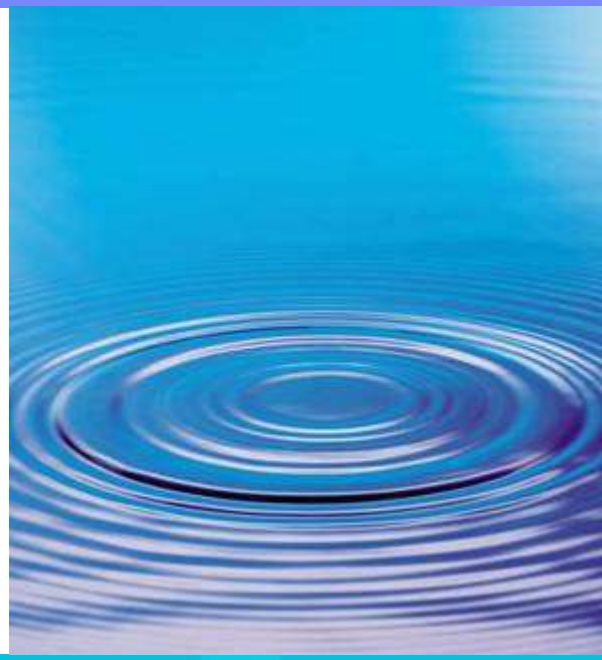
IBM Software Group

Modules, service programs, activation groups, oh my!



Barbara Morris
IBM

Agenda

- **Why use ILE features for RPG applications**
- Procedures, modules and service programs
- Activation group strategies
- Compiling and binding
- Debugging

ILE vs OPM

Integrated Language Environment (ILE)

vs

Original Program Model (OPM)

With ILE

- Easier maintenance and testing with smaller code components
- More control
- More exception-handling options

Also

- Access to many APIs including the entire C runtime library
- Better performance in general

Smaller code components (modularity)

One big program that does everything vs many small and even tiny procedures that each do one thing
(Also possible with OPM, but difficult to maintain)

- Easier to understand a small procedure
- Easier to share a small procedure across multiple applications if the procedure doesn't have additional application-specific logic in it
- Easier to test one procedure at a time
- Easier to isolate changes to a small piece of code
- Easier division of labour (junior vs senior programmers work on different types of procedures)

Smaller code components (modularity)

The main benefit of ILE: being able to have many many little procedures instead of big programs.

The Integrated Language Environment is optimized to make procedure calls very very fast.

More control

Use Activation Groups to isolate your overrides, shared-opens, and commitment control to just a portion of your job

- File overrides can be scoped to the activation group
  - One application can't interfere with another
- When an activation group ends, all files used in the activation group are closed

More exception handling options

With an OPM RPG program (or DFTACTGRP(*YES) RPG IV program)

- Either handle the exception with an error indicator, (E), *PSSR, INFSR

OR

- Inquiry message

With ILE, additional options

- Percolate the exception to the caller
- Register a procedure to run when there is an exception (CEEHDLR)
- Register another procedure to run when your procedure crashes (CEEUTX)

Percolate an exception to the caller

Say you have an application with three programs

- A calls B calls C
- A has an error indicator on the call to B
- C gets a divide by zero exception

OPM

- C gets an inquiry message about the divide by zero
- B gets an inquiry message about the failed call to C
- A's error indicator gets set on

ILE

- A's error indicator gets set on
- B and C just end immediately without showing an inquiry message

Percolate an exception to the caller

From the previous slide ...

ILE
- A's error indicator gets set on
- B and C just end immediately without showing an inquiry message

Q: What if you want to know that B and C ended suddenly?

A: You could add some code in A to log error information when the call to B fails, including the joblog which would contain the information about B and C's errors.

Or you could use an ILE cancel handler in B or C.

Cancel handlers

Say you want some processing to be done even if your program or procedure gets cancelled

Enable a cancel handler

```
CEETREC (%paddr(myCancelHandler) : ... );
... do the work of your procedure
cleanupProc();       // do your cleanup normally
return; // this disables the handler


P myCancelhandler     b

     ...
     cleanupProc();  // do the cleanup during cancel
```

Cancel handlers

Since you are calling the cleanup procedure during normal processing and from the cancel handler

- Add some checks to make sure you only do each action once

```
D cleanupInfo           ds                  qualified inz
D   tempf_deleted                   n
D   usrspc_deleted                  n

P cleanupProc       b
 /free
    if not cleanupInfo.tempf_deleted;
       ... delete the temp file
       cleanupInfo.tempf_deleted = *on;
    endif;
    ...
```

Agenda

- Why use ILE features for RPG applications
- **Procedures, modules, programs and service programs**
- Activation group strategies
- Compiling and binding
- Debugging

How to divide up your application into parts

Q: What is the optimum number of procedures, modules, service programs, binding directories?

