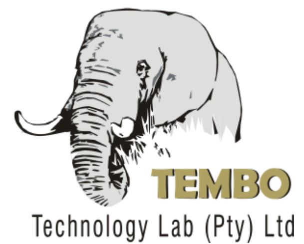


Modernized RPG IV for ILE Application Code Error Handling Services

Author: Tommy Atkins (May 2016)

Chief Development Officer: TEMBO Technology Lab (Pty) Ltd.

Introduction

Many of today's heritage programs, due to their origins and years of evolution, are large and complex. In addition, they have been maintained, probably by a significant number of different programmers and therefore contain a mixture of styles and standards which add significantly to their complexity.
This complexity made maintenance and enhancement difficult and sometimes risky. The tendency was (yours truly not excluded) to code for and handle only those errors which were most likely to occur and to ignore the less likely ones, knowing that the OS would crash the program if something happened. It would then be possible to enhance the program to deal with that issue.

Over the years the RPG language and the AS/400, iSeries and IBMi operating systems have matured significantly adding many functions, methods and APIs to allow for the more effective trapping and handling of errors.

As we progress with the modernization of our application code into the ILE (Integrated Language Environment) paradigm and begin to implement the MVC model, the large monolithic programs of the past will begin to change.

This change will involve the shifting of validation/synchronization code from the application into the DDL defined databases as triggers and constraints, encapsulating multi-use code into procedures bound into service programs, providing "border control" in the form of I/O Servers and fragmenting the remaining application code into more manageable and maintainable service programs.

As this modernization change begins to take hold and evolve many things will also change in the underlying processes supporting the application jobs. The two things which specifically relate to this topic of "Error Handling" are the increasing depth of the program/procedure call stack and the ring-fencing of application resources into "Activation Groups".

Both factors as well as an increasing need to identify and locate errors accurately and report them in User Friendly terms to the correct level within the application, which may be a graphical interface, requires the development of a new philosophy regarding the trapping and handling of error conditions within modernized applications.

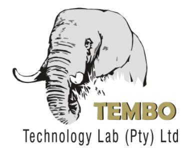
A Modern Method

In the modern RPG IV for ILE application development methodology the most important truth regarding errors is that ALL errors that occur should be trapped and handled in the most appropriate manner for the existing circumstances.

In the highly procedural, component-based structures of the modern MVC development style it is not wise to allow any error to go unhandled. This could allow invalid data into the database, procedures which execute partially or not at all or any number of other unpredictable situations all of which can be detrimental to the successful operation of the application.

Fortunately for us the IBM operating systems, right from the early CPF (System/38 operating system) and OS/400 to the latest IBMi operating system, have provided us with an example of the best way to handle errors and has also, with the evolution of the OS and RPG, provided us with all the tools to allow us to mimic the method which the OS has been using with great success for many years.

This will provide a method of handling messages in a consistent way across applications and operating system and ensure that the consistency is of benefit to all developers in the team.



Before I begin with the more technical aspects, I would like to say that the IBMi and RPG IV provide numerous mechanisms to both trap and handle errors occurring within application solutions and there are probably dozens of ways of solving the requirements for managing errors.

The method I am providing here was initially developed in the mid-nineties with the specific intention of mimicking the way the operating system moves both diagnostic as well as error messages up the call stack (percolation) until the error reaches a level where it can be handled. This method has been enhanced continuously since then to accommodate variations and additional requirements as well as adopting new functionality as it became available. The last change to these procedures was made in 2013 to accommodate the need for more user-friendly messaging when using SQL imbedded into RPG IV code.

AO Open Source Project

The source code making up the complete service program is available as open-source code from the Adsero Optima web site (https://www.adsero-optima.com/open-source/). It is covered under the Apache License V2.0 for open-source software, a copy of which is available from (https://www.apache.org/licenses/LICENSE-2.0).

I will be describing each of the procedures making up the "Error Handler" service program individually to ensure that anybody who wishes to use these procedures has a complete understanding of their function.

When using open-source code, it is advisable and essential to "Own" the code and make every effort to understand the method and mechanics of the procedures & functions. It is only in this way that it will be possible to evolve the code as is needed when moving forward with the modernization process.

The Service Program (ERRSRV@@)

The service program is bound together from 5 modules, containing 19 callable procedures, each performing a specific function. There are a few other procedures included in module 3 are exported to be used internally by the service program and should not be called by an application module/program.

The first thing to do when designing a service program of this type, which will be used in many applications, is to think of the parameters which will be needed. The ideal is to reduce the number of parameters required on a call to an absolute minimum thereby simplifying the use of the procedures inside the application code. The perfect solution is to have the same parameter list for each procedure, but this is not always possible, as in this case.

