Použití ILE v jazyku RPG

Vladimír Župka

Obsah

Úvod

Tento kurs je určen programátorům znalým jazyka RPG IV (přesněji ILE RPG), kteří potřebují získat důkladnější znalosti o konceptu ILE.

Zkratka ILE znamená *Integrated Language Environment*, tj. integrované jazykové prostředí. Hlavní záměry konceptu ILE jsou zhruba následující:

- usnadnit vzájemné propojení programů psaných v různých jazycích,
- zvýšit schopnost opětovného využití programů (formou spojování modulů) a tím také zlepšit jejich údržbu,
- zvýšit výkonnost programů, zvláště jejich vzájemného volání,
- umožnit oddělené zpracování aplikací formou aktivačních skupin (tj. "podúloh" v rámci úloh v OS/400).

Jazyk RPG IV se také funkčními schopnostmi přibližuje jazyku C v tom smyslu, že umožňuje použít *ukazatele* (pointers) a *podprocedury*. Tak je např. možné používat systémové programy (API) převzaté ze systému Unix, které těchto prostředků využívají a které jsou v hojném počtu zastoupeny v systému OS/400. K tomu je ovšem výhodné znát alespoň trochu jazyk C, z něhož se přebírají a transformují popisy parametrů.

Kromě jazyka RPG mohou v prostředí ILE pracovat také jazyky CL, Cobol, C a C++.

Tento materiál odpovídá verzi 7.3 jazyka i systému.

Ukazatele a bázované proměnné

Ukazatel (pointer) je paměťové místo dlouhé 16 bajtů a umístěné ve virtuální paměti na adrese, která je násobkem 16. Jeho obsah je v podstatě virtuální adresa ukazující na místo, kde leží nějaká paměťová oblast. Tato paměťová oblast je např. obyčejná proměnná, vektor či datová struktura, nebo dokonce paměť obstaraná při výpočtu z operačního systému (dynamická paměť).

Ukazatele se často používají v novějších systémových funkcích (API), zejména funkcích převzatých z operačních systémů typu Unix. Pomocí ukazatele je daná paměťová oblast adresována nepřímo. Změnou obsahu ukazatele můžeme adresovat jiné místo paměti. S ukazateli úzce souvisí klíčové slovo BASED spolu s vestavěnými funkcemi %ADDR, %ALLOC, %DEALLOC a %REALLOC.

Klíčové slovo **BASED(*ukazatel*)** přiděluje proměnné ukazatel, který ukazuje na určitou paměťovou oblast; říkáme, že proměnná je "bázovaná" ukazatelem. Adresu pro ukazatel lze funkcí %ADDR.

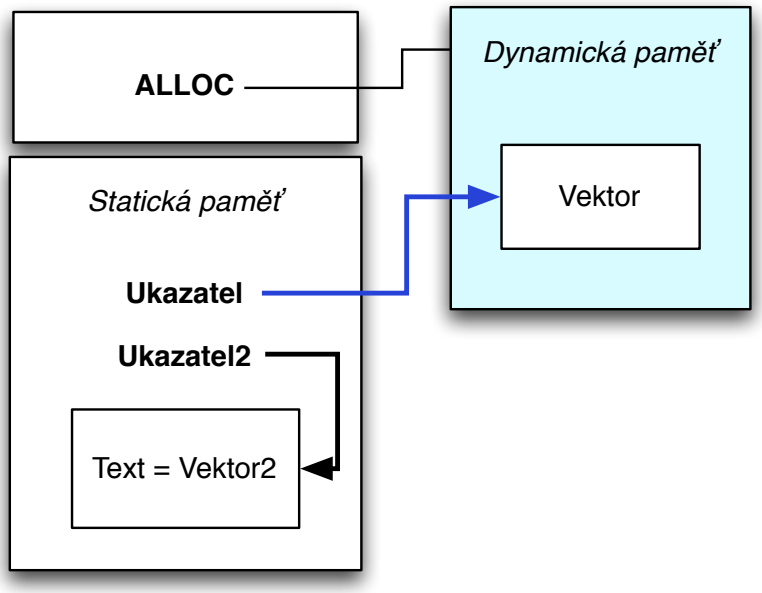
Funkce **%ADDR(*proměnná*)** poskytuje adresu proměnné v programu. Tuto adresu přidělíme zvolenému ukazateli, třeba přiřazovacím příkazem nebo funkcí INZ v definici dat.

Funkce **%ALLOC** obstará (alokuje) paměť požadované délky z volné virtuální paměti (space) a její adresu uloží do ukazatele. Kontroluje také, zda žádaný objem paměti je vůbec k dispozici a případně způsobí chybový stav.

Funkce %DEALLOC ruší obstaranou paměť a funkce %REALLOC mění velikost obstarané paměti.

Adresování bázovaných proměnných

Ukazatel (*pointer*) je proměnná, jejíž typ se označuje slovem POINTER. V následujícím příkladu je ilustrován způsob adresace proměnných pomocí ukazatelů a způsob, jakým se s takovými proměnnými pracuje.



V programu jsou definovány proměnné *Vektor* a *Vektor2* bázované ukazateli *Ukazatel* a *Ukazatel2*. Zatímco hodnota ukazatele Ukazatel2 (tj. adresa proměnné Vektor2) bude známa již po kompilaci, hodnotu ukazatele Ukazatel získáváme teprve při výpočtu funkcí %ALLOC. Adrese proměnné se někdy říká báze. Je-li tedy známa báze proměnné, lze s ní pracovat obvyklým způsobem.

Program A07 - ukazatele

```
**FREE
dcl-c PocetPolozek     100;
dcl-s Text             char(800);

// Proměnná Vektor je "bázovaná" ukazatelem Ukazatel, tzn. že její
// umístění v paměti bude určeno teprve, až bude známa hodnota ukazatele:
dcl-s Vektor           packed(5: 0) based(Ukazatel) dim(PocetPolozek);
dcl-s Vektor2          char(8) based(Ukazatel2) dim(PocetPolozek);
dcl-s Delka            int(10);

// Ukazatel (pointer), který se teprve musí naplnit;
// zpočátku má hodnotu *NULL (i kdyby nebyla výslovně uvedena):
dcl-s Ukazatel         pointer inz(*null);

// Ukazatel2 se nastaví na adresu proměnné Text. Tím se určí poloha
// proměnné Vektor2 - překrývá proměnnou Text:
dcl-s Ukazatel2        pointer inz(%addr(Text));
dcl-s Idx              packed(8);

Delka = %size(Vektor:*all);

// Získat prostor zadané délky z operačního systému a umístit
// jeho adresu do ukazatele Ukazatel:
Ukazatel = %alloc(Delka);

// Teď se už může proměnná Vektor používat normálně, protože
// má přidělenou paměť. Proměnná Vektor2 má také adresu (stejnou jako Text).
// Celý Vektor naplníme čísly 10001, 10002, ...,
// Celý Vektor2 naplníme hvězdičkami a vpravo do každé položky dosadíme
// její pořadové číslo.

Vektor2 = *all'*';
for Idx = 1 to PocetPolozek;
    Vektor(Idx) = 10000 + Idx;
    Vektor2(Idx) = %editc(Idx: 'X');
endfor;

dump(a) 'výpis paměti';
*inlr = *on;
```