A:

- Exactly three procedures per module
- Exactly four modules per service program
- Exactly two service programs per binding directory
- Exactly five binding directories

NOT

How to divide up your application into parts

- Unfortunately, there's no simple answer.
- Fortunately, the answer is usually inherent in the application itself.

Imagine dividing up all the objects in your house:

- One object per building
- One object per room
- Ten objects per room
- (The answer is obvious: each room gets the objects that best fit in that room – there may be some controversy over individual objects, but everyone agrees on the general strategy)

A new way of thinking

If you are accustomed to thinking about large programs that do many complex things, it can be hard to switch over to thinking about having many procedures that each do one thing

```
// do something
 … several statements
// do something else
 … several statements
dow ...;
   // do some complex thing
     … several statements
   dow ...;
      // another complex
  thing
        … several statements
      exsr reallyComplex;
   enddo;
enddo;
```

```
something();
somethingElse();
dow ...;
  doMore();
enddo;
```

Procedure doMore()

```
doSomeComplexThing();
dow …;
  doAnotherComplexThing();
  reallyComplex();
enddo;
```

A new way of thinking

Advantages of each style of coding

```
// do something
 … several statements
// do something else
 … several statements
dow ...;
   // do some complex thing
     … several statements
   dow ...;
```

- All in one place
- ... ?

```
something();
somethingElse();
dow ...;
  doMore();
enddo;
```

- Procedure names make your code self-documenting
- Flow of control is easier to see
- Can step over procedure calls in the debugger

What not to consider: call performance

Some considerations that you should not consider

Breaking up the application into many tiny procedures involves too many calls which will slow things down

- ILE is optimized for very fast calls
- The reason ILE was added to the system was to enable this type of programming

What not to consider: Contention over source code

Putting more than one procedure in a module will make it difficult for two people to work on the application

- The benefits of grouping similar procedures together outweigh the occasional contention over the source code
- If the module contains related procedures, it can be easier for one programmer to finish making all their changes, vs having to confer with another programmer over whether the changes interfere with each other

What not to consider: Arbitrary rules

Don't even try to figure out the optimum

- size of a procedure
- nesting level of code in a procedure
- number of procedures per module
- modules per service program
- service programs per binding directory
- number of binding directories

There is no optimum for any of these.

What to consider: The natural hierarchy of your app

Deciding how to divide up your code into procedures, and your procedures into modules, etc is an art, not a science.

In some cases, it is obvious: if several procedures A1, A2, A3 all need to use some subprocedure B1, put them all (A1-A3 and B1) into the same module

In other cases, it may not be obvious. Here, you use the same judgment that you already use to decide what programs go in what library etc.

What to consider: Privacy

Privacy: ILE offers four levels of privacy

1. Procedure level – variables and files only visible from a single procedure
2. Module level – variables, files and procedures visible from any procedure in the module
3. Service-program level – variables and procedures visible from any procedure in the service program
4. Everyone – variables and procedures visible from any procedure in the application

What to consider: Privacy

Each variable, file, or procedure should have the highest possible level of privacy

- The narrower the visibility, the less code that has to be considered when making changes

Example:

- You have some user-preferences that all the procedures in your application need to access

Privacy example

A possible mechanism: Keep the preferences in a file, and each module that needs them can read from that file

- What if the preferences can change during the course of the application? All the programs must be checked, to make sure they always read the file every time they need a preference
- What if you decide that each user should have their own file, or you decide to use a data area instead, or you decide to keep the preferences in a user-space? All the programs must be changed.

Privacy example

The ILE way: Create a service program to handle the preferences

- Have simple procedures that anyone can call, to retrieve or set a particular preference
- Have simple procedures that anyone can call, to cause the service program to set itself up
- Have other low-level procedures to actually manipulate the file – these procedures would be private to the service program
- If you decide to change the way you store the preferences, you only have to change **and test** the service program. Callers don't change at all.

ILE modules

ILE modules contain one or more procedures.

- Procedures that are exported from the module
- Procedures that are local to the module (not exported)

Use the EXPORT keyword on a Procedure-Begin spec to export a procedure in an RPG module

ILE programs

ILE programs consist of one or more modules

One module is designated as the “Program Entry Procedure Module” or PEP module.

