

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

APIs by Example: Copying System i Message Descriptions

[System iNetwork Programming Tips Newsletter](#)

[Carsten Flensburg](#)

Carsten Flensburg

Thu, 03/26/2009 (All day)

Message files and message descriptions offer a convenient method of storing messages and text. The IBM i OS itself makes extensive use of message descriptions for all sorts of user communication, program to program messages, display panel constants, command definition prompts, and error text to mention some of the most prominent and visual employments. I've previously published a couple of articles discussing and displaying how to use message descriptions as an extension to the system request menu as well as how to replace display file constants with message identifiers and texts. Look below for links to these articles.

Being such a versatile and widely used facility, message files and message descriptions also have a lot to offer in the domain of application development. In terms of for example user dialog, error handling, and multi-language support, message descriptions provide a flexible and comprehensible approach that can easily be incorporated and maintained by using the many message handling APIs and CL commands available. Having taken advantage of the message facilities included with the IBM i, I've often needed to copy a message description, but IBM hasn't provided a Copy Message Description (CPYMSGD) command, but this article does!

If you need to copy message descriptions from one message file to another, the Merge Message File (MRGMSGF) command helps you, but only if the message identifier won't change. Copying and renaming a message description is impossible. Because the Retrieve Message (QMHRTVM) API can return all message description attributes, I can write my own CPYMSGD command! More about the CPYMSGD command in a minute.

A message description has quite a few attributes. A quick count on the Add Message Description (ADDMSGD) command prompt finds 17 parameters in addition to the primary Message identifier (MSGID) and Message file (MSGF) parameters. Some of these are really only useful for messages sent by the IBM i, but others are valuable to programmers like me. I briefly mention some of the most useful ones here, but I urge you to look up the ADDMSGD command's help text for more details.

The *Message* (MSG) parameter holds the first-level message text that appears immediately on the screen, in the joblog or where else a message queue or program message queue is displayed. The text itself can have a maximum length of 132 bytes, but you can include substitution variables in the format &1, &2, &3, etc. Each substitution variable is replaced on message display or retrieval by the corresponding value defined by the *Message data fields formats* (FMT) list parameter and parsed from the Message data (MSGDTA) string submitted with the message when it is sent. This is very useful, as I show you later, if you want to tailor and adapt the final message text to reflect information that is variable by nature.

Also note that if you display a message in a message queue, the message text is retrieved dynamically. This implies that if a message description's message text is changed after the message is sent, you'll see the current text on your screen as opposed to the text in effect when the message was sent.

The *Second-level message text* (SECLVL) parameter contains the part of the message text shown if, for example, a message is prompted by using function key F1 to display the Additional Message Information panel or if your job's log setting has been configured to include second-level help text. The SECLVL text lets you specify up to 3000 bytes of additional information and help text to further explain and detail the first-level message text.

Substitution variables are also allowed and in effect in this part of the message, which further supports three message format control instructions (&N, &P, and &B) that let you control text wrapping and indentation when the second-level message text is displayed on screen. And now that we're talking about the second-level message text, this is also where you find an explanation of how to use the format control instructions mentioned: On the ADDMSGD or CHGMSGD command's SECLVL parameter press F1 to display the parameter's online help text and scroll down to get all the details.

The *Message data fields formats* (FMT) parameter mentioned earlier defines a consecutive number of substitution variables by data type and length. For each &n substitution variable defined in the MSG or SECLVL parameter, a corresponding FMT list element must be present; otherwise an error message is returned, and the ADDMSGD or CHGMSGD command fails. This behavior can be annoying if you've spent a lot of effort composing the text parts of the message description, because unless you specified the ADDMSGD or CHGMSGD command on a command line, you have to start all over again if you run into a mismatch between substitution variables and their FMT specified definitions.

There's also an impressive number of message attributes enabling you to control the interaction between a user and an inquiry type of message: The *Reply type* (TYPE), *Maximum reply length* (LEN), *Valid reply values* (VALUES), *Special reply values* (SPCVAL), *Range of reply values* (RANGE), *Relationship for valid replies* (REL) and the *Default reply value* (DFT) parameters let you compose a set of rules enforced by the OS when a user enters a value and tries to respond to an inquiry message. This ability can dramatically reduce the number of variations that you have to deal with in the program receiving the reply.

I leave the rest of the message description attribute to your own study. Apart from the online help text referred to earlier, you'll also find a lot of information covering this topic in the [CL manual](#). In the release 5.4 version of this manual, the section documenting the message concept begins on page 452.