Operace s ukazateli

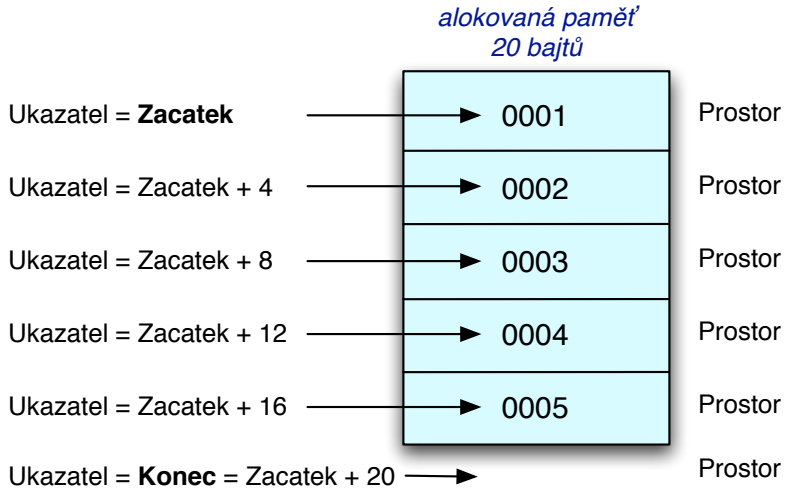
S ukazateli lze provádět *operace*:

- k ukazateli přičíst celé číslo,
- od ukazatele odečíst celé číslo,
- vypočítat rozdíl dvou ukazatelů,
- porovnat velikost ukazatelů.

V následujícím příkladu je ilustrován způsob, jak lze manipulovat s ukazateli a měnit tak obsah bázovaných proměnných.

Program A08 – operace s ukazateli

```
**FREE
dcl-s Prostor   char(20) based(Ukazatel);
dcl-c Delka     const(%size(Prostor));
dcl-s Zacatek   pointer;
dcl-s Konec     pointer;
dcl-s Ukazatel  pointer;
dcl-s Idx       zoned(4: 0) inz(1);

// Vyhradím prostor. Je-li příliš velký, končím.
Zacatek = %alloc(Delka);

Konec = Zacatek + Delka;
Ukazatel = Zacatek;
// Cyklus naplní prostor čísly 0001 až 0005
dow Ukazatel < Konec;
    Prostor = %editc(Idx: 'X');
    Idx += 1;
    Ukazatel = Ukazatel + %size(Idx);
enddo;

// Nakonec vrátím ukazatel na začátek prostoru pro DUMP
Ukazatel = Zacatek;

dump(a) 'výpis paměti';
*inlr = *on;
```

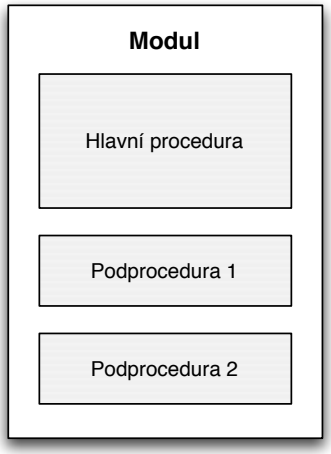


Proměnná *Prostor* má pokaždé jinou adresu podle nastavení ukazatele *Ukazatel*. Ukazatel Konec ukazuje na první bajt za koncem alokované oblasti, tj. na adresu *Zacatek + 20*.

Procedury

Program RPG IV je strukturován více než RPG III. Zdrojový člen je základem *modulu*, který z něj vzniká kompilací. Člení se na hlavní proceduru (main procedure), a na podprocedury (subprocedures).

Struktura programu - modulu




```
     // Hlavní procedura – hlavní program
     {ctl-opt MAIN;   // lineární hlavní procedura}
     {ctl-opt NOMAIN; // bez RPG cyklu}
     Popisy souborů - globální data

     DCL-PR Proc1;      Prototyp procedury
     ...                   definice vracené hodnoty a parametrů
     END-PR;
     {...               další prototypy procedur}

     DCL-PI Hprogram;   Interface hlavního programu
     ...                Parametry hlavního programu
     END-PI;
     DCL-S ...;         Globální data
     ...                Výpočty v hlavním programu + podprogramy

     // Podprocedura 1:
     DCL-PROC Proc1;       Začátek procedury
     DCL-PI;               Interface procedury
        ...                definice vracené hodnoty a parametrů
     END-PI;
     DCL-S ...;            Lokální definice dat
     ...                   Výpočty v proceduře + podprogramy
     END-PROC Proc1;       Konec procedury
     ...                   další procedury

     // Přikompilované vektory a tabulky:
**CTDATA
...
```

Modul s RPG cyklem – hlavní procedura

Hlavní procedura představuje dřívější *program* (v RPG III). Nazývá se také *hlavní program*. Může použít definici parametrů volání (rozhraní programu DCL-PI), které umožňuje volat hlavní program příkazem CALLP s parametry. Příkazem CALL či CALLB lze volat jen program bez definice parametrů (ve volném formátu není možný příkaz *ENTRY PLIST).

Modul bez RPG cyklu (lineární modul)

Jestliže do řídicího příkazu CTL-OPT zapíšeme klíčové slovo MAIN nebo NOMAIN, nelze v proceduře používat RPG cyklus. Procedura v tomto případě neobsahuje vstupní bod programu a není schopna spuštění.

- Popisy souborů a dat lze definovat jako globální před popisem podprocedur nebo lokální uvnitř podprocedur.
- Výpočetní příkazy lze zapsat jen uvnitř podprocedur.
- Při prvním volání první podprocedury se inicializují globální proměnné a otevřou se globální soubory.
- Při každém volání podprocedury se vytvoří lokální proměnné a otevřou se lokální soubory.
- Při ukončení podprocedury se smažou lokální proměnné a lokální soubory. Nesmažou se globální proměnné a nezavřou globální soubory. Indikátor LR nefunguje jako v cyklu.

NOMAIN modul

```
CTL-OPT  NOMAIN
```

Nemá žádnou hlavní proceduru. Jeho podprocedury lze volat z jiných jeho podprocedur a z jiných objektů (programů, modulů, servisních programů).

MAIN modul

```
CTL-OPT  MAIN (PGM1)
```

Jedna procedura je označena jako hlavní, zvaná lineární hlavní procedura a volá se dynamicky jako program. Volá se příkazem CALLP s parametry, obsahuje-li definici rozhraní, nebo příkazem CALLP, CALL či CALLB bez parametrů. Formálně se zapisuje jako podprocedura

```
DCL-PROC PGM1;
...
END-PROC;
```

Ostatní podprocedury modulu lze volat z hlavní procedury, z jiných jeho podprocedur a jiných objektů (programů, modulů, servisních programů).

Podprocedury

Podprocedury představují samostatné části programu (modulu). Podprocedury lze vyvolávat z hlavního programu a z ostatních podprocedur. Podproceduru lze volat také samu ze sebe, přímo nebo nepřímo - *rekursivně*. Podprocedury se zapisují (definují) za hlavním programem (ale před přikompilovanými vektory a tabulkami). K definici podprocedury slouží příkazy DCL-PROC (začátek) a END-PROC (konec). Podprocedury se často nazývají prostě *procedury*.