The PEP module must have a “Program Entry Procedure” (called the “main procedure” in RPG)

When you call the program, the system actually calls the Program Entry Procedure within the PEP module.

Only the PEP procedure can be called from outside the program.

ILE programs

- Program PGM1
  - Module MOD1
    - Procedure P1 ← Program Entry Procedure
    - Procedure P2
    - Procedure P3
  - Module MOD2
    - Procedure P4
    - Procedure P5
  - Module MOD3
    - Procedure P6
    - Procedure P7
    - Procedure P8
    - Procedure P9

```
===> CALL PGM1
```

The system calls procedure P1 in program PGM1. P1 can then call other procedures in the program, either in module MOD1 or in the other modules.

ILE service programs

ILE service programs consist of one or more modules.

You can't call a service program directly. You can only call the procedures within the service program.

Service programs have two types of procedures:

- Exported from the service program (listed in the binder source)
  - These procedures would have had to be exported from their module
- Local to the service program
  - Might not even have been exported from their module
  - Might have been exported from the module, but not listed in the binder source, so not exported

More on binder source later …

Agenda

- Why use ILE features for RPG applications
- Procedures, modules and service programs
- **Activation group strategies**
- Compiling and binding
- Debugging

Activation groups

Activation groups give you a way to divide up your job so that each application or part of an application is independent.

An activation group gets created automatically by the system when a program or service program that uses that activation group is called.

An activation group ends

- when it is reclaimed (RCLACTGRP command)
- when the only program running in the activation group crashes
- when the job ends.

What happens when an activation group ends

1. Any files opened in the activation group get closed
2. The static storage used by the programs and service programs gets released
3. Allocated storage (%ALLOC) gets released
4. Activation-scoped overrides end
5. Activation-scoped commitment-control ends

Activation groups can be frustrating

A common symptom of getting activation groups wrong is having your CL do some overrides, and having your RPG programs not picking up the overrides

- Caused by the CL not being in the same activation group as the RPG

One solution

- Have the CL issue job-level overrides, remembering that they have to be deleted explicitly

Better solution

- Have the CL run in the same activation group as the RPG

How to have CL and RPG run in the same actgrp

1. Have both the CL and RPG run in the same named activation group
2. Have the CL run in a named activation group, and have the RPG run in the same activation group as its caller (*CALLER)
3. Have the CL run in a new activation group every time it is called (*NEW), and have the RPG run in *CALLER

With 1 and 2, the activation group remains active until it is reclaimed (RCLACTGRP command)

- 2 is easier to manage if you want to change the name of the activation group

With 3, the activation group ends when the program ends

How to have CL and RPG run in the same actgrp

Which way is best?

- Many people recommend using *NEW for the top level program in the application and *CALLER for everything else. The *NEW activation group is automatically reclaimed, so everything gets cleaned up when the top-level program ends.
- Others recommend using a named activation group, either the same one for everything (say your company name) or a name for the top-level program and *CALLER for everything else. That lets you control when or if the activation group is reclaimed.

Neither answer is "right".

The "default activation group"

Just a warning. "Default activation group" can mean two completely different things.

1. The "OPM compatibility mode" for RPG and CL where an ILE program behaves like an OPM program through the use of DFTACTGRP(*YES).
2. The default value, "QILE", for the ACTGRP parameter of the commands. QILE is not a special activation group, it is just the activation group that you get if you aren't specific about what activation group you want.

QILE activation group

Using QILE is the worst choice you can make.

There's nothing inherently wrong with QILE, but just because it's the default, it's often the activation group being used by sloppily-written applications and programs which might interfere with your app.

- They might reclaim QILE
- They might issue overrides, or use your overrides
- If they have serious storage corruption issues, they might interfere with the static storage of your application

Agenda

- Why use ILE features for RPG applications
- Procedures, modules and service programs
- Activation group strategies
- **Compiling and binding**
- Debugging

Compiling

- Use the CRTRPGMOD command to create a module, or CRTSQLRPGI OBJTYPE(*MODULE)
- Use the CRTPGM command to create a program
- Use the CRTSRVPGM command to create a service program

If you are accustomed to using PDM option 14 to create your programs, you will need to make some adjustments. Maybe have a CL program to create each program and service program.

Warning: It's easy to fall into the trap of thinking that because it's more awkward than "14" to compile an ILE application, that it's more awkward to use ILE in general.

The benefits of using ILE outweigh the loss of simplicity in creating *PGMs.

