

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

## APIs by Example: Tricky Retrieve APIs and How to Process the Receiver Variable

[System iNetwork Programming Tips Newsletter](#)

[Carsten Flensburg](#)

Carsten Flensburg

Thu, 08/26/2010 (All day)

Most retrieve APIs are simple to work with. The data returned is defined by a data structure, and the information provided is in character or integer format. While often more than one alternative or accumulating return format is available, each is simple in structure and straightforward to process. As an example of an API providing alternative return formats, see the Retrieve Job Information (QUSRJOBI) API, which offers 12 different return formats. As an example of accumulating return formats, see the Retrieve Member Description (QUSRMBRD) API, which offers three return formats, the second including the first format, and the third including the second format.

Today, I discuss a couple of return formats that are a bit more challenging to handle. To help me do that, I've created a Display System Configuration (DSPSYSCFG) command involving the use of many different retrieve APIs—some of which demonstrate the simple approach of returning data, others a variety of the more challenging methods. This article's focus is on the latter retrieve API type and also includes a couple of MI built-ins. In MI built-in terminology, retrieve functionality is referred to as materialize, and sometimes, when no adequate retrieve API is available, there's a materialize MI built-in that will fit the bill nicely. More about that in a minute.

The Retrieve System Values (QWCRSVAL) and Retrieve Network Attributes (QWCRNETA) APIs resemble each other in the way that the return data is requested and how it is returned. Both API interfaces comply with the documentation below applying to the QWCRSVAL API:

Required Parameter Group:

1	Receiver variable	Output	Char(*)
2	Length of receiver variable	Input	Binary(4)
3	Number of system values to retrieve	Input	Binary(4)
4	System value names	Input	Array(*) of Char(10)
5	Error code	I/O	Char(*)

Parameter 1 defines the storage available to the API to return the requested information. Parameter 3 specifies the number of system values for which to return data and providing the count of the system value names specified in parameter 4, a simply array of 10-byte character fields, one element for each specified system value name. Parameter 5 is the standard API error code data structure. If you are somewhat familiar with APIs in general, the above parameter list should not cause any headache—at least not at this point.

Calculating the required length of the Receiver variable to be specified as the API's second parameter is, however, a bit more involved—here are the words of the manual on that subject—in my formatting for ease of reading and comprehension:

- To determine the length of the receiver variable, the following calculation should be done.
- For each system value to be returned, get the length of the data returned for the system value and add 24.
- After adding the lengths for each system value, add 4. This calculation takes into account the data alignment that needs to be done; therefore, this value is a worst-case estimate.
- If the calculated length is less than what is needed to return all the system value information, then the value of the Number of system values returned field will match the actual number of system values returned.
- The system value information for the system values that won't fit will not be returned. For example, if a request is made to return information about 1 system value, and that information will not fit, then the *Number of system values returned* field will be 0, and there will be no information returned in the *System value information table* field.

Now let us continue to the next part, the actual format of the receiver variable. Here's the header section of that:

Offset			Field
Dec	Hex	Type	
0	0	BINARY(4)	Number of system values returned
4	4	ARRAY(*) of BINARY(4)	Offset to system value
information table			
*	*	CHAR(*)	System value information table.
This field			
returned.			is repeated for each system value

At the offset(s) specified by the array of four-byte integers, you find the following data structure, which is repeated for each returned value:

Offset			Field
Dec	Hex	Type	
0	0	CHAR(10)	System value
10	A	CHAR(1)	Type of data
11	B	CHAR(1)	Information status
12	C	BINARY(4)	Length of data
16	10	CHAR(*)	Data

Let's put the above specifications to work and start with the calculation of the required receiver variable length. Even though I plan to retrieve more than one system value, for simplicity I decided to retrieve them one at a time. This gives me the following prototype for the GetSysVal() procedure encapsulating the QWCRSVAL API call:

```

**-- Get system value:
D GetSysVal          Pr          4096a    Varying
D   PxSysVal         10a        Const