Volání podprocedur

Podprocedura se vyvolává dvěma způsoby:

- příkazem *CALLP s parametry*, nevrací-li procedura hodnotu nebo když nás vracená hodnota nezajímá, operační znak CALLP nemusí být zapsán, nepotřebuje-li modifikátor (E M R).
- ve výrazu svým *jménem s parametry*, vrací-li procedura hodnotu (pak se nazývá *funkce*); volání je stejné jako volání vestavěné funkce BIF až na to, že chybí úvodní znak %.

Popis prototypu a popis rozhraní

Pro každou podproceduru musí být zapsán *popis prototypu* s úvodním příkazem **DCL-PR** a koncovým příkazem **END-PR**, a to *tam, odkud se procedura vyvolává* (v hlavním programu nebo ve volající proceduře).

V definici podprocedury musí být zapsán odpovídající *popis rozhraní* (*procedure interface*) s úvodním příkazem **DCL-PI** a koncovým příkazem **END-PI**. Popis rozhraní musí zopakovat stejné složení parametrů (a návratové hodnoty) jako popis prototypu.

Popis prototypu musí být zapsán i tehdy, když podprocedura volaná příkazem CALLP nepracuje s žádnými parametry (a případně ani nevrací hodnotu). V tom případě ale nemusí být v definici procedury zadán popis rozhraní.

Správnost sestavy parametrů se kontroluje již při kompilaci, aby se nemusela kontrolovat při výpočtu. Kontrola je založena na tom, že parametry se vlastně popisují dvakrát, jednou v popisu rozhraní a jednou v prototypu.

Popisy prototypu se často umísťují do zvláštního zdrojového členu, který se začlení do kompilace příkazem /INCLUDE nebo /COPY. V takovém členu může být zadáno více prototypových popisů než je v programu podprocedur, tzn., že do členu lze připravit prototypy pro budoucí podprocedury nebo pro podprocedury definované v jiných zdrojových členech (modulech).

Příklady podprocedur

Zjistíme, zda 5znakový řetězec (parametr funkce) obsahuje znak větší než 5. Jestliže ano, hodnota funkce bude ’1’, jestliže ne, hodnota bude ’0’. Hlavní program jen naplní indikátor 25 hodnotou funkce Proc1. Kompilace CRTBNDRPG musí proběhnout s parametrem **DFTACTGRP(*NO)** nebo s parametrem téhož znění v řídicím příkazu CTL-OPT.

Program A09 – s deklarací a voláním funkce

```
**free
// Hlavní program
// --------------
ctl-opt dftactgrp(*no);

// Globální data
dcl-s Retezec char(5) inz('11191');

// Prototyp procedury Proc1
// Procedura vrací jednoznakovou hodnotu (pro indikátor)
// a definuje 5znakový parametr předávaný hodnotou:
dcl-pr Proc1 char(1);  // návratová hodnota
    Parametr char(5) value;  // parametr
end-pr;
// Hlavní výpočet - naplní hodnotu indikátoru hodnotou funkce:
// ---------------
*in25 = Proc1(Retezec); // Volání funkce Proc1
dsply 'zastavení';
dump(a) 'hlavní prog.';
*inlr = *on;

// Podprocedura Proc1
// ------------------
DCL-PROC Proc1;
// Rozhraní (interface) procedury - musí se shodovat s prototypem
dcl-pi Proc1 char(1);
    DCL-PARM Parametr char(5) value;  // symbol DCL-PARM je nepovinný
end-pi;
// Lokální data
dcl-s Vekt   char(1)       dim(%len(Parametr));
dcl-s Do     packed(4: 0);
dcl-s index  packed(4);

// Tělo podprocedury
// Přesun parametru do vektoru a vypnutí indikátoru 12:
for index = 1 to 5;
    Vekt(index) = %subst(Parametr: index: 1);
endfor;
*in12 = *off;
Do = %Len(Parametr);
//Hledáme znak větší než '5' v parametru (cyklus):
for index = 1 to Do;
    if Vekt(index) > '5';
        // Když se našel, zapneme indikátor 12
        *in12 = *on;
    endif;
endfor;

dump(a) 'Proc1';
// Navracená hodnota je indikátor 12 (vypnutý nebo zapnutý)
return *in12;
END-PROC Proc1;
```

Program A09FAKT – faktoriál s rekurzí

Podprocedury lze volat rekursivně, následuje ukázka výpočtu faktoriálu čísla 7. Kompilace CRTBNDRPG musí proběhnout s parametrem **DFTACTGRP(*NO)** nebo s parametrem téhož znění v řídicím příkazu CTL-OPT.

```
ctl-opt dftactgrp(*no);

dcl-s NFaktorial      packed(15);
dcl-s N               packed(15) inz(7);

// Prototyp funkce NFakt
// Funkce vrací hodnotu N faktoriál
// a definuje číselný parametr N předávaný hodnotou:
dcl-pr Fakt  packed(15);
    *n packed(15) value;
end-pr;

// Hlavní výpočet - naplní výsledek hodnotou N!
NFaktorial = Fakt(N);
dump(a) 'hlavní prog.'
*inlr = *on;

// Definice funkce NFakt
dcl-proc Fakt;
// Interface (musí se shodovat s prototypem):
dcl-pi Fakt  packed(15);
    N packed(15) value;
end-pi;
dump(a) 'Fakt';
if N > 2;
    N = N * Fakt(N - 1);
endif;
// Navrácená hodnota je N faktoriál
return N;
end-proc Fakt;
```

Výsledek výpočtu je NFaktorial = 5040.

Program LINEAR – program bez RPG cyklu

Kompiluje se normálně – CRTBNDRPG (volbou 14)

```
**free
CTL-OPT MAIN (LINEAR)
        dftactgrp(*no); // kvůli podproceduře PROC1
DCL-PROC LINEAR;        // deklarace procedury
    DCL-PI *N;          // rozhraní nemusí opakovat jméno procedury
        name char(21);  // parametr
    END-PI;
    dsply name;
    PROC1();            // volání procedury PROC1
END-PROC;

DCL-PROC PROC1;         // deklarace podprocedury
    dsply ('ABC');
END-PROC;
```

Volání z RPG (s prototypem):

```
// prototyp programu LINEAR
DCL-PR linear EXTPGM;   // jméno se převádí do velkých písmen
    *N char(21);
END-PR;

dcl-s name char(21);
name = 'Josef';

CALLP LINEAR(name);     // volání programu s parametrem
return;
```