Parameters

Parm	Description	Attributes	Keyword	Default
MI	Message Id.	char(7)	const	
MF	Message File	char(10)	const	ERRMSGF
MD	Message Data	varchar(2048)	const	*blank
MT	Message Target	char(1)	const	1

The procedures/functions available in this service program use one or more of the parameters listed above. Optional parameters have been provided with specific default values, which if not satisfactory will need to be changed in the source code of the procedures.

If an optional parameter needs to be left out and none of the following parameters are specified, the call statement can be cut off at the last needed parameter.

However, if an optional parameter needs to be followed by a subsequent parameter value, then the unquoted placeholder *OMIT must be used in place of the omitted parameter.

All procedures in this service program are provided with a 'Copy Book', included in the ERRSRC file, which contains the procedure definition and any other variables which may be needed.

Message Id.
This parameter is mandatory and wherever used must contain a valid pre-defined message id.

Message File
The default value for the message file is currently set in the code as "ERRMSGF" using the *LIBL.

Message Data
This parameter contains the variable message data to be substituted into the specified predefined message variables at the time of sending, retrieving or receiving. This parameter has a variable length of 2048 bytes and can be used, when required, in one of 2 ways.

For Message Data from multiple sources, or a single field:
Populate this parameter from a structure or single field in accordance with the variables defined in the message or pass the message data as a constant, expression, structure name or field name on the call.

For Message Data contained all in one record:
Populate the parameter from the record data structure so that it contains the whole record. This means specifying the record data structure name on the call.

The message variables within the message definition would then be constructed to utilize whatever information it required from the record.

Different messages would then pick different elements from the record for substitution, making the application code standard no matter what the message was that was being used.

Unused chunks of the record can be stepped over by using a single variable as a filler. Be careful to calculate the length of this filler in BYTES correctly otherwise misalignment of the substitution will occur.

Message Target
This parameter identifies the call stack message queue which is to be the target when moving or sending messages. The default is '1' which means the target is the caller of the procedure which called the error handling procedure. Other numeric values indicate which call stack entry is to be the target of the message. Any numeric call stack entry will always be incremented automatically by 1 by the handler. This is due to the additional call level generated when calling the Error Handler procedure itself.

Two other special values are catered for;

- A value of 'P' indicates that the message must be moved or sent to the procedure at the closest "Program Boundary" in the call stack.
- A value of 'C' indicates that the message must be moved or sent to the procedure at the closest "Control Boundary" of the current activation group in the call stack.

NOTE:
The keyword "const" is defined in the error handling procedures to ensure that these procedures do not change the values of the parameter and to allow for the use of either variables or literal constants as parameters on the call.

Application Programming Interfaces (API's)

A significant amount of use is made of system APIs within these modules. If there is any doubt as to the usage or behaviour of an API used in one of these procedures, please use the link below to access the IBM documentation for a specific API to fully understand its operation.

https://www.ibm.com/docs/en/i/7.2?topic=category-message-handling-apis

Module #1 (ERRSRV@01)

Module #1 contains 10 procedures related mainly to the movement of messages from one call stack level to another level higher up the call stack. This is the process I call the "percolation" of messages up the call stack.

This module also contains a procedure that is used internally and is not exported, called “Cst_Error”. This procedure is used in the ERR03 procedure only and uses the AOFCMLF data file to translate constraint errors into more user-friendly messages if they have been defined,

The following sections describe the action of each of these procedures in detail.

ERR00: Clear Current Message Queue

Parameters: None

All messages are cleared from the message queue of the calling program.

ERR00A: Clear All Inactive Message Queues

Parameters: None

All messages are cleared from the message queues of all the inactive programs in the call stack.

ERR01: Move *DIAG Messages

Parameters:

```
MT      int(10)      const   options(*nopass);
```

Moves all *DIAG messages from the calling programs message queue to the program message queue of the program in the call stack identified by the target parameter.

ERR02: Re-Send *ESCAPE Message

Parameters:

```
MT      int(10)      const   options(*nopass);
```

Resends the last *ESCAPE message in the calling programs message queue to the program message queue of the program in the call stack identified by the target parameter.

ERR03: Move *DIAG and Re-Send *ESCAPE Messages

Parameters:

```
MT      int(10)      const   options(*nopass);
```

This procedure combines the action of ERR01 followed by the action ERR02 into a single call. This is the most used procedure to achieve "percolation" of messages up the call stack.

ERR04: Retrieve *LAST Message Id

Parameters: None

Function Returns: Message Id. char(7)

This procedure returns the message id. of the last message received by the calling program. It is frequently used to identify the message which has occurred and to take appropriate action based on the value.