```

You specify a system value name, for example QSRLNBR, as input and get the retrieved value back in the procedure's return value. The system value can have a maximum length of 4096 bytes, which should safely cover all currently possible system values. Numeric system values are returned as edited character strings. Below I've defined the QWCRSVAL API input parameters 1, 2, 3, and 4 the RtnVar data structure and the three ApiPrm data structure subfields, respectively:

```

D SysVal             s          4096a    Varying
**
D ApiPrm             Ds          Qualified
D   RtnVarLen        10i 0
D   SysValNbr        10i 0 Inz( %Elem( ApiPrm.SysVal ))
D   SysVal           10a      Dim( 1 )
**
D RtnVar             Ds          Qualified
D   RtnVarNbr        10i 0
D   RtnVarOfs        10i 0 Dim( %Elem( ApiPrm.SysVal ))
D   RtnVarDta        4096a
**
D SysValInf          Ds          Qualified Based( pSysVal )
D   SysValKwd        10a
D   DtaTyp           1a
D   InfSts           1a
D   DtaLen           10i 0
D   Dta              4096a
D   DtaInt           10i 0 Overlay( Dta )

```

As mentioned above, I retrieve only one value at a time, but in order to be able to easily adapt my code in case I at some point decide to retrieve more values in one call, I use the element count of the system value array together with the size of the return variable in my calculation, which is performed following the instructions outlined above:

```

/Free

  ApiPrm.RtnVarLen = %Elem( ApiPrm.SysVal ) * 24 + %Size( SysVal ) +
4;
  ApiPrm.SysVal(1) = PxSysVal;

  RtvSysVal( RtnVar
             : ApiPrm.RtnVarLen
             : ApiPrm.SysValNbr
             : ApiPrm.SysVal
             : ERRC0100
             );

/End-Free

```

Following the API call, the RtnVar data structure defines the number of system values returned as well as the offset from the beginning of the RtnVar data structure to each of the system value

information data structures returned. The system value information data structure is defined by the above data structure SysValInf in turn based on the pSysVal pointer. So for each system value returned, the SysValInf data structure is mapped to the address of the corresponding location in the receiver variable:

```

/Free

  For  Idx = 1  to RtnVar.RtnVarNbr;

    pSysVal = %Addr( RtnVar ) + RtnVar.RtnVarOfs(Idx);

    If  SysValInf.SysValKwd = PxSysVal;

      Select;
      When  SysValInf.DtaTyp = 'C';
        SysVal = %Subst( SysValInf.Dta: 1: SysValInf.DtaLen );

      When  SysValInf.DtaTyp = 'B';
        SysVal = %Char( SysValInf.DtaInt );

      Other;
        SysVal = NULL;
      EndSl;
    EndIf;
  EndFor;

/End-Free

```

Based on the data type, the actual system value is then copied to the SysVal return value variable. This is, of course, the simple way of calling APIs like QWCRSVAL and QWCRNETA. Although this is how I chose to use these APIs in this example, due to the fact that much of the complexity connected to calling these APIs is related to their capacity to return multiple return values, I think it makes sense to also demonstrate how to go about that. Let's say I want to retrieve three system values at a time. Here's the prototype for the corresponding GetSysVal() procedure:

```

**-- Get system value:
D GetSysVal          Pr          10i 0
D  PxSysValKwd       10a      Const  Dim( 3 )
D  PxSysVal          4096a          Dim( 3 )  Varying

```

And here's the adapted version of the GetSysVal() procedure:

```

D RtnVar              Ds              Qualified  Based( pRtnVar )

D  RtnVarNbr          10i 0
D  RtnVarOfs          10i 0 Dim( %Elem( ApiPrm.SysVal ) )
D  RtnVarDta          4096a

/Free

  ApiPrm.RtnVarLen = %Elem( ApiPrm.SysVal ) * 24 +

```

```

                                %Size( PxSysVal: *All ) + 4;

pRtnVar = %Alloc( ApiPrm.RtnVarLen );

ApiPrm.SysVal = PxSysValKwd;

RtvSysVal( RtnVar
           : ApiPrm.RtnVarLen
           : ApiPrm.SysValNbr
           : ApiPrm.SysVal
           : ERR0100
           );

If  ERR0100.BytAvl = *Zero;

  For  Idx = 1  to RtnVar.RtnVarNbr;

    pSysVal = pRtnVar + RtnVar.RtnVarOfs(Idx);

    Select;
    When  SysValInf.DtaTyp = 'C';
      SysVal = %Subst( SysValInf.Dta: 1: SysValInf.DtaLen );

    When  SysValInf.DtaTyp = 'B';
      SysVal = %Char( SysValInf.DtaInt );

    Other;
      SysVal = NULL;
    EndSl;

    PxSysVal( %Lookup( SysValInf.SysValKwd
                      : PxSysValKwd
                      )) = SysVal;

  EndFor;
EndIf;

DeAlloc(n)  pRtnVar;

If  ERR0100.BytAvl = *Zero;
  Return  *Zero;
Else;
  Return  -1;
EndIf;

/End-Free

