

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

## Writing RPG Code Then and Now

[System iNEWS Magazine](#)

[Carsten Flensburg](#)

Carsten Flensburg

Wed, 10/01/2008 (All day)

[Click here](#) to download the code bundle.

To report code errors, email [SystemiNetwork.com](mailto:SystemiNetwork.com)

It recently occurred to me that I've been programming longer than I've done anything else in my professional life. It's scary —

there's still so much to learn and comprehend, and so many hard-acquired skills seem to last only so long before emerging new technologies, methodologies, and programming languages du jour take over the agenda. In fact, change often seems to be the only constant in this business.

In the early '90s, I ventured into the world of RPG and CL and have since been devotedly occupied with the challenge of writing solid, well-structured programs. Here, I share with you the style and techniques I use when I write an RPG application or program. I've chosen and slightly rewritten a command called Work with Output Queue Authorities (WRKOUTQAUT), a utility that displays the output queue authorities assigned to one or more user profiles.

My approach is based primarily on personal preference and perception developed and adapted over the years rather than scientific principles or "programming schools." Given this limitation, it is still my intention to provide you with the inspiration that comes from looking over a fellow programmer's shoulder.

### The WRKOUTQAUT Command

The original objective of the WRKOUTQAUT command was to provide a quick overview of user profiles' authorities to a specific output queue. Output queues contain all kinds of information in the form of spooled files waiting to be printed — some of which contain confidential and sensitive data. Spooled files are part of important business processes such as invoicing, confirming order confirmations, and distributing papers, so access to and management of the output queues handling these spooled files should be very restrictive and well devised. The ability to immediately verify and manage authority to a specific output queue from one Work with panel allows for tighter security and reduces the time spent on management tasks when compared with native commands.

Now to go into detail about specific requirements and design considerations for WRKOUTQAUT (for a list of source specifications for WRKOUTQAUT, see "WRKOUTQAUT Source Specification," below. For the UI, I usually choose User Interface Manager (UIM) to define the screens and printed lists. The necessary dialog and interaction with the list panel group is performed by UIM APIs and exit programs that provide a robust, consistent, and highly functional UI with a minimum of programming effort.

Because the UIM handles the screen dialog, I can concentrate on the core functionality of my utility. Although the UIM programming techniques are beyond the scope of this article, the System

iNetwork Programming Tips newsletter ([SystemiNetwork.com/sipt](http://SystemiNetwork.com/sipt)) contains a number of articles that explain the anatomy, employment, and usefulness of UIM list panel groups.

## Analysis Precedes Coding

Designing, planning, and coding the components that comprise the final utility begins with the modeling of the initial program interface, which in this case is provided by the WRKOUTQAUT command definition. As [Figure 1](#) shows, it's a simple interface, primarily made up of an output queue name and a generic user profile name. The latter includes the special value \*ALL, which lets you select all user profiles on a system. Also, an optional parameter lets you direct the output to a spooled file instead of the display panel.

Whenever a command interface is involved, I usually include a command validity-checking program (VCP) to validate input parameters — in this case, the qualified output queue name parameter. If the specified output queue can't be found (e.g., because of a typo or lack of library qualification), the user should be informed right away and given the chance to correct the input. There's no point in calling the command processing program (CPP) to find out, because this would force the user to reprompt the command in order to carry on. The same precaution also applies when a specific user profile name is specified for the second parameter. In addition to the other checks, the VCP also verifies that the user profile executing the WRKOUTQAUT command includes the required \*ALLOBJ and \*SECADM special authorities.

Another important aspect of employing a command interface is the option (and in my opinion, requirement) to include a help text panel group to explain the command's usage, parameters, restrictions, and error messages. Providing online, cursor-sensitive help text increases the usability of the command and helps avoid user errors and misunderstandings derived from lack of precise instructions and documentation. For the WRKOUTQAUT command, you must also provide help text for the Work with panel, including a general explanation of the panel as well as all fields and list columns. All this help text goes in the same help text panel group.

