Getting Started With
APIs from RPG

Presented by
Scott Klement
http://www.scottklement.com

© 2006-2016, Scott Klement

A programmer's wife tells him: "Run to the store and pick up a loaf of bread; If they have eggs, get a dozen." The programmer comes home with 12 loaves of bread.

What's an API?

API = Application Programming Interface

• An Interface
  APIs represent a way for one application to interface with another one. For example, Order Entry software might need to interface with Shipping software to determine a shipping charge.

• Program or Procedure Calls
  Usually APIs are implemented as programs or subprocedures that you call and pass parameters to.

• Program to Program (or Procedure to Procedure)
  APIs are designed to be used by programs. They're not (usually) intended to be called from the command line, menu, etc. Instead, they're called from a program. They don't take their input from a keyboard, but instead from a parameter. They write their output to a parameter, and not to a screen or paper. They are programs intended to be called by programs.

• Who writes APIs?
  Anyone can write an API. In fact, you've probably already written some.
The IBM i APIs

IBM provides over 3200 different programs and procedures that you can call to interface with the various functions of the operating system!

This presentation focuses on how to get started using the IBM i APIs from an RPG IV (ILE RPG) program.

We’ll start by examining how IBM’s documentation is laid out, and discuss how to find the API you’re looking for, as well as which parameters it needs.

Methods for Finding APIs

There are two different scenarios where you might be looking for information about APIs:

- When you know the name of the API, but you don’t know what it does (usually when you’re trying to understand someone else’s code)
- When you know what you want to do, but you don’t know which API does the job.

IBM provides 3 ways of finding APIs:

- APIs by Category (When you don’t know the API name.)
- API finder (When you do know the API name or title.)
- Alphabetical Listing of APIs (I’ve never found a use for this.)
For example...

Let’s say you’re reading a program, and you see code like the following:

```plaintext
CHGVAR VAR(%BIN(&RCVVARLEN)) VALUE(1000)

CALL PGM(QDCRDEVD) PARM( &RCVVAR +
 &RCVVARNLEN +
 'DEVD0600' +
 &DEV +
 &ERRCODE )

CHGVAR VAR(&ADDR) VALUE(%SST(&RCVVAR 878 15))
```

In this case, you may not be sure what QDCRDEVD does, but you know it’s name. In that case, you want to be able to type the name and get information about the API.

To do that, you’ll use the API Finder.
The API finder is for searching for an API. It’s the “Google” for IBM i API information.
Found it… Now What?

Either the API finder has found a link to the API you were looking for, or you’ve found it by browsing the categories.

The next step is to click that link and read the documentation for the API itself.

This information is divided into sections that will be explained in upcoming slides:
- Parameter Summary Area
- API Description, Locks & Authority Info
- Detailed Information about parameters
- Error Information

API Parameter Summary

At the top of each API’s page

Send Program Message (QMHSEDPM) API

<table>
<thead>
<tr>
<th>Required Parameters Group:</th>
<th>How the API uses the parm:</th>
<th>Data type of the parm:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  Message identifier</td>
<td>Input</td>
<td>Char(7)</td>
</tr>
<tr>
<td>2  Qualified message file name</td>
<td>Input</td>
<td>Char(20)</td>
</tr>
<tr>
<td>3  Message data or immediate text</td>
<td>Input</td>
<td>Char(*)</td>
</tr>
<tr>
<td>4  Length of message data or immediate text</td>
<td>Input</td>
<td>Binary(4)</td>
</tr>
<tr>
<td>5  Message type</td>
<td>Input</td>
<td>Char(10)</td>
</tr>
<tr>
<td>6  Call stack entry</td>
<td>Input</td>
<td>Char(*) or Pointer</td>
</tr>
<tr>
<td>7  Call stack counter</td>
<td>Input</td>
<td>Binary(4)</td>
</tr>
<tr>
<td>8  Message key</td>
<td>Output</td>
<td>Char(4)</td>
</tr>
<tr>
<td>9  Error code</td>
<td>I/O</td>
<td>Char(*)</td>
</tr>
</tbody>
</table>

Optional Parameter Group 1:

| 10  Length of call stack entry | Input                  | Binary(4)               |
| 11  Call stack entry qualification | Input             | Char(20)               |
| 12  Display program messages screen wait time | Input      | Binary(4)               |

Optional Parameter Group 2:

| 13  Call stack entry data type | Input                  | Char(10)               |
| 14  Coded character set identifier | Input             | Binary(4)               |

QMHSNDPM is the name of the program to call.