```

Since I'm working with a multiple of system values, I'm now allocating the calculated storage to ensure that the receiver variable will always reflect the actual requirement. This will also make it easier and less error prone to adapt the GetSysVal() procedure in the future, if required. Apart from that, the process is pretty much the same as outlined for the simple version. I hope you get the picture. Along these lines, you could take it one step further and make the GetSysVal() procedure

accept and process an arbitrary number of system values, but I'll leave that as an exercise for you. Should you take up the challenge and need any kind of assistance, please let me know.

Another challenge you can be presented with comes from return data following other conventions in terms of data type or character set than usually applying on the IBM i in general, and within RPG development in particular. In the DSPSYSCFG CPP, I employ the Retrieve Partition Information (dlpar\_get\_info) API in order to retrieve the partition name of the current partition. This piece of information is available in format 1 of the two return formats available with the aforementioned API, and here's how the dlpar\_get\_info API documentation describes the *Partition name* subfield of the format 1 data structure:

```
Partition name is the name that has been assigned to this partition.
This
field is a null-terminated UTF-8 character string.
```

Null-terminated character strings and the UTF-8 character set require special attention to process correctly. Null termination is taken care of by the %STR (Get Null Terminated String) ILE/RPG built-in function, which converts the null-terminated string into a regular character string containing the value found up to but not including the null terminator. Converting the character value from UTF-8 to your job's current Coded Character Set Identifier (CCSID) involves a bit more work.

On the i system, UTF-8 is defined by CCSID 1208. The recommended approach as far as converting between CCSIDs on the i is concerned, is using the iconv APIs. At the end of this article, I've included a link to an article explaining the iconv() (Code Conversion) API in great detail. If the conversion task at hand is as limited—as in this case, in which only one character value needs to be converted—the Convert a Graphic Character String (QTQCVRT) API provides the exact same conversion facilities as the iconv APIs because it uses these APIs under the covers. And it is a little simpler to code, since it requires only one API call as opposed to conversion session oriented iconv APIs' minimum of three API calls. So for this task, I go with the QTQCVRT API.

If you want to learn more about the iconv() APIs, however, I recommend Scott Klement's article covering the topic and to which I provide a link at the end of this article. While the QTQCVRT API still requires some coding efforts to satisfy the 12 parameters required, for ease of use I've created a procedure that wraps it up and only requires the three crucial input parameters and one return value:

```
**-- Convert string by CCSID:
D CvtStrCcsId      Pr          1024a   Varying
D  PxCcsId         10i  0  Const
D  PxCvtStr        1024a   Const
D  PxCvtStrLen     10i  0  Const
```

The CvtStrCcsID() procedure is designed to handle smaller strings of a size of up to 512 double-byte characters and 1024 single-byte characters and implicitly converts from the specified CCSID to the CCSID of the current job. Since the dlpar\_get\_info API returns the partition name in a null-terminated UTF-8 string, all that remains now to convert this value into the job CCSID is to strip the null termination from the string prior to conversion. This is handled in the CvtStrVal() procedure, which takes the address of a null-terminated string value and converts the string from the CCSID specified as the second parameter:

```

**-- Convert string value:
D CvtStrVal      Pr      1024a   Varying
D  PxStrVar      *      Value
D  PxCcsId      10i 0 Value

```

So to take the UTF-8 null-terminated string returned by the `dlpar_get_info` API and convert it to the job CCSID now becomes the simple task of running the `CvtStrVal` procedure as it is done in the `GetLparName()` procedure from which I've picked the following code snippet:

```

/Free

  RtvPtnInf( PtnInf01: PTN_STC_INF: %Size( PtnInf01 ));

  Return  CvtStrVal( %Addr( PtnInf01.PtnNam ): 1208 );

/End-Free

```

As discussed many times earlier, apart from code readability, it also makes good sense to wrap up the string and conversion functions as demonstrated above because it allows you to encapsulate the procedures in a service program and easily reuse the code in case you for some reason want to inline the functions.