Volání z CL (bez prototypu):

```
CALL PGM(LINEAR) PARM('Josef')
```

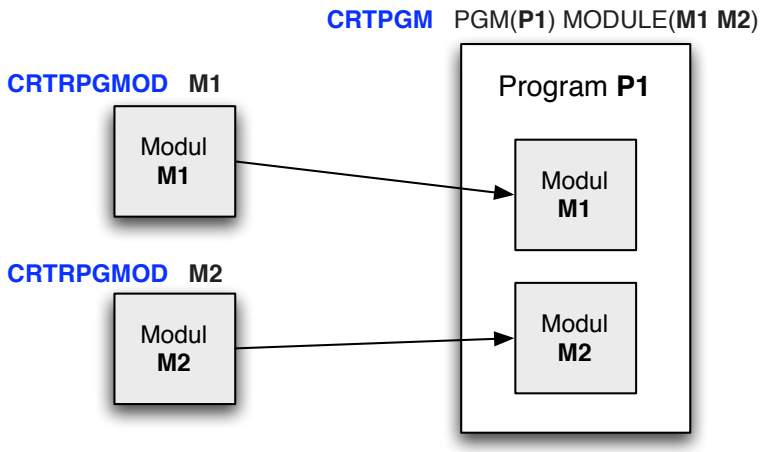
Výstup do job logu:

```
DSPLY  Josef
DSPLY  ABC
```

ILE - Integrated Language Environment

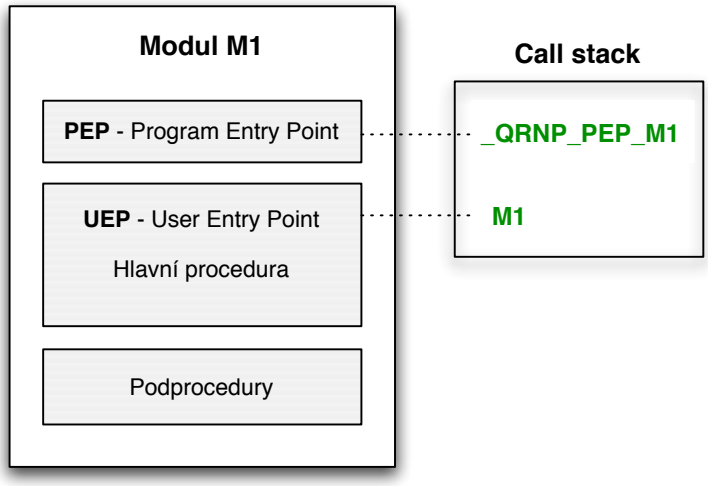
Moduly a program

ILE zavádí nový typ objektu ***MODULE** - modul. Modul je výsledkem kompilace jednoho zdrojového členu. Kompilace se spouští příkazem **CRTRPGMOD** nebo volbou 15 v PDM. Kompilací CRTRPGMOD vznikne objekt typu *MODULE, který však není schopen spuštění (příkazem CALL). Spustitelný program musí být objekt typu *PGM. Ten vznikne z jednoho nebo několika modulů tzv. *spojením kopií (bind by copy)*. Spojení modulů do programu provádí *spojovací program (binder)* příkazem **CRTPGM**.



Máme-li jen jeden modul M1 a chceme z něj vytvořit spustitelný program M1, můžeme místo dvojice příkazů CRTRPGMOD, CRTPGM použít příkaz **CRTBNDRPG** (Create Bound RPG Program - vytvořit spojený RPG program), který provádí totéž co zmíněná dvojice použitá pro jediný modul, přičemž modul je jen meziprodukt, který se nakonec zruší.

Co obsahuje objekt modulu



Modul M1 (a program M1) zde obsahuje kromě příkazů napsaných programátorem ještě úsek příkazů doplněný kompilátorem. Ten se nazývá **PEP** – Program Entry Point a slouží jako vstupní bod při spuštění programu, jestliže tento modul je vstupním modulem programu (*entry module*). První výkonný příkaz modulu se nazývá **UEP** – User Entry Point. Je to místo, na něž přejde výpočet po provedení úseku PEP. Název úseku se skládá z předpony _QRNP_PEP_ a jména modulu. V obrázku jsou názvy, které jsou vidět v zásobníku volání (call stack), zastavíme-li

program příkazem DSPLY a použijeme klávesu *System Reques*t s volbou *3. Display Current Job* a dále *11. Display call stack, if active*). Můžeme to zkusit s programem A09.

Volání programů a procedur

Volání programů

V prostředí OPM (Original Program Model) používaném v RPG III se programy - objekty typu ***PGM** - vyvolávají pouze příkazem **CALL**. Toto volání se nazývá *dynamické* nebo externí, a to proto, že vyhodnocení jména programu a získání jeho adresy probíhá až při výpočtu. Příkaz CALL nejprve vyhledá programový objekt (typu *PGM) v systému podle jména, tím získá jeho adresu a spustí jej. Součástí spouštění programu je také vyhodnocení parametrů a oprávnění, což zabírá určitý výpočetní čas. Jak víme, parametry se v příkazu CALL předávají referencí, tedy nepřímým odkazem (adresou) na proměnné ve volajícím programu.

Stejný typ volání se používá také v ILE. Z programů RPG IV lze příkazem CALL volat jak programy vzniklé z RPG III v prostředí OPM, tak programy vzniklé z RPG IV v prostředí ILE.

Programy, tedy objekty typu *PGM, však lze spouštět nejen příkazem CALL, ale i příkazem CALLP, definuje-li volající program prototyp a volaný program definuje popis rozhraní (interface). Prototyp programu vypadá stejně jako prototyp podprocedury, ale nemůže v něm být definována návratová hodnota, ani parametr předávaný hodnotou. Příkaz **CALLP** se jménem programu představuje dynamické volání stejně jako příkaz CALL.

Volání procedur

V ILE lze ovšem kromě programů vyvolávat i procedury. Příkazy pro volání procedur jsou:

- **CALLB** (Call *Bound* Procedure) - volat *připojenou* proceduru. Toto volání nepoužívá prototyp; parametry se předávají příkazy PLIST a PARM podobně jako u příkazu CALL. Příkazem CALLB lze volat i hlavní proceduru *připojeného* modulu stejného jména.
- **CALLP** (Call *Prototyped* Procedure or Program) - volat prototypovou proceduru (nebo i program - viz výše). Prototypová procedura je taková, která používá prototyp. Jméno procedury se zapisuje bez apostrofů a parametry oddělené dvojtečkou se píšou do závorky za jméno procedury. Parametry mohou mít formu *výrazu*. Symbol CALLP před jménem procedury lze vynechat, nepotřebujeme-li u něj modifikátory (E M R).
- **Jméno funkce** ve výrazu. Funkce je podprocedura, která vrací hodnotu příkazem RETURN. Parametry (argumenty) oddělené dvojtečkou se zapisují do závorky za jméno procedury. Funkce vždy používá prototyp. Parametry mohou mít formu výrazu.

Tyto příkazy se nazývají *statické* volání (static call) nebo také *spojené* volání (bound call) a lze je použít dvěma způsoby:

- *pevně*, kdy adresa volané procedury je známa již při spojování modulů,
- *ukazatelem*, kdy adresa volané procedury je předem umístěna v proměnné typu ukazatel (pointer), což musí zajistit programátor; každá volaná procedura ovšem musí být obsažena v některém z připojených modulů.

Volání jménem funkce ve výrazu

Ze dvou modulů, A10 a A11 vytvoříme program A10PGM. Modul A10 volá funkci *Proc1* definovanou v modulu A11 s jedním parametrem. Funkce vrací jednoznakovou hodnotu (`*on` nebo `*off`).