As for examples of how to exploit message files and descriptions in an application development context apart from the examples published earlier and mentioned herein, there's of course the straightforward approach much similar to how the IBM i incorporates messaging in its components. The basic steps involve using the Send program message (SENDPGMMMSG) command or (QMHSNDPM) API to send predefined message descriptions contained in a message file. The send program message command or API allows you to configure a number of parameters to control the target message queue, message data, message type and severity, and so on. You'll find plenty of examples of this type of usage in practically all previously published APIs by Examples utilities.

Another approach involves the use of message files and descriptions as an application message and text repository. Again, you create a number of message files and message descriptions, but instead of sending the message directly, you use the Retrieve Message (QMHRTVM) API (or RTVMSG

command) to retrieve the message text, optionally specifying a string of message data to replace embedded substitution variables. Developing a naming scheme for your message descriptions, message files, and libraries lets you devise a simple yet powerful and transparent message and text store.

For the sake of demonstrating the basic idea, let's take an example of how to support a variety of application modules, country languages, and product brands. The design objective is to return a single message or text string based on a unique message ID in the range of 1 to 9999 and an input of three contextual parameters from the application:

- A three-character application ID defining the origin of the request.
- A numeric ISO country code defining the language in which to return the text.
- A three-character brand code allowing you to differentiate the dialogue based on product branding requirements.

A naming scheme supporting the above requirement could be established as follows:

- Message ID is composed by application ID and message ID in the format *AAAMMMM*
- Message file name is composed by company code and country code in the format *XXXXCCC*
- Library name is composed by company code and brand code in the format *XXXXBBB*

```
A=Application ID
M=Message ID
X=Company code
B=Brand code
```

So for company ACME, Inc.'s web application for the brand DeLuxe's German website, you would end up with the following naming scheme:

- Message IDs in the range WEB0001-WEB9999
- Message file name ACME28o
- Library name ACMEDLX

Adding new applications, countries, and brands in this scheme is structurally a fairly simple task (but due to involved translation efforts, potentially work intensive). To establish support for German and UK English websites as well as for message ID 201, the following commands would be executed:

1. CRTLIB ACMEDLX
2. CRTMSGF ACMEDLX/ACME28o
3. CRTMSGF ACMEDLX/ACME826
4. ADDMSGD MSGID(WEB0201) MSGF(ACMEDLX/ACME28o) MSG('User ID &1 ist unbekannt.') FMT((*CHAR 10))
5. ADDMSGD MSGID(WEB0201) MSGF(ACMEDLX/ACME826) MSG('User ID &1 is unknown.') FMT((*CHAR 10))

Retrieving the messages and texts can then be done by a single service program subprocedure defining an interface along the following lines:

```
**-- Retrieve message:
D RtvMsg          Pr          256a    Varying
D PxBrdId         3a          Value
D PxAppId         3a          Value
```

D	PxCtrCod	3a	Value	
D	PxMsgId	4a	Value	
D	PxMsgDta	128a	Varying	Const Options
(*NoPass)		

Retrieving the applicable message text would then be simply a matter of concatenating the relevant parameters and executing the API call:

```

**-- Local variables:
D MsgId          s          7a
D MsgFil         s          10a
D MsgLib         s          10a
D MsgDta         s          256a    Varying
**-- Local constants:
D RPL_SUB_VAL    c          '*YES'
D NOT_FMT_CTL    c          '*NO'
D COMP_ID        c          'ACME'
D NULL           c          ''

/Free

  MsgId  = PxAppId + PxMsgId;
  MsgFil = COMP_ID + PxCtrCod;
  MsgLib = COMP_ID + PxBrdId;

  If  %Parms >= 5;
    MsgDta = PxMsgDta;
  Else;
    MsgDta = NULL;
  EndIf;

  RtvMsgD( RTVM0100
    : %Size( RTVM0100 )
    : 'RTVM0100'
    : MsgId
    : MsgFil + MsgLib
    : MsgDta
    : %Len( MsgDta )
    : RPL_SUB_VAL
    : NOT_FMT_CTL
    : ERRRC0100
    );

  If  ERRRC0100.BytAvl > *Zero;
    Return  NULL;

  Else;
    Return  %Subst( RTVM0100.Msg: 1: RTVM0100.RtnMsgLen );
  EndIf;

/End-Free

```

And to retrieve the message text, the following lines of code in a program binding to the service program in question would do the trick:

```

/Free

If  VfyUsrId( CliRqs.UsrId ) = *Off;
    SvrRsp.MsgId  = '0201'
    SvrRsp.MsgDta = CliRqs.UsrId;
EndIf;

...

SvrRsp.MsgTxt = RtvMsg( CliRqs.BrdId
                        : CliRqs.AppId
                        : CliRqs.RqsSite
                        : SvrRsp.MsgId
                        : SvrRsp.MsgDta
                        );

/End-Free

```

Depending on the combination of the Brand ID, Application ID, and Requesting Site, the same message ID and message data can lead to differently worded messages in different languages with no further programming efforts. I hope you get the picture of both the idea behind the message text repository infrastructure and the relative simplicity of making tailored messages and texts available to your applications and modules in such a setup. The above scheme of course can and should be adapted to reflect the individual requirements applicable to each specific application or module.