Finally, I'm going to briefly discuss the wealth of system information and functions accessible through the MI built-ins. To see the complete list of available MI built-ins, please follow the link at the end of this article. In today's API by Example, I use the Materialize Machine Attributes (MATMATR) and the Materialize Resource Management Data (MATRMD) MI built-ins. MATMATR among many other things returns partition information, and MATRMD is capable of delivering system processor and DB capability thresholds and limits as well as many other system resource related data.

MI built-ins are basically MI instructions made available to the IBM i ILE compilers through a bound program access interface and are documented in the MI instructions section of the API manual. To find out if an MI instruction has an equivalent ILE built-in version, you look up the MI instruction in the manual and check if there's a Bound program access box describing the built-ins interface at the beginning of the section documenting the MI instruction in question.

As an example of how the presence of an MI built-in is verified and documented, follow the links provided below to the MATMATR and MATRMD MI built-ins and note the section at the beginning of the documentation. Here's what is specified for the MATMATR MI instruction:

#### **Bound program access**

```

Built-in number for MATMATR1 is 92. MATMATR1 ( materialization :
address
  machine_attributes : address (of just a selector value) )

```

The MI built-in interface is further explained in the general section of the MI instruction documentation. Note that MI built-in names are typically preceded by an underscore, below the resulting prototype for MATMATR1:

```

**-- Materialize machine attributes:
D MatMatr          Pr          ExtProc( '_MATMATR1' )
D  Atr              32767a      Options( *VarSize )
D  Opt              2a         Const

```

The MATMATR MI instruction supports a large number of different return formats. You specify which of the formats to return by specifying the appropriate selection value as the instruction's second parameter, also referred to as operand in the MI terminology, and the adequately formatted data structure as the first parameter. Here's the data structure and the named constant used as input to the MATMATR1 MI built-in call, followed by the call itself:

```

**-- Constants:
D MMTR_LPAR_INFO  c          x'01E0'

**-- Partition information:
D MMTR_01E0_T      Ds          Qualified
D  BytPrv          10i 0  Inz( %Size( MMTR_01E0_T ))
D  BytAvl          10i 0
D  CurNbrPtn       3u 0
D  CurPtnId        3u 0
D  PriPtnId        3u 0
D  SrvPtnId        3u 0
D  FmwLvl          3u 0
D                  3a
D  LglSrlNbr       10a
D  MinPctInt       5u 0  Overlay( MMTR_01E0_T: 87 )
D  MaxPctInt       5u 0  Overlay( MMTR_01E0_T: 89 )
D  CurPctInt       5u 0  Overlay( MMTR_01E0_T: 91 )
D  NbrPhyPrc       5u 0  Overlay( MMTR_01E0_T: 93 )
D                  2a    Overlay( MMTR_01E0_T: 95 )

/Free

  MatMatr( MMTR_01E0_T: MMTR_LPAR_INFO );

  DtlRcd.CurNbrPtn  = MMTR_01E0_T.CurNbrPtn;
  DtlRcd.FmwLvl     = MMTR_01E0_T.FmwLvl;
  DtlRcd.LglSrlNbr  = MMTR_01E0_T.LglSrlNbr;

/End-Free

```

In addition to the API manual, it is often helpful to consult the QSYSINC library's C library MIH file's include members to verify and troubleshoot the MI built-in interfaces. While written in C, they will often give you an idea about the context, even if you're not that robust in the C language. Each MI built-in has a member in the MIH file defining the structures and constants employed by that built-in, and often also includes interesting comments and explanations relating to the use and history of the built-in. This is also where you can verify the exact name of the MI built-in. The *ILE C/C++ MI Library Reference* is a very useful resource too. I've provided a link to a PDF version of this manual at the end of this article.