These Parameters are Required, (must always be passed.)

These aren’t required, but if you pass one, you have to pass them all.

Same here, PLUS in order to pass group 2, you must also pass group 1.
QMHSNDPM Example (1 of 2)

QMHSNDPM( 'CPF9897': 'QCPFMSG   *LIBL': Msg:
: %len( %trimr(Msg) )
: '*DIAG': '*': 0: MsgKey: ErrorCode );

*INLR = *ON;
/end-free

Use CONST for "Input" parameters.

BINARY(4) is always "10I 0" in RPG IV.

More about data types later.

QMHSNDPM Example (2 of 2)

Display All Messages

Job . . . : User . . . : KLEMS01 Number . . . : 229997

> call senddiag
  This is a test. Don't read this.
>> dspjoblog

Press Enter to continue.

F3-Exit  F5-Refresh  F12-Cancel  F17-Top  F18-Bottom
Data Types

The data types that are listed for each API are usually pretty self explanatory.

Examples:
- CHAR(20) = character field, 20 long (20A in RPG)
- PACKED(15,5) = packed, 15 digits w/5 decimal places (15P 5 in RPG)
- POINTER(SPP) = Pointer. (Data type * in RPG – more info later!)

However, there are two data types that seem to cause a lot of confusion:
- BINARY(4) = 4 byte binary integer. (10I 0 in RPG)
- BINARY(4), UNSIGNED = 4 byte unsigned binary integer (10U 0 in RPG)
- CHAR(*) = Character field with a special length, not VARYING (Declare this as a long character field with options (*VARSIZE) on the prototype.)

NOTE: In RPG, we declare our numeric fields by the number of digits we can store in them. So a “9P 0” field is 5 bytes long, but stores a 9 digit number. A “10I 0” field is a binary integer that’s 4 bytes long, but stores a 10 digit number. NEVER USE THE “B” DATA TYPE, IT’S NOT A TRUE BINARY INTEGER. THE I AND U DATA TYPES ARE, AND THEY RUN MUCH FASTER, TOO.

API Description

On the API’s page, after the Parameter Summary.

The Send Program Message (QMHSNDPM) API sends a message to a call message queue or the external message queue. (The external message queue is the part of the job message queue that handles messages between an interactive job and the work station user. It is not associated with a specific call stack entry.) This API allows the current call stack entry to send a message to its caller, a previous caller, or itself.

In a multithreaded job, messages can be sent only to call message queues in the thread in which this API is called or to the external message queue. Messages cannot be sent to call message queues in other threads.

To send a message to a nonprogram message queue, see Send Nonprogram Message (QMHSNDNM) API.

Before coding your call to the QMHSNDPM API, see Dependencies among Parameters.

Authorities and Locks

Message File Authority
*USE
Message File Library Authority
*EXECUTE
Detailed Parameter Descriptions

On the API’s page, after the Authorities and Locks

Required Parameter Group

Message identifier

INPUT, CHAR(7)

The identifying code for the predefined message being sent, or blanks for an immediate message.

When sending an escape, notify, or status message, you must specify a message identifier. When sending a request message, you must use blanks. When sending other types of messages, you can use either a message identifier or blanks.

If you specify a message identifier, you must specify a qualified message file name. If you do not specify a message identifier, the API ignores the qualified message file name parameter.