Modul **A10** – Statické volání funkce ve výrazu

```
**free
// Hlavní procedura
// ----------------
dcl-s Retezec   char(5) inz('11191');

// Volaná procedura Proc1 vrací jednoznakovou hodnotu (pro indikátor)
// a definuje 5znakový parametr předávaný hodnotou

// Prototyp procedury Proc1:
dcl-pr Proc1 char(1); // vrací znak
     Parametr char(5) value;
end-pr;

// Hlavní výpočet - naplní hodnotu indikátoru hodnotou funkce
*in25 = Proc1(Retezec); // volání funkce ve výrazu

dump(a);
*inlr = *on;
```

Modul **A11** - Funkce Proc1 volaná z modulu A10

```
**free
ctl-opt nomain;

// Začátek procedury Proc1
dcl-proc Proc1 export;

     // Interface procedury (musí se shodovat s prototypem):
     dcl-pi Proc1 char(1);
         Parametr char(5) value;
     end-pi;

     dump(a) 'Proc1';
     return Parametr < '99999';
end-proc Proc1; // Konec procedury Proc1
```

Příkazy k vytvoření modulů ze zdrojových členů A10 a A11 – kompilace klávesou F15.

```
CRTRPGMOD MODULE(*CURLIB/A10) SRCFILE(*LIBL/QRPGLESRC)
CRTRPGMOD MODULE(*CURLIB/A11) SRCFILE(*LIBL/QRPGLESRC)
```

Příkaz k vytvoření programu A10PGM z modulů A10 a A11:

```
CRTPGM PGM(A10PGM) MODULE(A10 A11) ACTGRP(QILE)
```

Výpočet začne modulem A10, protože je uveden jako první a jiný modul není zadán jako startovní.

Poznámka: Procedura musí být exportována klíčovým slovem **EXPORT** v úvodním příkazu definice procedury (DCL-PROC), jestliže se její jméno liší od jména modulu.

Volání příkazem CALLB

Ze dvou modulů, A12 a A13 vytvoříme spojením program A12PGM. Modul A12 volá modul A13 se dvěma parametry. Příkaz CALLB nemůže vracet hodnotu, je tedy ke kontrole výsledku použit druhý parametr. Parametry nemohou být předávány hodnotou, protože příkaz CALLB nepoužívá prototyp.

Poznámka: Příkaz CALLB a jeho parametry musí být napsány v pevném tvaru. Volaný program lze napsat v pevném, smíšeném nebo volném formátu.

Modul **A12** – Statické volání modulu A13

```
     dcl-s Retezec       char(20) inz(*all'?');

     // Hlavní výpočet - naplní se parametr AnoNe ( '0' / '1' )
C                    callb     'A13'
C                    parm                    Retezec
C      *in25         parm                    AnoNe             1

     dump(a) 'hlavní modul';
     *inlr = *on;
```

Modul **A13** – Modul s hlavní procedurou volanou z modulu A12

```
C      *entry        plist
C                    parm                    Parametr1        20
C                    parm      *in01         Parametr2         1
C                    eval      *in01 = (Parametr1 = *all'?')

     dump(a) 'volaný modul';
     return;
```

Alternativní zápis modulu **A13** s prototypem

```
     dcl-PI *n;
        Parametr1   char(20);
        Parametr2   ind;
     end-PI;

     Parametr2 = (Parametr1 = *all'?');

     dump(a) 'volaný modul';
     return;
```

Příkazy k vytvoření modulů ze zdrojových členů A12 a A13 – kompilace klávesou F15.

```
CRTRPGMOD MODULE(*CURLIB/A12) SRCFILE(*LIBL/QRPGLESRC)
CRTRPGMOD MODULE(*CURLIB/A13) SRCFILE(*LIBL/QRPGLESRC)
```

Příkaz k vytvoření programu A12PGM z modulů A12 a A13:

```
CRTPGM PGM(A12PGM) MODULE(A12 A13) ACTGRP(QILE)
```

Volání procedury příkazem CALLP

Ze dvou modulů, A14 a A15 vytvoříme program A14PGM. Modul A14 volá proceduru *Proc2* definovanou v modulu A15 se dvěma parametry. Příkaz CALLP nemůže vracet hodnotu, je tedy ke zjištění výsledku použit druhý parametr. Parametry lze předávat hodnotou.

Modul **A14** – Statické volání procedury Proc2 modulu A15 příkazem CALLP

```
**free
dcl-s Retezec     char(5) inz('11191');
dcl-s JedenZnak   char(1) inz(*off);

// Prototyp procedury Proc2
//  definuje 5znakový parametr předávaný hodnotou
//  a jednoznakový parametr, kde bude hodnota pro indikátor:
dcl-pr Proc2;
    Retezec       char(5) value;
    JedenZnak     char(1);
end-pr;

// Hlavní výpočet - naplní hodnotu indikátoru hodnotou funkce:
callp  Proc2 (Retezec : JedenZnak);
*IN25 = JedenZnak;
dump(a) 'hlavní modul';
*inlr = *on;
```

Modul **A15** - Procedura Proc2 volaná z modulu A14

```
**free
ctl-opt nomain; // bez základního cyklu

// Nepovinný prototyp
dcl-pr Proc2;
    Parametr1 char(5) value;
    Parametr2 char(1);
end-pr;

// Definice procedury Proc2
dcl-proc Proc2 export;
    // Interface procedury (musí se shodovat s prototypem):
    dcl-pi Proc2;
        Parametr1 char(5) value;
        Parametr2 char(1);
    end-pi;

    // Výpočet procedury
    if Parametr1 = '11191';
        Parametr2 = '1';
    endif;
    dump(a) 'Proc2';
end-proc Proc2;
```

Příkazy k vytvoření modulů ze zdrojových členů A14 a A15 – kompilace klávesou F15.

```
CRTRPGMOD MODULE(*CURLIB/A14) SRCFILE(*LIBL/QRPGLESRC)
CRTRPGMOD MODULE(*CURLIB/A15) SRCFILE(*LIBL/QRPGLESRC)
```

Příkaz k vytvoření programu A14PGM z modulů A14 a A15:

```
CRTPGM PGM(A14PGM) MODULE(A14 A15) ACTGRP(QILE)
```

Předávání parametrů procedurám

Předávání parametru odkazem - referencí

Běžně se parametry (ve volání zvané též argumenty) předávají *odkazem - referencí*. To znamená, že argument volání je proměnná, která je definována a umístěna ve volajícím programu nebo podproceduře. Do volajícího programu nebo podprocedury se nepředává hodnota proměnné, ale *ukazatel* na ni, čili adresa proměnné. Ve volání programů je tento způsob předávání parametrů jediný možný. Volaný program nebo podprocedura může změnit hodnotu proměnné, která je parametrem; v tom případě se změní i hodnota odpovídající proměnné ve volajícím programu nebo podproceduře. Při volání podprocedur však lze předávat parametry i jinými způsoby.