To sum it all up, you can see examples of the above programming techniques in the DSPSYSCFG CPP, and I suggest you run the code in a source debugger to see how the pieces fit together. As for the DSPSYSCFG command itself, here's what the command prompt looks like:

```

                                Display System Configuration (DSPSYSCFG)

Type choices, press Enter.

Reset statistics . . . . . *NO          *NO, *YES

Output . . . . . *          *, *PRINT

```

Specify whether the system status statistics and elapsed time are reset to zero prior to retrieval of the system configuration information and also whether the command output should go to a display panel or a printed list. Here's an example of what the command would look like if you decided to display the system configuration information without resetting the system status statistics and elapsed time:

```

DSPSYSCFG RESET(*NO)
          OUTPUT(*)

```

The resulting display panel's first page would have the following appearance:

```

                                Display System Configuration

WYNDHAMW                                                                22-08-10

12:25:31
System name . . . . . : WYNDHAMW

Serial number . . . . . : 4321CBA

Type and model . . . . . : 9406-525

Processor feature . . . . . : 7792

Processor group . . . . . : P10

Partition name . . . . . : 43-21CBA

Partition ID . . . . . : 1

Logical serial number . . . . . : 4321CBA1

```

```

Processor share attribute . . . : *DEDICATED

Number of partitions . . . . . : 1

Firmware level . . . . . : 16


OS release . . . . . : V5R4M0

CUM package level and status . : 10117 Installed

System state . . . . . : *AVAILABLE

TCP status . . . . . : *ACTIVE


More...
F3=Exit    F5=Refresh    F12=Cancel    F19=Display partition
information
F20=Work with PTF groups    F21=Display software resources
F24=More keys

```

In addition to the system configuration information displayed, there are shortcuts in the form of function keys F19, F20, and F21 providing access to the Display Partition Information (DSPPTNINF), Work with PTF Groups (WRKPTFGRP), and Display Software Resources (DSPSFWRSC) commands, respectively. If it turns out that your system knows nothing about the DSPPTNINF command, don't worry; it'll be part of an upcoming article in the APIs by Example series. To see the DSPSYSCFG command's second page of configuration information, press the Page Down button:

```

                                Display System Configuration

WYNDHAMW                                                                22-08-10

12:25:31
Main storage size . . . . . : 15975968

Total aux storage size . . . . : 986031

System ASP size . . . . . : 986031

System ASP used . . . . . : 84,9857

System ASP threshold . . . . . : 90,0


Number of processors . . . . . : 2

```

```

Processor interactive threshold:    100,0

Processor interactive limit   . :   100,0

CPU percent used . . . . . :   22,2

DB capability threshold . . . :   100,0

DB capability limit . . . . . :   100,0

DB capability used . . . . . :   14,8


IPL date and time . . . . . :   18-08-2010   05:11:25

IPL type . . . . . :   B

Key lock position . . . . . :   Normal

```

```

      Bottom
F3=Exit   F5=Refresh       F12=Cancel   F19=Display partition
information
      F20=Work with PTF groups   F21=Display software resources
      F24=More keys

```

The display panel and all fields shown are explained in the cursor-sensitive help text associated with the display. Point the cursor to the area or field of interest and press F1 to access the help text provided.

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

```

CBX218  -- RPGLE  -- Display System Configuration - CPP
CBX218E -- RPGLE  -- Display System Configuration - UIM General Exit
CBX218H -- PNLGRP -- Display System Configuration - Help
CBX218P -- PNLGRP -- Display System Configuration - Panel Group
CBX218X -- CMD    -- Display System Configuration

CBX218M -- CLP    -- Display System Configuration - Build command

```

To create all these objects, compile and run the CBX218M program, following the instructions in the source header. You'll also find compilation instructions in the respective source headers.

### **Related article:**

[Converting Data Between CCSIDs](#) (June 2006, article ID 52786)

### **This article demonstrates the following APIs and MI Built-ins:**

[Retrieve System Values \(QWCRSVAL\) API](#)

[Retrieve Network Attributes \(QWCRNETA\) API](#)

[Retrieve System Status \(QWCRSSTS\) API](#)

[Retrieve Product Information \(QSZRTVPR\) API](#)

[List PTF Groups \(QpzListPtfGroups\) API](#)

[Retrieve TCP/IP Attributes \(QtocRtvTCPA\) API](#)

[Retrieve Partition Information \(dlpar\\_get\\_info\) API](#)

[Materialize Machine Attributes \(MATMATR\) MI Built-in](#)

[Materialize Resource Management Data \(MATRMD\) MI Built-in](#)

[i5/OS Machine Interface](#)

[ILE C/C++ MI Library Reference \(PDF\)](#)

[Convert a Graphic Character String \(QTQCVRT\) API](#)

[iconv\(\) Code Conversion API](#)

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

**Source URL:** <http://iprodeveloper.com/rpg-programming/apis-example-tricky-retrieve-apis-and-how-process-receiver-variable>