Qualified message file name

INPUT, CHAR(20)

For a predefined message, the name of the message file and the library in which it resides. The first 10 characters specify the file name, and the second 10 characters specify the library. You can use these special values for the library name:

*CURLIB  The job’s current library
*LILIB  The library list

Errors the API can Return

At the end of each API’s manual page

Error Messages

<table>
<thead>
<tr>
<th>Message ID</th>
<th>Error Message Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPF24C5 E</td>
<td>Pointer to call stack entry not valid.</td>
</tr>
<tr>
<td>CPF24C6 E</td>
<td>Value of To call stack entry data type parameter not valid.</td>
</tr>
<tr>
<td>CPF24C8 E</td>
<td>Control boundary not found on call stack.</td>
</tr>
<tr>
<td>CPF24C9 E</td>
<td>Program boundary not found on call stack.</td>
</tr>
<tr>
<td>CPF24CB E</td>
<td>*PGMINAME requires a specified program name.</td>
</tr>
<tr>
<td>CPF24CC E</td>
<td>Call stack entry &amp;2 for *PGMINAME not found.</td>
</tr>
<tr>
<td>CPF24CD E</td>
<td>Module name cannot be specified when *PGMBDY is used.</td>
</tr>
<tr>
<td>CPF24CE E</td>
<td>Qualifier &amp;1 incorrect for use with pointer.</td>
</tr>
<tr>
<td>CPF24AC E</td>
<td>Either message identifier or message text must be specified.</td>
</tr>
<tr>
<td>CPF24AD E</td>
<td>Messages to remove must be *ALL if program message space is *ALLINACT.</td>
</tr>
<tr>
<td>CPF24A3 E</td>
<td>Value for call stack counter parameter not valid.</td>
</tr>
<tr>
<td>CPF24BF E</td>
<td>Module or bound-program name is blank.</td>
</tr>
<tr>
<td>CPF24B3 E</td>
<td>Message type &amp;1 not valid.</td>
</tr>
<tr>
<td>CPF24B4 E</td>
<td>Severe error while addressing parameter list.</td>
</tr>
<tr>
<td>CPF24B6 E</td>
<td>Length of &amp;1, not valid for message text or data.</td>
</tr>
<tr>
<td>CPF24B7 E</td>
<td>Value &amp;1 for call stack entry name length not valid.</td>
</tr>
</tbody>
</table>
### API Error Handling (1/2)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Dec</th>
<th>Hex</th>
<th>Use</th>
<th>Type</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Input</td>
<td>Binary(4)</td>
<td>Bytes Provided</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>Output</td>
<td>Binary(4)</td>
<td>Bytes Available</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>F</td>
<td>Output</td>
<td>Char(7)</td>
<td>Exception ID</td>
</tr>
<tr>
<td>15</td>
<td>F</td>
<td>8</td>
<td>Output</td>
<td>Char(1)</td>
<td>Reserved</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>0</td>
<td>Output</td>
<td>Char(*)</td>
<td>Exception Data</td>
</tr>
</tbody>
</table>

- This structure is passed in a parameter to the API.
- **Bytes Provided** should tell the API how big the DS is. (That way, you can control the size of the Exception Data field!) *You must set this before calling the API. Don't leave it blank!* (\(40404040 = 1,077,952,576\))
- **Bytes Available** tells you how much error data the API sent back.
- You can leave off the fields on the end, as long as Bytes Provided is correct.
- You can set **Bytes Provided** to zero if you’d like the API to send you an *ESCAPE* message when it fails.

**NOTE:** The CEE APIs, and the Unix-type APIs have separate mechanisms for error handling that I do not cover here. They are documented in the Knowledge Center, however.

### API Error Handling (2/2)

If you assume the API will always succeed do this. Then, if something weird does happen, the program will halt and there’ll be good diagnostic info in the job log.

```c
D ErrorCode     ds
D BytesProv     ds
D BytesAvail    ds

if ( BytesAvail > 0 )
    ErrMsg = MsgId + ' occurred called QMHSNDPM API!';
    // show ErrMsg to user!
endif;
```

If you want to handle errors in your code, use this syntax instead. Nothing will go to the job log, it’s up to you to handle errors:

```c
D ErrorCode     ds
D BytesProv     ds
D BytesAvail    ds
D MsgId         ds
D BytesAvail    ds
D MsgData       ds

if ( BytesAvail > 0 )
  CALLP QMHSNDPM( ...other parms here... ; ErrorCode);
  if ( BytesAvail > 0 )
    ErrMsg = MsgId + ' occurred called QMHSNDPM API!';
    // show ErrMsg to user!
  endif;
endif;
```

- The use of %SIZE is a good idea. Let the compiler do the work, and help you when you need to make changes.
- This way, if BytesAvail isn’t zero after calling the API, you know there’s an error.
Complex Parameters (Formats)

A format is a code that identifies the format of a data structure. (It’s similar in concept to a record format.)