You will need several subprocedures to perform a variety of functions related to validation and message handling as well as retrieval of output queue, user profile, and authority information. If more than a couple of subprocedures are in play, and if grouping the functions make sense, I use service programs and binder language to create one or more function library service programs and place my subprocedures in these service programs. That helps me focus on the purpose and structure of the program itself, and also supports and encourages reuse of the subprocedures. For general-purpose functions, I create and maintain application-wide or systemwide service programs and enforce naming standards to avoid name conflicts.

## What You'll Need

The service program will contain all the subprocedures that I can isolate from the VCP and CPP. In other words, I must identify all the code chunks and functions that make sense to encapsulate into individual subprocedures, each having one specific function and each provided with a well-defined procedure interface specifying the applicable input parameters, return value, and sometimes output parameters to satisfy special requirements.

To begin, I often sketch out the program's or utility's data output to help me establish more precisely what functions I'll need for the desired outcome. In this case, the following requirements come out of that process:

### Output queue information:

- queue library
- queue owner
- public authority
- authorization list

**User authority information:**

- output queue
  - start writer
  - add spooled file
  - work with
  - clear, hold, release
  - change
  - spooled files
  - display, copy, send
  - change, delete, hold, release
- output queue authority and authority source

All this information is readily available through APIs, so I can quickly complete my design of the functions to include in my service program and the core functionality that must remain in the CPP. Following a similar requirement analysis of the VCP, the service program contains the subprocedures listed in "Service Program Subprocedures" below.

**The Service Program**

Let's briefly walk through the essential parts of the service program implementing our list of subprocedures (you can download or extract the service program in the code bundle at [SystemiNetwork.com/code](http://SystemiNetwork.com/code)). In the global declaration section preceding the subprocedure prototypes, I've defined some API return information data structures to be shared by the subprocedures. One is the API error data structure `ERRCO100`, which is used by practically all subprocedures. Because the data structure is always interrogated immediately following an API call, I can be certain the data structure subfields always reflect the actual status of the API call and contain current values, so there's no need to define the `ERRCO100` data structure repeatedly in each subprocedure.

I have the option to override the global definition of the `ERRCO100` data structure by placing a different version of the data structure locally in a subprocedure. For example, if I want to receive an exception message instead of the message ID and message data in a specific situation, I can code a short version of the `ERRCO100` data structure defining zero bytes being available for return information, thereby triggering the API to behave as desired.

I also define the `USRIO200` user information data structure returned by the Retrieve User Information (`QSYRUSRI`) API as a global data structure. I decided to do so to avoid paying an unnecessary performance penalty caused by calling `QSYRUSRI` multiple times to retrieve the same user profile's different attributes when loading the list entries of the Work with panel. Because this information retrieval is performed consecutively for each user profile to be added to the list panel, I need to call the `QSYRUSRI` API only the first time; the following three times the data is retrieved, the `USRIO200` data structure already contains the correct information. I simply added a check in each of the relevant subprocedures, matching the input parameter user profile name with the `USRIO200` data structure user profile name, and then call the API only if this test is not passed.

In the special case where the WRKOUTQAUT command is called for one specific user profile, and I use the F5 key is to update the panel (e.g., following an update of the user profile or user profile authorities), I need to ensure that these updates are correctly reflected upon redisplay of the panel. I achieve this update by adding a subprocedure that enables the caller to clear the USRIO200 data structure user profile name, and then call that subprocedure immediately before entering the loop that builds the list of user profiles. This technique lets the QSYRUSRI API be called again during the subsequent user profile information retrieval, thus refreshing the USRIO200 data structure.

The message-handling subprocedures need some attention — sending a program message requires a specific and correctly defined target program message queue to make the dispatched message appear in the right place on the right screen. The Send Program Message (QMHSNDPM) API allows for a multitude of combinations of special values, program and module names, call stack counters, and invocation pointers. To keep the scheme as simple and straightforward as possible, I took advantage of the ILE activation group facility.