Předávání parametru hodnotou

Především je to předávání parametrů *hodnotou*. V prototypu (DCL-PR) i v rozhraní (DCL-PI) procedury se u příslušného parametru uvede klíčové slovo **VALUE**. Ve volaném programu pak lze hodnotu takového parametru změnit, aniž to má vliv na proměnnou (argument) ve volajícím programu. Zápis VALUE dovoluje ve volání procedury zapsat argument i jako *konstantu* nebo *výraz*, což není u volání referencí možné.

Předávání parametru jen pro čtení

Další způsob, jak předávat parametry, je *pouze pro čtení*. Vyjadřuje se zápisem klíčového slova **CONST** v prototypu i v rozhraní procedury a má podobné účinky jako VALUE. Používá se tehdy, když musíme předávat parametry referencí (jako proměnné), ale chceme zajistit, aby se ve volající proceduře nemohly změnit. Argumenty lze však zadávat také jako *konstanty* nebo jako *výrazy*, kterým se vyhradí samostatná paměť, na niž ukazuje ukazatel (reference).

Volby předávání parametrů

Zadávají se klíčovým slovem OPTIONS s parametry.

```
OPTIONS(*NOPASS *OMIT *VARSIZE *STRING *TRIM *RIGHTADJ *NULLIND)
```

Vynechávání parametrů volání

Parametry lze při volání procedury také *vynechávat*. Buď je lze v prototypu a v rozhraní označit klíčovým slovem OPTIONS(***OMIT**), a pak ve volání může být místo argumentu zapsáno klíčové slovo *OMIT. Nebo je lze označit OPTIONS(***NOPASS**), což dovoluje argument ve volání úplně vynechat. Je-li jeden z parametrů takto označen, musí být jako *NOPASS označeny také všechny parametry, které následují za ním.

Zjištění počtu převzatých parametrů volání

Počet převzatých parametrů lze ve volané proceduře zjistit vestavěnou funkcí **%PARMS**, která má hodnotu rovnou počtu předávaných parametrů. Parametry označené ve volání jako OPTIONS(*OMIT) se do tohoto počtu *zahrnují*. Parametry označené jako OPTIONS(*NOPASS) se do počtu *nezahrnují*.

Modifikace znakových parametrů

Znakové proměnné nebo vektory lze také předávat s volbou OPTIONS(***VARSIZE**). U takto označeného parametru lze ve volání zadávat argumenty *kratší* než je zadaná maximální délka parametru.

Volba OPTIONS(***STRING**), znamená, že volanému programu se předá znakový řetězec zakončený bajtem X'00', jak je obvyklé v jazyku C. Takto může být označen jen *ukazatel* na

znakový řetězec předávaný hodnotou (VALUE) nebo pouze ke čtení (CONST).

Volba OPTIONS(***TRIM**) zkrátí před předáním znakovou hodnotu o mezery z obou stran.

Volba OPTIONS(***RIGHTADJ**) zarovná před předáním znakovou hodnotu vpravo. Je dovolena jen pro znakový řetězec předávaný hodnotou (VALUE) nebo pouze ke čtení (CONST).

Volba OPTIONS(***NULLIND**) předává mapu null-bajtů pro pole schopná nabývat hodnot NULL.

Servisní programy

Kromě programů vytvářených příkazem CRTPGM (nebo CRTBNDRPG) lze v ILE vytvářet a používat také tzv. *servisní programy*. Na rozdíl od programů (objektů typu *PGM) se servisní programy nedají vyvolávat příkazem CALL nebo CALLB, ani příkazem CALLP. Slouží k tomu, aby poskytovaly procedury často používané různými programy v aplikaci. Poskytují tedy ostatním programům servis.

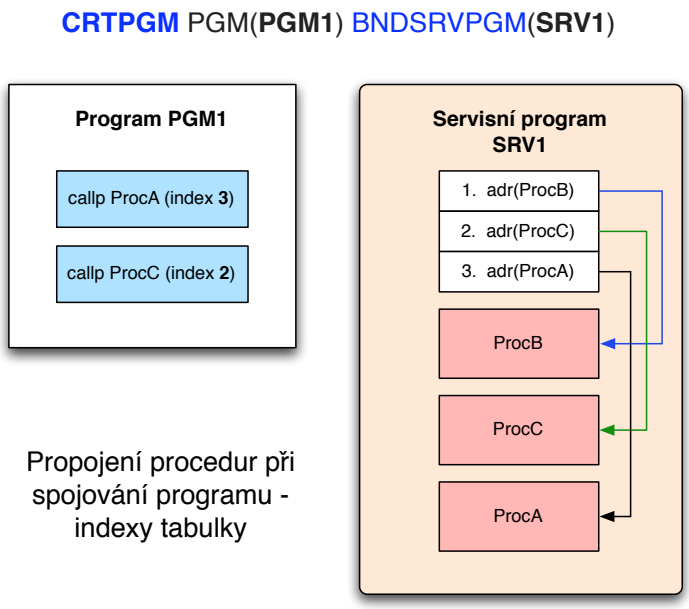
Servisní programy se sestavují z modulů, a to příkazem **CRTSRVPGM**. Příkaz CRTSRVPGM vytvoří *objekt typu* ***SRVPGM**. Každý modul v servisním programu může obsahovat několik procedur, které lze samostatně vyvolávat příkazy CALLB, CALLP nebo jménem funkce, podle toho, jak jsou procedury definovány.

Objekt servisního programu obsahuje *tabulku*, v níž jsou uvedeny adresy procedur a dat, které servisní program dává k dispozici navenek, tzv. *exportů*.

Připojení referencí

Servisní programy se používají tak, že se připojují k jiným programům. Připojují se ale jinak než moduly. V příkazu CRTPGM zapíšeme jejich jména do parametru **BNDSRVPGM**, popř. použijeme *spojovací seznam - binding directory* (viz dále), jehož jméno zapíšeme do parametru **BNDDIR**. Takové připojení modulů (prostřednictvím servisních programů) se nazývá *připojení referencí (bind by reference)*.

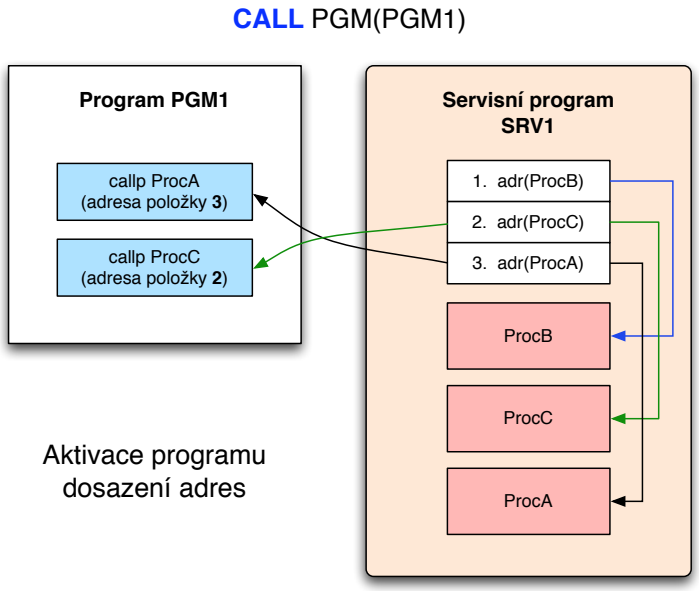
Servisní program obsahuje *tabulku adres exportů*, tj. procedur a dat, které dává k dispozici ostatním. Po připojení se servisním programem se v programu zachovají *jen indexy* na ty tabulkové položky, které program používá (nemusí používat všechny). Jde o tzv. nevyřešené reference.