A format name typically looks something like this:

DEVD0600

When an API can return different types of data, or can return it in many different formats (or would like to be able to do that at some point in the future!) it requests a format.

Let’s say you’re writing an interactive program, and you want to know the IP address of your user’s PC.

To find this out, you’ll need to retrieve information about the Display Device that he’s using. This is done with the “Retrieve Device Description (QDCRDEVD)” API.

This API returns all sorts of information about a device. There are hundreds of fields that it can return!

It returns different information, depending on what sort of device you’d like information about. A tape device (*TAP) has very different information than a display device (*DSP)!

 Formats in the Manual (1/3)

Retrieve Device Description (QDCRDEVD) API

<table>
<thead>
<tr>
<th>Required Parameter Group:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Receiver variable</td>
</tr>
<tr>
<td>2 Length of receiver variable</td>
</tr>
<tr>
<td>3 Format name</td>
</tr>
<tr>
<td>4 Device name</td>
</tr>
<tr>
<td>5 Error Code</td>
</tr>
</tbody>
</table>

Default Public Authority: *USE

Threatsafe: Yes

The Retrieve Device Description (QDCRDEVD) API retrieves information about a device description.

The first two parms tell the API which data structure to return info into.

The format name tells the API what the data structure looks like.

The “device name” tells the API which device you’re interested in.

But, what do you put for the format name?
To find the possible format names, scroll down to the detailed information for the required parameter group. This is what you’ll find:

**Format name**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT; CHAR(8)</td>
<td>The content and format of the information returned for each device description. The possible format names are:</td>
</tr>
<tr>
<td>DEVDO000</td>
<td>Basic device information.</td>
</tr>
<tr>
<td>DEVDO200</td>
<td>Detailed information for device category *APPC.</td>
</tr>
<tr>
<td>DEVDO300</td>
<td>Detailed information for device category *ASC.</td>
</tr>
<tr>
<td>DEVDO400</td>
<td>Detailed information for device category *BSC.</td>
</tr>
<tr>
<td>DEVDO500</td>
<td>Detailed information for device category *DKT.</td>
</tr>
<tr>
<td>DEVDO600</td>
<td>Detailed information for device category *DSP.</td>
</tr>
<tr>
<td>DEVDO700</td>
<td>Detailed information for device category *FNC.</td>
</tr>
<tr>
<td>DEVDO800</td>
<td>Detailed information for device category *HOST.</td>
</tr>
<tr>
<td>DEVDO900</td>
<td>Detailed information for device category *INTR.</td>
</tr>
<tr>
<td>DEVDI000</td>
<td>Detailed information for device category *NET.</td>
</tr>
<tr>
<td>DEVDI100</td>
<td>Detailed information for device category *PRT.</td>
</tr>
</tbody>
</table>

To learn how your data structure must be formatted, scroll down to the detail info for DEVDO600 (part of it is shown on the right.)
Sometimes, the offset of data that it returns won’t be at a fixed position. Instead, it’ll be variable.

OFFSET = Distance (in bytes) between the start of the data, and the point where the field starts. In other words, it’s a count of bytes from the start. The first field is always at offset 0 because it’s at the start.

Sometimes, the offset of data that it returns won’t be at a fixed position. Instead, it’ll pass you a variable that contains the offset of the field!

This is common when:

• Preceding data was variable-length.

• A list of repeating fields is returned. (such as a list of jobs on the system, list of objects in a library, etc.)

The best way to deal with variable offsets is with pointer logic.

Never, ever hard-code an offset when an API passes it to you in a parameter!
This is from format JOB10750 of the Retrieve Job Information (QUSRJOBI) API.
It’s for retrieving the library list for a given job.