Taking advantage of message descriptions and message files in the way described here will often bring you into a situation in which using an existing message description as a model for a new one will speed up the process significantly, especially if substitution variables are involved. That need turned my attention to another utilization of the QMHRTVM API and more specifically its return format RTVMO400, which makes available all information required to perform a copy operation of a message description.

The data structures embedded in the RTVMO400 format as well as the complex parameter lists of the ADDMSGD command that I eventually use to create the copied message description, however, turned my ambition of creating a Copy Message Description (CPYMSGD) command into a labor-intensive task, as you might agree if you [take a glance at the code accompanying this article](#). Given the usefulness of the CPYMSGD command and the time it can help my colleagues and myself save down the line while managing and adapting application message descriptions, this was time well spent.

Apart from mapping API output to command input, I had to deal with the QMHRTVM API's sensitivity to message file overrides. Calling the API in jobs in which one or more message file overrides are in effect due to previous executions of the Override Message File (OVRMSGF) would cause the API output to reflect the message file override, in case an override refers to the same message file as the API message file input parameter. Some other APIs have input parameters to define whether to ignore overrides, but alas the QMHRTVM API does not. So to ensure that the API is returning information applying to the specific message file requested, you have to perform an override to that message file immediately before the API call and delete the override again

immediately after the API call. Although it works, I would prefer an API parameter to achieve the same result.

The CPYMSGD command retrieves the message description attributes by means of a prompt override program (POP), which, based on the two key input parameters Message ID and the qualified Message file name, returns and formats all the necessary message information as a prompt string to the CPYMSGD command. This method, however, implies that you must prompt the command in order for the prompt override program to be called. Here's what the initial CPYMSGD command prompt looks like:

```

                                Copy Message Description (CPYMSGD)

Type choices, press Enter.

Message identifier . . . . . Name
Message file . . . . . Name
Library . . . . . *LIBL Name, *LIBL,
*CURLIB

```

Entering an existing message ID and message file and pressing Enter causes the command's prompt override program to fill in all other command parameters for you, as in the following example:

```

                                Copy Message Description (CPYMSGD)

Type choices, press Enter.

Message identifier . . . . . > CPF22A5 Name
Message file . . . . . > QCPFMSG Name
Library . . . . . > QSYS Name, *LIBL,
*CURLIB
To message identifier . . . . . *MSGID Name, *MSGID
To message file . . . . . *MSGF Name, *MSGF
Library . . . . . Name, *LIBL,
*CURLIB
First-level message text . . . . . 'Object &1 in &3 type *&2 not
secured by aut

```

horization list &4.'

Second-level message text . . . '&N Cause : The user specified

authorization list &4 to be revoked from object &1 in &3, type *&2.

The specified

object is not secured by authorization list &4. &N Recovery . . . :

Use the

display object authority (DSPOBJAUT) command to determine what authorization

list is securing the object, if any. Issue the RVKOBJAUT command again with the

authorization list that is securing the object to revoke the authorization

list's authority.'

...

Severity code 40 0-99

Message data fields formats:

Data type *CHAR *NONE, *QTDCHAR,
*CHAR...

Length 10 Number, *VARY

*VARY bytes or dec pos 0 Number

Data type *CHAR *QTDCHAR, *CHAR,
*HEX...

Length 7 Number, *VARY

*VARY bytes or dec pos 0 Number

Data type *CHAR *QTDCHAR, *CHAR,
*HEX...

Length 10 Number, *VARY

*VARY bytes or dec pos 0 Number

Data type *CHAR *QTDCHAR, *CHAR,
*HEX...

Length 10 Number, *VARY

*VARY bytes or dec pos 0 Number

+ for more values

Reply type *NONE *CHAR, *DEC,
*ALPHA, *NAME...