By having the message subprocedure caller (either the WRKOUTQAUT command's VCP or CPP) run in a \*NEW activation group, I ensure that the caller programs also constitute an activation group control boundary. Specifying \*CALLER as the service program activation group makes the service program run in the same activation group as the program activating it. With this simple setup, I can specify the special value \*CTLBDY (control boundary) as the target message queue and indicate 1 as the call stack counter on the QMHSNDPM API call. These values let me identify the call stack immediately preceding the VCP or CPP as the target program queue. As a result, the message will display on the screen from where the WRKOUTQAUT command was executed. Relying on such a technique warrants proper documentation or a command build script to ensure that the crucial activation group attributes survive possible recompilations of the involved programs.

## The VCP

As I mentioned earlier, the VCP performs three simple checks, and if an error occurs, the program returns the two standard VCP messages from the system QCPFMSG message file. The first is the diagnostic CPD0006 message, which specifies the actual cause of the validity check failure as the message data. The CPD0006 message is followed by the CPF0002 escape message, which immediately terminates the command being validity-checked and displays the messages directly on the command prompt.

The validity checks verify the existence of the specified output queue and user profile parameter (as far as the latter is concerned, only in the event that a specific user profile was given as input). Furthermore, the VCP verifies that the user profile running the WRKOUTQAUT command contains the \*ALLOBJ and \*SECADM special authority. Both of these special authorities are required to list any given user profile as well as to perform the available actions and function against the output queue and user profiles from the WRKOUTQAUT command's list panel. Since the CPP doesn't rely on adopted authority from its owner, but instead uses the actual authority of the current user, it's acceptable to run the special authority check from the VCP. Otherwise, I would perform or duplicate the check in the CPP as well ensure that it is not circumvented by calling the CPP directly.

## The CPP

At this point I'm ready to write the CPP. Since the UIM performs the user interaction, and it isn't the first time I've written a Work with list program based on a UIM list panel group, I locate suitable existing programs from which to copy the basic skeleton and relevant code snippets. When necessary, I update the code to reflect current programming standards, style, and techniques. That

effort helps me both maintain continuity in my programming practice and continue to reflect on and improve my coding style, standards, and techniques.

The CPP constitutes the core of the WRKOUTQAUT command, so let's walk through some of the details and discuss the considerations that are involved in the process.

To ensure that my program will keep working as intended following future compilations and to document possible dependencies, I use a combination of source header comments specifying the required module compilation and (service) program creation commands and H-spec compiler directives. If necessary, I also document other types of setup and configuration information. Additionally, I try to organize the D-specs in a common sequence and style so I can quickly locate the various types of specifications as I write or edit the program. The sequencing approach I use is as follows:

1. system and file information data structures as well as the API error data structure
2. global variables and constants (variables specified in mixed case and constants in all upper case to allow immediate distinction between the two)
3. API input, return data structures, and all other data structures
4. external system program and procedure specifications (I like to group these according to the functionality they provide and the interaction that they perform: UIM APIs in one group, Open List APIs in another group, and so on)
5. external user program and procedure specifications, typically provided by application or function library service programs and grouped as item 4
6. internal and local subprocedures, defined within the program itself
7. module entry parameter data and parameter list specification, thus appearing immediately before the executable code

These groupings work for me, but something else might work better for you. The main idea is that keeping some sort of consistency and standard within a shop is helpful for navigating, reviewing, and editing your RPG source code.

### Old Principles, New Code

When it comes to writing executable code, I'm basically still adhering to the principles I learned in early *NEWS/400* articles: To make the program flow distinct and easy to follow, I aim to make it stand out in those places where the decisions controlling the different paths of execution are made. To achieve that goal, I separate the individual actions or functions derived from these decisions, leaving only the primary functions and basic decisions in the main line of the code.

Obviously, this method requires that I place the extracted functions and code chunks elsewhere. Whenever possible and appropriate, I prefer to create subprocedures to perform well-defined and well-interfaced functions. This approach provides superior documentation of the function performed, as it not only names the function but also directly documents both the exact input parameters and the target of the return value. There are other significant advantages to subprocedures, and they are truly the most prominent and crucial enhancements added to the RPG language in its long history. Not only do they influence and improve the programming syntax, but even more so the design considerations and mindset of the RPG programmer.

However, if there's no natural interface for input parameters or return values and the code is acting on global variables only, I use mostly subroutines to organize and structure my code. Thus, when I encounter a subroutine, I'm also given information about the intention and scope of the work being performed in that subroutine.

