'll also find compilation instructions in the respective source headers.

### **[Retrieve the source code for this API example.](#)**

### **Previously published related articles:**

[APIs by Example: Have a Peek at Validation List Entries](#)

[APIs by Example: User Function Registration APIs, Part 1](#)

[APIs by Example: User Function Registration APIs, Part 2](#)

[APIs by Example: User Function Registration APIs, Part 3](#)

[APIs by Example: Validation List APIs](#)

[APIs by Example: Profile Authorization Management](#)

[APIs by Example: Cryptographic Services APIs, Part 3](#)

[APIs by Example: Cryptographic Services APIs, Part 7](#)

### **This article demonstrates the following Validation List APIs:**

[Add Validation List Entry \(QsyAddValidationLstEntry\) API](#)

[Verify Validation List Entry \(QsyVerifyValidationLstEntry\) API](#)

[Remove Validation List Entry \(QsyRemoveValidationLstEntry\) API](#)

[Find Validation List Entry \(QsyFindValidationLstEntry\) API](#)

[Find Validation List Entry Attributes \(QsyFindValidationLstEntryAttrs\) API](#)

[Validation List APIs](#)

[Digital Certificate Management API](#)

**[Retrieve the source code for this API example.](#)**

**Source URL:** <http://iprodeveloper.com/rpg-programming/apis-example-validation-list-entrys-life-cycle-cl-commands>