Each Library at the variable offsets follows the format of a “Library Array Entry”. Here’s that format:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Type</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dec</td>
<td>Hex</td>
<td>Field</td>
</tr>
<tr>
<td>64</td>
<td>40</td>
<td>Binary(4) Offset to libraries in system library list</td>
</tr>
<tr>
<td>68</td>
<td>44</td>
<td>Binary(4) Number of libraries in system library list</td>
</tr>
<tr>
<td>72</td>
<td>48</td>
<td>Binary(4) Offset to product libraries</td>
</tr>
<tr>
<td>76</td>
<td>4C</td>
<td>Binary(4) Number of product libraries</td>
</tr>
<tr>
<td>80</td>
<td>50</td>
<td>Binary(4) Offset to current library</td>
</tr>
<tr>
<td>84</td>
<td>54</td>
<td>Binary(4) Number of current libraries</td>
</tr>
<tr>
<td>88</td>
<td>58</td>
<td>Binary(4) Offset to libraries in user library list</td>
</tr>
<tr>
<td>92</td>
<td>5C</td>
<td>Binary(4) Number of libraries in user library list</td>
</tr>
<tr>
<td>96</td>
<td>60</td>
<td>Binary(4) Length of one library array entry</td>
</tr>
</tbody>
</table>

Note: The decimal and hexadecimal offsets depend on the number of libraries you have in the various parts of your library list. The data is left-justified with a blank pad at the end. The array is sequential. It is an array or data structure. See CL Programming book for the total number of libraries that can be returned.

The fields repeat for each library object returned in the array.

Note that the length of that array entry is also variable.

The offset from the previous slide tells us where the first library is. The second library will be immediately after the first.
The Length of One Library Array Entry field tells us where the second one starts. (As well as the third, and fourth, and so on.)
Introduction to Pointers

The best way to handle variable offsets is with pointer logic.

\[ \text{POINTER} = \text{a variable that stores an address in the system’s main storage (or, “memory”).} \]

Just as a packed (or zoned) field is a variable designed to hold a decimal number, and a date field is designed to hold a date, and a time field is designed to hold a time, a pointer field is designed to hold an address in your system’s memory.

What can you do with pointer fields?
- Set them to \text{*NULL} (“this pointer doesn’t currently have an address in it.”)
- Store the address of another variable in them.
- Ask the system to reserve (or “allocate”) memory, then store the address of that memory in them.
- Add an offset to them (calculate the address X number of bytes later in memory)
- Subtract one pointer from another to calculate the offset between them.
- Base a variable on them

Based Variables
- Memory isn’t automatically reserved to store their values.
- Instead, you control the place in memory where they reside.
- You can change it on the fly by changing the pointer.

Trivial Pointer Examples

\[
\begin{align*}
\text{D FIELD1} & \quad s & 10A \\
\text{D p\_Field2} & \quad s & * \quad \text{inz}(\text{*NULL}) \\
\text{D FIELD2} & \quad s & 1A \quad \text{Based(p\_Field2)} \\
\text{D FIELD3} & \quad s & 7P \ 0 \ \text{inz}(1234567) \\
\end{align*}
\]

/\free
\[
\begin{align*}
\text{Field1} &= \text{‘Mashville’}; \\
p\_Field2 &= \%\text{addr(Field1)}; \\
\text{Field2} &= \text{‘N’};
\end{align*}
\]

Field1 now contains “Mashville”

\[
\begin{align*}
p\_Field2 &= \%\text{addr(Field1)} + 5;
\end{align*}
\]

Field2 now contains “i”

\[
\begin{align*}
\text{Field1} &= \text{‘Strawbeary’};
\end{align*}
\]

Field2 now contains “b”

\[
\begin{align*}
p\_Field2 &= p\_Field2 + 2; \\
\text{Field2} &= \text{‘x’}; \\
p\_Field2 &= \%\text{addr(Field3)} + (\%\text{size(Field3)} - 1); \\
\text{Field2} &= \text{‘x’}0D’;
\end{align*}
\]

Field1 now contains “Strawberry”; Field3 contains -1234560
Hey wait, what happened to APIs?