Program ovšem jakožto objekt stále zůstává oddělen od objektů servisních programů. Skutečné adresy ukazující do tabulky exportů (místo indexů) se získají až při *aktivaci programu*.

Aktivace programu

Když se program spouští (dynamicky příkazem CALL nebo CALLP), probíhá jeho *aktivace*. Po kontrole oprávnění systém v první řadě vytvoří paměť pro své statické proměnné. Pak dokončí připojení servisních programů zadaných v příkazu CRTPGM. Při tom se na místo symbolických odkazů - indexů dosadí *adresy* položek ukazujících do tabulky adres exportů daného servisního programu.



Je to umožněno tím, že se současně s programem zavedou do paměti i servisní programy zadané v příkazu CRTPGM.

Pro všechny moduly každého servisního programu se vyhradí *statická paměť*, tedy proměnné definované v jeho hlavních procedurách. Týká se to také proměnných označených klíčovým slovem STATIC definovaných v podprocedurách.

V této době se také kontrolují tzv. *signatury* servisních programů (viz dále).

Signatura a spojovací zdrojový text

Servisní programy při vytváření získávají tzv. signaturu. *Signatura* je 16bajtový údaj vyjadřující určitou verzi servisního programu. Je to něco podobného jako identifikátor úrovně (level ID) u databázových souborů. Signatura představuje prostředek kontroly přístupu k servisnímu programu z programů nebo jiných servisních programů. Důvodem zavedení signatury je skutečnost, že servisní program se vytváří nezávisle na programech a na jiných servisních programech a jeho skladba se může časem změnit, což se týká zejména jeho *exportů*.

Servisní program totiž exportuje jména *procedur* nebo *dat* prostřednictvím parametru EXPORT spojovacíno programu CRTSRVPGM, kde se zadá buď hodnota *ALL nebo spojovací zdrojový text. Je-li zadána hodnota *ALL, signatura se vytvoří automaticky.

Spojovací zdrojový text (binder source) představuje zápis v jednoduchém specifikační jazyku, kterým určíme, zhruba řečeno, které procedury servisního programu může spojený program používat. Tento text slouží zejména při údržbě programů a jejich verzí. Zejména je užitečný při doplňování servisních programů o nové procedury, kdy při správném postupu odstraňuje nutnost nového spojování všech aplikačních programů, které daný servisní program používají. Tak lze zařídit, že servisní program obsahuje *tabulku několika signatur*, které odpovídají jeho různým verzím. Zdrojový typ spojovacího textu je BND a předvolený zdrojový soubor je QSRVSRC.

Text se nekompiluje! Obsahuje příkazy tohoto tvaru:

```
/*   První blok - pro současnou verzi servisního programu   */
STRPGMEXP  PGMLVL(*CURRENT) LVLCHK(*YES) SIGNATURE(*GEN)
EXPORT     SYMBOL(jméno-procedury-nebo-proměnné)
EXPORT     SYMBOL(jméno-procedury-nebo-proměnné)
EXPORT     SYMBOL(jméno-procedury-nebo-proměnné)
ENDPGMEXP
/*   Druhý blok - pro předchozí verzi servisního programu   */
STRPGMEXP  PGMLVL(*PRV) LVLCHK(*YES) SIGNATURE(*GEN)
EXPORT     SYMBOL(jméno-procedury-nebo-proměnné)
EXPORT     SYMBOL(jméno-procedury-nebo-proměnné)
ENDPGMEXP
```

Parametr **PGMLVL** specifikuje úroveň exportní signatury, současnou (*CURRENT), nebo předchozí (*PRV). Bloků s parametrem *PRV může být více.

Parametrem **LVLCHK** můžeme buď nařídit, aby se signatura při aktivaci servisního programu kontrolovala (*YES), nebo nekontrolovala (*NO).

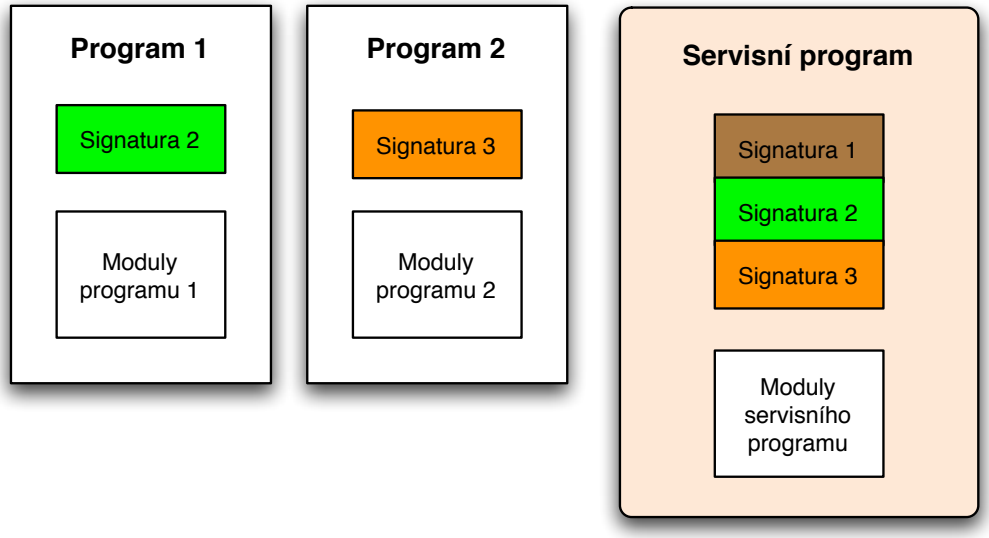
V parametru **SIGNATURE** říkáme, jakou hodnotu má signatura mít: Buď *GEN, což znamená signaturu vygenerovanou systémem, nebo hexadecimální hodnotu, až 32 znaků, nebo znakovou hodnotu, až 16 znaků. Hexadecimální hodnota se zapisuje např. takto: X'F1F2F3', znaková takto: '123'.

Je-li zadána hodnota *GEN, spojovací program vytváří signaturu na základě seznamu exportů servisního programu. Každá změna (pořadí, jména) v exportech uvedených v připojovacím zdrojovém textu způsobí změnu příslušné signatury.

V parametru SIGNATURE můžeme zadat libovolnou hodnotu. Často se volí označení verze nebo nějaká jiná dohodnutá veličina, rozdílná v různých exportních blocích (*CURRENT a *PRV) spojovacího textu.