Types of RPG modules

- Modules with a main procedure and subprocedures
  - Can be compiled with CRTRPGMOD*
  - Can possibly be compiled with CRTBNDRPG**
    - If they don't need any other modules
    - Or if you have the modules in a binding directory specified on the H spec
    - Decide if you want to use CRTBNDRPG** for these, or to be consistent and use CRTRPGMOD* + CRTPGM
    - Be aware that using CRTBNDRPG** now makes it difficult to add more modules later
- Modules with only subprocedures (NOMAIN modules)
  - Always compiled with CRTRPGMOD*

* CRTRPGMOD or CRTSQLRPGI OBJTYPE(*MODULE)
** CRTBNDRPG or CRTSQLRPGI OBJTYPE(*PGM)

Binding

Binding can occur in two ways

- Modules can be bound together into programs or service programs (called “static binding”)
- Programs can be bound to service programs (called “dynamic binding”). Also, service programs can be bound to other service programs

When you create a program or service program, you can list the modules for the program

```
===> CRTPGM PRODLIB/MYPGM MODULE(BLDLIB/MOD1 BLDLIB/MOD2
  BLDLIB/MOD3)
```

Or you can list them in a “binding directory”

```
===> CRTPGM PRODLIB/MYPGM BNDDIR(BLDLIB/MYPGM)
```

Binding directory listing modules to create a program

A binding directory is just a list, containing modules and/or service programs.

```
===> CRTBNDDIR BLDLIB/MYBNDDIR
===> ADDBNDDIRE BLDLIB/MYBNDDIR
          OBJ((BLDLIB/MOD1 *MODULE))
...

===> ADDBNDDIRE BLDLIB/MYBNDDIR
          OBJ((PRODLIB/SRVPGM1 *SRVPGM))
```

Binding directory has either modules or srvpgms

A binding directory containing modules should just be used (if at all) to list the modules that go into a program (or service program). Such a binding directory could also list the service programs needed for that program, or another binding directory could be used.

A binding directory containing service programs can be used to list the service programs that might be needed by any program or service program in your application

Binding directory has either modules or srvpgms

Rules to live by:

1. Never allow any one module to be part of more than one program or service program

2. A binding directory containing modules should be used to build only one object (pgm or srvpgm)

Specifying the service programs when creating a pgm

- Using a binding directory
  - Using the BNDDIR keyword on the H spec of your RPG modules
  - Using the BNDDIR keyword on the CRTPGM or CRTBNDRPG command
  - Or both
- Using the BNDSRVPGM keyword on the CRTPGM (or CRTSRVPGM) command
- Or both

```
===> CRTPGM PRODLIB/MYPGM
          BNDDIR(BLDLIB/MYBNDDIR)
          BNDSRVPGM(BLDLIB/SRVPGM1 BLDLIB/SRVPGM2)
```

Specifying the service programs when creating a pgm

```
===> CRTPGM PRODLIB/MYPGM
           BNDDIR(BLDLIB/MYBNDDIR)
           BNDSRVPGM(BLDLIB/SRVPGM1 BLDLIB/SRVPGM2)
```

If the system finds that one of the modules for the program wants to call procedure “proc1”, it will search for that procedure in the modules or service programs listed in the MODULE, BNDSRVPGM or BNDDIR parameters of the command.

Binder source

Binder source is used when creating a service program. It lists the procedures that should be exported from the service program.

```
/*====================================================*/
/* Even though the list might seem to have some order,     */
/* new exports MUST be added at the end.                   */
/* !!! The order MUST NEVER CHANGE !!!                     */
/*====================================================*/
STRPGMEXP PGMLVL( *CURRENT ) SIGNATURE('MYSIGNATURE')
  EXPORT SYMBOL( "doSomething"          )  /*  1  do not change */
  EXPORT SYMBOL( "doSomethingElse"      )  /*  2  do not change */
  EXPORT SYMBOL( "somethingComplex"     )  /*  3  do not change */
ENDPGMEXP
```

I recommend adding a comment at the beginning about adding new exports at the end, as well as putting numbers in comments for each export, to reinforce the idea that the order must not change.

Service program signature

Every service program has a signature. You can see it using DSPSRVPGM. For example, QSYS/QRNXIE:

```
Display Service Program Information
...
Current export signature . . . . :   D8D9D5E7C9C5404040404
 40404040
```

The signature is shown as hex characters, but you can see it as ordinary characters using F11:

```
Display Service Program Information
...
Current export signature . . . . :   QRNXIE
```

You set the signature using the SIGNATURE parameter of the STRPGMEXP command (see previous page)

Service program signature should never change

Ideally, the signature should never change.