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>Type</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>62</td>
<td>A</td>
<td>CHAR(4)</td>
<td>Reserved</td>
</tr>
<tr>
<td>64</td>
<td>40</td>
<td>BINARY(4)</td>
<td>Offset to libraries in system library list</td>
</tr>
<tr>
<td>68</td>
<td>44</td>
<td>BINARY(4)</td>
<td>Number of libraries in system library list</td>
</tr>
<tr>
<td>72</td>
<td>48</td>
<td>BINARY(4)</td>
<td>Offset to product libraries</td>
</tr>
<tr>
<td>76</td>
<td>4C</td>
<td>BINARY(4)</td>
<td>Number of product libraries</td>
</tr>
<tr>
<td>80</td>
<td>50</td>
<td>BINARY(4)</td>
<td>Offset to current library</td>
</tr>
<tr>
<td>84</td>
<td>34</td>
<td>BINARY(4)</td>
<td>Number of current libraries</td>
</tr>
<tr>
<td>88</td>
<td>58</td>
<td>BINARY(4)</td>
<td>Offset to libraries in user library list</td>
</tr>
<tr>
<td>92</td>
<td>5C</td>
<td>BINARY(4)</td>
<td>Number of libraries in user library list</td>
</tr>
<tr>
<td>96</td>
<td>60</td>
<td>BINARY(4)</td>
<td>Length of one library array entry</td>
</tr>
</tbody>
</table>

Note: The decimal and hexadecimal offsets depend on how many variables there are in the various parts of your library list. The data is stored sequentially at the end. The array is sequential. It is an array or data structure. See the CL Programming book for the total number of libraries that can be returned.
API Variable Offset Example (1/3)

FQSYSPRT O F 80 PRINTER

D QUSRJOBI   PR     ExtPgm('QUSRJOBI')
D RcvVar     32767A options(*varsize)
D RcvVarLen  10I 0 const
D Format     8A const
D QualJob    26A const
D InternalId 16A const
D ErrorCode  32767A options(*varsize: *nopass)
D Reset      1A options(*nopass)

D MyData     ds     based(p_MyData)
D OffsetUsrLibl 10I 0 overlay(MyData: 89)
D NumUsrLibl 10I 0 overlay(MyData: 93)
D EntryLen   10I 0 overlay(MyData: 97)

D LibEntry   ds     based(p_LibEntry)
D LibName    10A
D Text       50A
D ASPNo      10I 0
D ASPName    10A

for x = 0 to (NumUsrLibl - 1);
    p_LibEntry = p_MyData + OffsetUsrLibl + (x * EntryLen);
    except PrintLib;
endfor;
dealloc p_MyData;
*inlr = *on;
/end-free

DataSize = 1024 * 1024;
p_MyData = %alloc(DataSize);
QUSRJOBI( MyData: DataSize: 'JOBI0750': '*': *blanks );
### User Spaces

**USER SPACE** = A disk object that acts very much like a memory allocation.

**Characteristics of a user space:**

- Created by calling an API.
- Can be marked “auto-extend” so that they’ll automatically get bigger as needed. (With memory, you have to re-allocate to get a larger space.)
- You can get a pointer to a user space, and use it just as you would memory.
- You can base variables on a user space pointer, and use those variables like you would any other RPG variable.
- As a disk object, it can be saved in-between calls.
- Useful for remember “last time I ran this” values.
- It can be backed up to tape or optical media.
- It can be shared with other jobs on the system.
- APIs exist for reading/writing user spaces for languages that don’t support pointers.
- That includes OPM languages.
- APIs that need to return data that might be too large for an HLL variable will put their data in a user space. That way, it’s accessible from any IBM i language.
List APIs

Many of the APIs that need to return a list of something (jobs, libraries, objects, modules, etc.) are called “List APIs”.

Characteristics:
- Accept a user space library/name to store the results in.
- The generated user space always starts with a “generic header”
- Generic header contains offset, count and entry size information needed to read the list.
- The format of the list entries will vary depending on the API.

For example, you might want to get a list of the interactive jobs that are active on the system. So you’d look for an API that does that.
- APIs by Category
- Work Management (deals with how the system processes it’s workload)
- List Jobs (QUSLJOB) sounds good!