ERR05: Move *DIAG and *ESCAPE Messages

Parameters:

```
MT      int(10)      const   options(*nopass);
```

This procedure performs a similar action to ERR01, except that both *DIAG and *ESCAPE messages are moved to the call stack identified by the target parameter. In this case any *ESCAPE messages that are moved are changed to *DIAG type messages in the target program message queue.

ERR06: Move *ESCAPE Messages Only

Parameters:

```
MT      int(10)      const   options(*nopass);
```

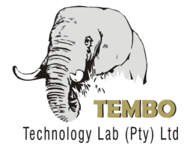
Like ERR05, except ONLY *ESCAPE messages are moved. The messages are still changed to *DIAG messages during the move.

ERR07: Receive *LAST Msg. Id. & Text + *REMOVE

Parameters:

```
MI      char(7);
MD      char(132);
MT      int(10)      const   options(*nopass);
```

Function Returns: Boolean 0=No Msg. Found, 1=Msg. Found



Receives the last message from the target message queue and returns the message id. in the MI parameter. The associated 1st level text is returned in the MD parameter.

ERR08: Receive Target Program Messages to USP Pointer

Parameters:

USPP	pointer	value;	
MT	int(10)	const	options(*nopass);

Receives up to 30 messages from the target procedures message queue and places them into a *USRSPC identified by the USPP parameter. The *USRSPC addressed by USPP must be of length of 4174 bytes, the first 4 of which is used as an uns(10) field to contain a count of the number of messages returned.

The calling program is responsible for creating the *USRSPC and providing the pointer (USPP).

Each entry returned in the user space consists of a message id. (7 bytes) + message text (132).

Module #2 (ERRSRV@02)

Module #2 contains 5 procedures, all of which are related to the sending of pre-defined messages to various destinations.

The following sections describe the action of each of these procedures in detail.

ERR10: Send *DIAG Message

Parameters:

MI	char(7)	const;	
MF	char(10)	const	options(*omit:*nopass);
MD	varchar(2048)	const	options(*omit:*nopass);
MT	char(1)	const	options(*nopass);

This procedure sends a pre-defined *DIAG message, specified by the first 3 parameters, to the program message queue as defined by the target parameter (MT), if specified.

If the target is not defined, the effect will be to send the *DIAG message to the caller of the program using this procedure.

ERR11: Send *ESCAPE Message

Parameters:

MI	char(7)	const;	
MF	char(10)	const	options(*omit:*nopass);
MD	varchar(2048)	const	options(*omit:*nopass);
MT	char(1)	const	options(*nopass);

This procedure sends a pre-defined *ESCAPE message, specified by the first 3 parameters, to the program message queue as defined by the target parameter (MT), if specified.

If the target is not defined, the effect will be to send the *ESCAPE message to the caller of the program using this procedure.

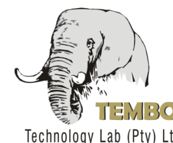
NOTE:

When sending an *ESCAPE message from an RPG program, the effect is to immediately terminate the program and return control to its caller. The same result occurs when using this procedure.

ERR12: Send *INFO Message to *EXT

Parameters: MI, MF, MD

This procedure is designed to send an *INFO message type to the external message queue (*EXT). The external message queue is known to most as the "Job Log". This procedure allows the application to annotate the job log for a job, if required.



ERR13: Send *STATUS Message to *EXT

Parameters:

MI	char(7)	const;	
MF	char(10)	const	options(*omit:*nopass);
MD	varchar(2048)	const	options(*omit:*nopass);

This procedure is designed to send a *STATUS message type to the external message queue (*EXT). The external message queue is known to most as the "Job Log".

A *STATUS message sent to the job log is immediately displayed on the "Message Line" of the screen display and remains there until either another *STATUS message is sent, or the display is removed.

Module #3 (ERRSRV@03)

Module #3 contains sundry callable procedures which are mostly intended for use from within the Error Handler Service Program.

When adding functionality to this service program all internal/sundry procedures of this nature should be inserted into Module #3.

GetCSE1: Retrieve Target CSE for Error Handlers

Parameters:

MD	varchar(2048)	const	options(*omit:*nopass);

Return: Call Stack Counter (Bin 4)

This procedure, or more correctly "Function", uses the "Retrieve Call Stack" (QWVRCSTK) API to get the next appropriate message queue up the call stack based on the "Message Target" parameter received.

Because this procedure is in a separate module it is required to be defined as exportable and needs to be included in the "External Symbol Table". Therefore, in theory, it can be called from any application code which needs the function. However please NOTE that this procedure was designed for use inside the error handlers and will therefore always return a call stack counter number one greater than required if used directly from an application procedure.