Once a program is bound to a service program with a particular signature, the system requires that the service program have that same signature at runtime.

If you really need to force all your programs to have to be recreated, you can change the signature on your service program. But that is a drastic step, especially if there is a possibility that you might miss one program and have it fail long after you thought you were done with that change.

Service program exports are by number, not by name

Once your program has been bound to a service program SRV1 to call procedure “proc1”, the call is not really setup to call “proc1”, it is setup to call procedure number X within SRV1. The number is assigned from the procedure’s order in your binder source. If you move “proc1” to a different place in the export list, existing programs will not be able to call “proc1” successfully.

That is why you must never change the order of your exports.

Service program exports are by number, not by name

Example:

1. Say you originally create SRV1 with exports P1, P2, P3.
2. Your program PGM1 calls P3 in SRV1.
3. You change your binder source so that SRV1 now exports P1, P2, P3A, and re-create SRV1 (but not PGM1).
4. Program PGM1 **thinks** it is calling P3, but it really is calling P3A because that is export #3. (Maybe ok, if P3A is a replacement for P3)
5. You change your binder source so that SRV1 now exports P1, P2, P2A, P3 and re-create SRV1 (but not PGM1).
6. Program PGM1 now calls P2A when it thinks it is calling P3. (Almost certainly not ok.)

Quotes or not within binder source

For the binder source EXPORT command, you can use quotes on the name, or not.

```
EXPORT SYMBOL( "doSomething"          )

EXPORT SYMBOL( doSomething            )
```

If you don't use quotes, the system will upper-case the name, so these two are equivalent

```
EXPORT SYMBOL( doSomething            )
EXPORT SYMBOL( DOSOMETHING            )
```

Quotes or not within binder source

Q: Which should you use?
A: Code your binder EXPORT command the same way as you code your EXTPROC keyword.

```
D myProc1         pr
```

- No EXTPROC, so no quotes, EXPORT(myProc1)

```
D proc2           pr                EXTPROC('myProc2')
```

- EXTPROC, so use quotes, with the name exactly the same as the EXTPROC parameter, EXPORT('myProc2')

You can use single or double quotes, but the system seems to like double quotes better. (It gives a warning if you use single quotes.)

Similarly, call from CL with or without quotes

If you call your RPG procedures from a CL module, you can use the same rules to decide whether to use quotes for the procedure name in the CL.

```
D myProc1          pr
```

- No EXTPROC, so no quotes, CALLPRC myProc1

```
D proc2            pr                  EXTPROC('myProc2')
```

- EXTPROC, so use quotes, with the name exactly the same as the EXTPROC parameter, CALLPRC 'myProc2'

(Use single quotes in CL.)

Agenda

- Why use ILE features for RPG applications
- Procedures, modules and service programs
- Activation group strategies
- Compiling and binding
- **Debugging**

Debug using RDP

- Compiling within RDP will automatically compile with useful debug views.
- Set a service entry break point on your program or service program to start the debugger.
- It will show your source in the editor.
  - Double click at the left edge of a line to add a breakpoint.
- Call your program, or some other program that will cause your program or service program to get called

Debug if you are not using RDP

- Compile with a DBGVIEW value other than *STMT to use the debugger.
- Use STRDBG to start the debugger.

```
===> strdbg mypgm
===> strdbg srvpgm(mysrvpgm)
```

- It will show the “Display Module Source” screen.
  - Use F6 to set break points.
  - Use F14 to show the source for other modules, or to add other programs
- Exit out of the “Display Module Source” screen with F12
- Call your program, or some other program that will cause your program or service program to get called

Debugging ILE programs and service programs

Within the debugger you can

- Step to the next statement
- Step into a called program or procedure
- Step over a program or procedure call
- Step out of a procedure
- Display or change the value of variables
- Set a breakpoint conditional on the value of a variable
- RDP only: Monitor the values of several variables

Additional reading

- The **ILE Concepts** manual
  - It is very readable
  - Don't try to read the whole thing at once
  - Read it together with other people and discuss the concepts
- "Code Complete" by Steve McConnell. You will find inspiration and general guidance on how to write a maintainable application.
  - Not RPG specific or even IBM-I related, but the concepts apply universally
  - Don't worry if you don't have time to read the whole thing. Even a few chapters will open your eyes
- Forums:
  - The midrange.com mailing lists, especially the RPG400-L list
  - The RPG Café forum
  - The systeminetwork.com RPG forum
  - The CODE400.com forum