List API Example (1/4)

<table>
<thead>
<tr>
<th>FQSYSPRT</th>
<th>O</th>
<th>F</th>
<th>80</th>
<th>PRINTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>D QUSCRUS</td>
<td>PR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D UserSpace</td>
<td>20A</td>
<td>CONST</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D ExtAttrib</td>
<td>10A</td>
<td>CONST</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D InitialSize</td>
<td>10I 0</td>
<td>CONST</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D InitialVal</td>
<td>1A</td>
<td>CONST</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D PublicAuth</td>
<td>10A</td>
<td>CONST</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D Text</td>
<td>50A</td>
<td>CONST</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D Replace</td>
<td>10A</td>
<td>CONST options(*nopass)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D ErrorCode</td>
<td>32767A</td>
<td>options(*varsize:*nopass)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

D QUSPTRUS  PR ExtPgm('QUSPTRUS')
D UserSpace | 20A | CONST |
D Pointer | * |

D QUSDLTUS  PR ExtPgm('QUSDLTUS')
D UserSpace | 20A | CONST |
D ErrorCode | 32767A | options(*varsize) |
List API Example (2/4)

List API Example (3/4)

QUSLJOB (‘JOBLIST QTMP’
: ‘QUSLJOB’
: 26A
: CONST)

QUSLJOB (‘JOBLIST QTMP’
: ‘QUSLJOB’
: 26A
: CONST)

QUSLJOB (‘JOBLIST QTMP’
: ‘QUSLJOB’
: 26A
: CONST)

Create a User Space called QTEMP/JOBLIST, that's 1mb long.

List all active jobs to user space.

Get a pointer to the user space.

Calculate the offset to each entry, and point the ENTRY data structure at it.

If it's an interactive job, print it out.

Delete the user space and end the program.
Drumroll please... and the results are....

<table>
<thead>
<tr>
<th>OQSYSPT</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>JobName</td>
</tr>
<tr>
<td>O</td>
<td>JobUser</td>
</tr>
<tr>
<td>O</td>
<td>JobNbr</td>
</tr>
</tbody>
</table>

**Output specs to print job identifiers:**

Display Spooled File

<table>
<thead>
<tr>
<th>File</th>
<th>Control</th>
<th>Find</th>
</tr>
</thead>
<tbody>
<tr>
<td>QSYSPRT</td>
<td></td>
<td>*...+...1...+...2...+...3...+...4...+...5...+...6...+...7...+...</td>
</tr>
<tr>
<td>QPADEV0001</td>
<td>BIZUJAME</td>
<td>239996</td>
</tr>
<tr>
<td>DSP01</td>
<td>KLEMSCOT</td>
<td>241320</td>
</tr>
<tr>
<td>ROGER</td>
<td>KLEMROGE</td>
<td>242304</td>
</tr>
<tr>
<td>SYSCON</td>
<td>QSECOFR</td>
<td>242331</td>
</tr>
<tr>
<td>DSP07</td>
<td>MARYZ</td>
<td>242326</td>
</tr>
<tr>
<td>89S1</td>
<td>CHERYL</td>
<td>242223</td>
</tr>
</tbody>
</table>

**More Information**

Getting Started with APIs (Scott Klement: System iNetwork Programming Tips)
http://iprodeveloper.com/rpg-programming/getting-started-apis-0

Getting Started with APIs, Part 2

Getting Started with APIs. Follow up to Part 2

Getting Started with APIs, Part 3
http://iprodeveloper.com/rpg-programming/getting-started-apis-part-3-0

Getting Started with APIs, Part 4
http://iprodeveloper.com/rpg-programming/getting-started-apis-part-4

APIs by Example (Carsten Flensburg)
http://iprodeveloper.com/search/results/APIs%20By%20Example

Fun with Pointers (Scott Klement: Personal Web site):
http://www.scottklement.com/rpg/pointers.html

IBM i Knowledge Center:
http://www-01.ibm.com/support/knowledgecenter/ssw_ibm_i/welcome
This Presentation

You can download a PDF copy of this presentation from:
http://www.scottklement.com/presentations/

Thank you!