PEP: Check if CSE is PEP

Parameters: None

Return: Boolean Indicator – Yes = PEP Entry

This procedure checks to see if the retrieved call stack entry is a PEP (Program Entry Point) entry or not.

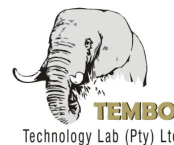
This procedure is not exported and can therefore only be used by procedures included into Module #3.

Cmd: Execute Command

Parameters:

Cmd$	varchar(4096)	const	options(*varsize);

This procedure is added to the “Error Handling” routines as it is frequently required to execute a CL command from within an RPG program. Using the ‘Copy Book’ to define the procedure it makes it easier to perform this task. No “RTV” type commands will work as this procedure cannot return the required variables.



Module #4 (ERRSRV@04)

Module #4 contains 2 procedures, both of which are related to the retrieval of different message text in different sizes. This fact that information is returned defines these as functions rather than procedures.

The following sections describe the action of each of these procedures in detail.

ERR20: Retrieve Message Text

Parameters:

MI	char(7)	const;	
MF	char(10)	const	options(*omit:*nopass);
MD	varchar(2048)	const	options(*omit:*nopass);

Return: CHAR(132)

This function returns a maximum of 132 characters of message text with the message data (MD) replacing the variables &1 etc.

This returned text is equivalent to the 1st level text displayed on a display screens message line when errors are encountered.

ERR21: Retrieve Message Help Text

Parameters: MD

MI	char(7)	const;	
MF	char(10)	const	options(*omit:*nopass);
MD	varchar(2048)	const	options(*omit:*nopass);

Return: CHAR(3000)

This function operates in the same manner as ERR20 above but returns a 3000 character field containing the substituted extended help text for the message.

Module #5 (ERRSRV@05)

Module #5 contains only 1 function which was specifically designed to simplify error handling around SQL imbedded into RPGLE code.

The following sections describe the action of this procedure in detail.

ERR50: Convert SQLCODE to Error Message

Parameters:

CAPtr	pointer	const;

Return: Boolean Indicator

This procedure is called following the execution of an SQL statement, embedded into RPG, usually FETCH.

This function performs 2 different actions;

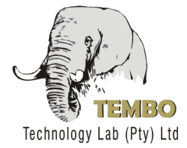
The first is to return a Boolean indicator in the following situations,

*ON if the SQLCODE is equal to 100, indicating that a record has been successfully found.

*OFF if the SQLCODE equals 0 indicating an end of file condition is encountered.

*OFF is also returned if the SQLCODE is greater than 0 (positive) indicating a non-terminal warning/information error condition.

The second action which is performed by this function occurs when the SQLCODE is less than 0 (negative). A negative numeric SQLCODE indicates a terminal error, and this function translates it into a standard pre-defined message, making it much more user friendly, then sends it as an *ESCAPE



message to the caller. This *ESCAPE message can then be monitored by the application code and dealt with accordingly.

Binding the Service Program (ERRSRV@@)

Once all modules have been compiled using the CRTRPGMOD command, they need to be bound into a service program. The method described here has proven to be the most satisfactory when binding service programs as it eliminates most of the complexities and problems which can occur.

Binder Source (ERRSRV@$)

Binder source is generated automatically by default when binding a service program. However, it is a much better solution to create a binder source member which allows you to take control of the service programs signature and define and control the "Export Symbol Table" for the service program.

The most important rule to ensure that problems with the service program are minimized is to ALWAYS add new exported procedures to the BOTTOM of the export symbol table and NEVER change the sequence or remove entries from the table once the service program is in use.

Binder source is not compiled, but saved into a source file, preferably in the same one that contains the modules for the service program.

Service Program Compiler (ERRSRV@@)

This is a simple CLLE program containing only the one CRTSRVPGM command. The purpose of this is to document which modules are included in the service program and where the binder source may be found.

In addition, if the service program requires other service programs to function, they would also be coded into this single statement using the BNDSRVPGM keyword. Do not use the BNDDIR keyword as this is usually the cause of "Duplicate Procedure" errors and other complications.

Compile this program in the normal way using CRTBNDCL and then run it to compile the service program. If you are using the compiler program, as provided, then don't forget to set the current library to the schema containing the modules and the binder source, which will also be the schema into which the *SRVPGM is created.

Final Thoughts

The final step after having successfully compiled the service program is to register it in a binding directory so that it can be found and used when compiling application code.

A small RPGILE test program (ERR_TEST) is provided in the source to illustrate the requirements for using these error handlers. Note the BNDDIR keyword in the H spec which identifies the binding directories to be examined for the export symbols required during compilation.

I always recommend that only one binding directory is used to register all service programs used by all application code and this binding directory contains ONLY service programs, no modules. As you can see, the binding directory used in this case is called "SRVPGMS".