Maximum reply length:

Length	*NONE	Number, *TYPE,
*NONE		
Decimal positions		Number

Valid reply values	*NONE	
------------------------------	-------	--

+ for more values

Special reply values:

Original from-value	*NONE	
-------------------------------	-------	--

Replacement to-value		
--------------------------------	--	--

+ for more values

Range of reply values:

Lower value	*NONE	
-----------------------	-------	--

Upper value		
-----------------------	--	--

Relationship for valid replies:

Relational operator	*NONE	*NONE, *EQ, *LE,
*GE, *GT...		

Value		
-----------------	--	--

Default reply value	*NONE	
-------------------------------	-------	--

Additional Parameters

Default program to call	*NONE	Name, *NONE
-----------------------------------	-------	-------------

Library		Name, *LIBL,
*CURLIB		

Data to be dumped	*NONE	1-99, *NONE, *JOB,
*JOBINT...		

+ for more values

Level of message:

Creation date	*CURRENT	Date, *CURRENT
-------------------------	----------	----------------

Level number	1	1-99
------------------------	---	------

Alert options:		
Alert type	*NO	*IMMED, *DEFER,
*UNATTEND...		
Resource name variable	*NONE	1-99, *NONE
Log problem	*NO	*NO, *YES
Coded character set ID	*JOB	*JOB, *HEX, 37,
256, 273...		

Specify either a To message identifier, a To message file, or both, then perform the desired changes to the command parameters and press Enter. The specified message description will be created in the specified message file. As always, you can also look up the command's online help text for more details. As for the CPYMSGD command objects involved in the utility, here's a brief walk through to give you an idea of how the command works:

- The CPYMSGD command definition. This is pretty much a copy of the ADDMSGD command with the addition of the TOMSGID and TOMSGF parameters.
- The CBX201O command prompt override program. The program is called by the command prompt facility once the CPYMSGD command's two key parameters have been entered. The program calls the QMHRTVM API to retrieve the specified message description's attributes and subsequently formats and returns a command prompt string specifying all the CPYMSGD command's remaining input parameters.
- The CBX201C command choice program uses Retrieve CCSIDs (QLGRTVCD) API to produce a list of all available and supported CCSID values for the CPYMSGD command's CCSID parameter. This is a function similar to the one that the ADDMSGD command provides. The ADDMSGD command's choice program is, however, sensitive to the name of the command calling it as this parameter controls the list of available CCSID values returned, so I had to write my own.
- The CBX201V command validity checker validates the primary CPYMSGD parameters among other things to ensure that the specified message identifier and message file actually do exist.
- The CBX201H help text panel group describes the command and all its parameters. The main part of the keyword help text is simply imported from the ADDMSGD command's help text panel group. Using this approach can be quite a time saver and ensures that both accurate and detailed information is provided.
- The CBX201 command processing program. This program retrieves the final CPYMSGD command parameters and formats an ADDMSGD command string defining all returned parameters. The command string is eventually processed by the Process Command (QCPCMD) API and possible errors are returned immediately to the caller by means of another message handling API, the Move Program Message (QMHMOVPM) API.

I've done my best to test the CPYMSGD command thoroughly, but given the number of parameters and possible combinations hereof, I cannot completely ensure that no unforeseen issues surface when you put the command to work. So please give the CPYMSGD command a test run before putting it into production. And be sure to let me know if you run into any issues using it.

This APIs by Example includes the following sources:

```

CBX201  -- RPGLE  -- Copy Message Description - CPP
CBX201C -- RPGLE  -- Copy Message Description - Choice program
CBX201H -- PNLGRP -- Copy Message Description - Help
CBX201O -- RPGLE  -- Copy Message Description - POP
CBX201V -- RPGLE  -- Copy Message Description - VCP
CBX201X -- CMD    -- Copy Message Description

CBX201M -- CLP    -- Copy Message Description - Build command

```

To create all these objects, compile and run CBX201M, following the instructions in the source header. As always, you'll find compilation instructions in the respective source headers.

IBM documentation:

[CL message concept](#)

[Message ID overview](#)

Previously published related articles:

[APIs by Example: Message Handling \(QMHRVTM/QMHMOVPM/QMHRVCVPM\)
- System request menu enhancement](#)

[APIs by Example: User Index APIs, Part One: Create User Index \(CRTUSRIDX\) command](#)

[APIs by Example: User Index APIs, Part Two: Convert Display File Constants \(CVTDSPFCNS\)
command](#)

This article demonstrates the following Message Handling APIs:

[Retrieve Message \(QMHRVTM\)](#)

[Move Program Message \(QMHMOVPM\) API](#)

[You can retrieve the source code for this API example from our website.](#)

Source URL: <http://iprodeveloper.com/rpg-programming/apis-example-copying-system-i-message-descriptions>