To support program readability, I exclusively use free-format specifications when possible. Doing so increases the space available for code statements and allows for indentation, which is critical to letting you quickly spot the code segments that are dependent on certain conditions. I'm really looking forward to the day when free format will be available for all RPG specifications. That will finally break RPG free of its aged reputation and historical chains. When writing RPG, I use visual effects. I put blank lines before and after statements that deserve particular attention or, for example, to group ENDxx statements. Likewise, for expressions that span more than one line, I break down the expression in single elements and place each element aligned on individual lines. Although some might view that approach as needless fiddling and wasting precious time, in my experience it increases the immediate readability and clarity of the code. For example, this format lets you quickly verify the exact number of parameters passed on a procedure call or spot where the concatenation of text lines and variables breaks. It also makes it easier to copy, replace, insert, or model elements in an expression. As an example of these style and design principles, [Figure 2](#) displays the main line of the CPP and the two subroutines controlling the display or printing of the user list.

### When It Works, You're Half Done

Although slightly exaggerated, the above subhead is true in that whether I write a program from scratch or find a starting point in my source library, I'm not at all done when I get my program working. Dealing with approaching deadlines or daily business tasks certainly can seem irresistible, but I always try to find the time to tidy up my code. My most prominent enemies are variables or constants that were not used after all, shortcuts and sloppy constructs introduced as quick paths to verify an assumption, unnecessary code complexity resulting from not fully comprehending the task at hand, and irrelevant, redundant, or replicated code. The cleaner and better organized you leave your code, the easier it is to get back to or hand over in the future.

I'll run out of space long before I finish the discussion that I've started here, but ideally the discussion shouldn't end as long as the RPG language is still developing as vigorously as it has been thus far. RPG is still rightfully regarded as the best programming language for writing robust, reliable, and competitive business applications. System architecture and design increasingly focus on system integration and separation of business logic and the presentation layer. This fact, paired with the steady growth in communication capacity and development of standardized, flexible data protocols, makes it clear that RPG programming skills will continue to provide value in tomorrow's marketplace, if RPG programmers fully exploit the ever-expanding applicability of RPG.

***Carsten Flensburg*** is a ***System iNEWS*** technical editor.

### Service File Subprocedures

Here is a list of subprocedures used in our service program and the functions they perform:

#### **Subprocedure: Function:**

ChkObj()	Check object existence
ClrUsrCch()	Clear user profile information cache
GetGrpPrf()	Get user profile's group profile
GetJobTyp()	Get job type
GetNbrSupGrp()	Get user profile's number of supplemental group profiles
GetObjAutL()	Get object authorization list

GetObjOwn()	Get object owner
GetPubAut()	Get object public authority
GetUsrCls()	Get user profile's user class
RtvMsg()	Retrieve message description text
SndCmpMsg()	Send completion message
SndDiagMsg()	Send diagnostic message
SndEscMsg()	Send escape message
SndStsMsg()	Send status message
ValSpcAut()	Validate user profile special authority

— *C.F.*

### WRKOUTQAUT Source Specifications

The following source members are used when creating the WRKOUTQAUT command:

Member	Type	Text
CBX605	RPGLE	Work with Output Queue Authorities - CPP
CBX605V	RPGLE	Work with Output Queue Authorities - VCP
CBX605S	RPGLE	Work with Output Queue Authorities - Services
CBX605B	SRVSR	Work with Output Queue Authorities - Binder source
CBX605H	PNLGRP	Work with Output Queue Authorities - Help
CBX605P	PNLGRP	Work with Output Queue Authorities - Panel Group
CBX605X	CMD	Work with Output Queue Authorities
CBX605M	CLP	Work with Output Queue Authorities - Build command

To ease the WRKOUTQAUT command build process, I've included the CBX605M CL program. Simply compile and run the CBX605M program, following the instructions in the source header, and providing your target library as the only parameter. To download the code bundle, go to [SystemiNetwork.com/code](http://SystemiNetwork.com/code).

— *C.F.*

**Source URL:** <http://iprodeveloper.com/rpg-programming/writing-rpg-code-then-and-now>