Je-li zadána kontrola signatury **LVLCHK(*YES)** a **SIGNATURE(*GEN)**, systém při aktivaci servisního programu kontroluje jen nenulové (hexadecimálně) hodnoty a zároveň žádá, aby v bloku s parametrem PGMLVL(*CURRENT) bylo stejně nebo více exportů (příkazů EXPORT) než v bloku s parametrem PGMLVL(*PRV), aby u dosavadních bylo zachováno *pořadí* a aby nové exporty byly umístěny *na konci*. Jinak ohlásí chybu a závislé programy, popř. jiné servisní programy, je nutné znovu spojit (příkazem CRTPGM, popř. CRTSRVPGM). Pochopitelně, že systém hlásí chybu také tehdy, když zjistí v příkazu EXPORT spojovacího textu jméno, které není exportováno z příslušného servisního programu.

Při *spojování programu* (příkazem CRTPGM) se prohlíží každý připojovaný servisní program. Jeho poslední (*CURRENT) signatura zároveň s jeho jménem a jménem knihovny se uloží do programu. Později, při *aktivaci* programu (příkazem CALL) systém zkoumá, zda signatura obsažená v programu se shoduje *alespoň s jednou ze signatur* obsažených v připojovaném servisním programu. Jestliže ano, program se spustí. V opačném případě skončí chybou.

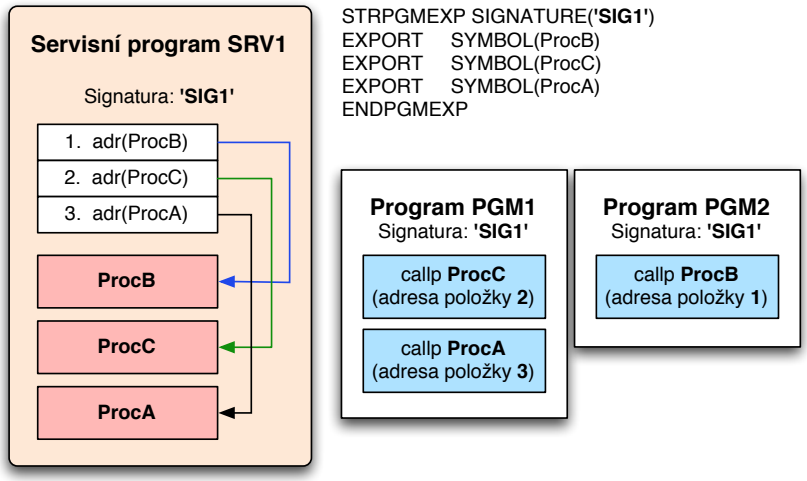


Na obrázku jsou v servisním programu je vyznačena tabulka tří signatur, z nichž každá byla vytvořena pro jinou sestavu exportů. Signaturu 2 zaznamenal program 1, když se spojoval se servisním programem a tato signatura byla právě aktuální. Obdobně aktuální signaturu 3 zaznamenal program 2. Signaturu 1 může obsahovat některý jiný nebo také žádný program.

Údržba servisního programu pomocí spojovacího textu

Pomocí spojovacího textu (binder source) lze udržovat servisní program, aniž je zapotřebí provádět jeho nové připojování k programům, které jej používají.

Při prvním vytváření servisního programu *SRV1* vznikne na základě spojovacího textu první signatura '*SIG1*', kterou si také při spojování poznamenají všechny programy, které tento servisní program používají. Program *PGM1* používá procedury *ProcA* a *ProcC*. Program *PGM2* používá proceduru *ProcB*.

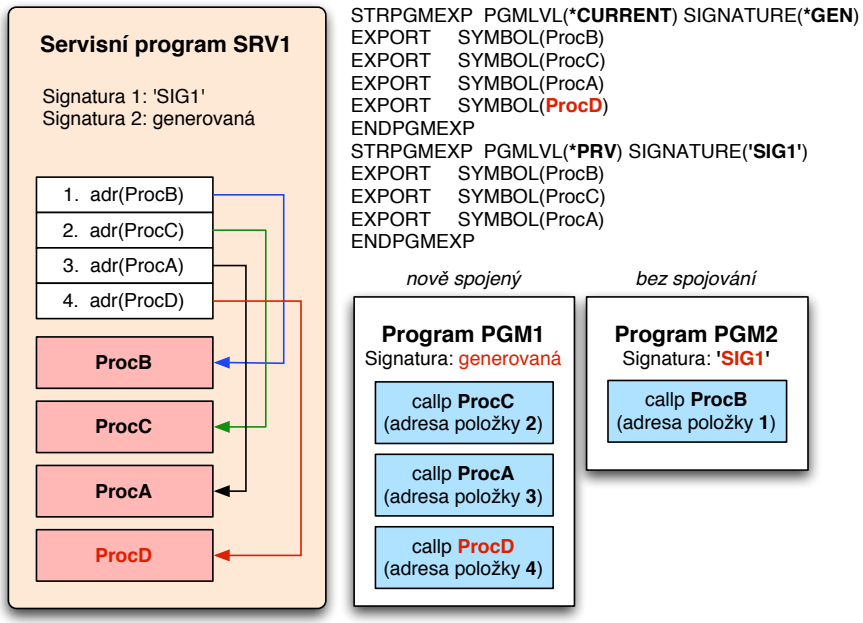


Cvičení

1. Napište modul servisního programu SRV1 s jednoduchými procedurami PROCA, PROCB, PROCC, které nepřijímají žádné parametry a nevracejí hodnotu. Každá procedura může pro kontrolu obsahovat příkaz DSPLY se svou identifikací. Zdrojový text přeložte příkazem CRTRPGMOD pod stejným jménem modulu.
2. Napište spojovací zdrojový text SRV1A (typu BND) s vyjmenovanými procedurami a signaturou SIG1 podle obrázku a uložte jej do souboru QSRVSRC.

3. Vytvořte servisní program SRV1 příkazem CRTSRVPGM, kde zadáte modul SRV1 a spojovací text (binder source) SRV1A.
4. Napište jednoduché programy PGM1 a PGM2 podle obrázku (nedělají nic jiného, než že volají procedury příkazem CALLP). Přeložte zdrojové texty příkazem CRTRPGMOD a vzniklé moduly spojte se servisním programem pomocí příkazu CRTPGM (vzniknou programy PGM1 a PGM2).
5. Spusťte program PGM1 a PGM2 a ověřte jejich funkci.
6. Vypište údaje o servisním programu příkazem DSPSRVPGM.
7. Vypište údaje o programech PGM1 a PGM2 příkazem DSPPGM.
8. Vypište údaje o modulech PGM1 a PGM2 příkazem DSPMOD.

V další fázi údržby je nutné doplnit do servisního programu další proceduru - *ProcD*, kterou potřebuje program *PGM1*. Spojovací text upravíme tak, že dosavadní text zkopírujeme, kopii umístíme na konec a označíme jako předchozí - *PRV. Původní text doplníme o proceduru ProcD a blok označíme jako *CURRENT. Doplníme ještě specifikaci signatury *GEN, což způsobí, že systém při novém vytváření servisního programu vygeneruje novou signaturu a přidá ji k dosavadní signatuře 'SIG1'. Druhá verze servisního programu bude tedy obsahovat dvě signatury. Program PGM1 musíme nově přeložit a spojit se servisním programem, protože používá novou proceduru. Při spojování se do programu PGM1 uloží nová, generovaná signatura servisního programu. Program PGM2 však můžeme nechat beze změny, protože nic nového ze servisního programu nepotřebuje.

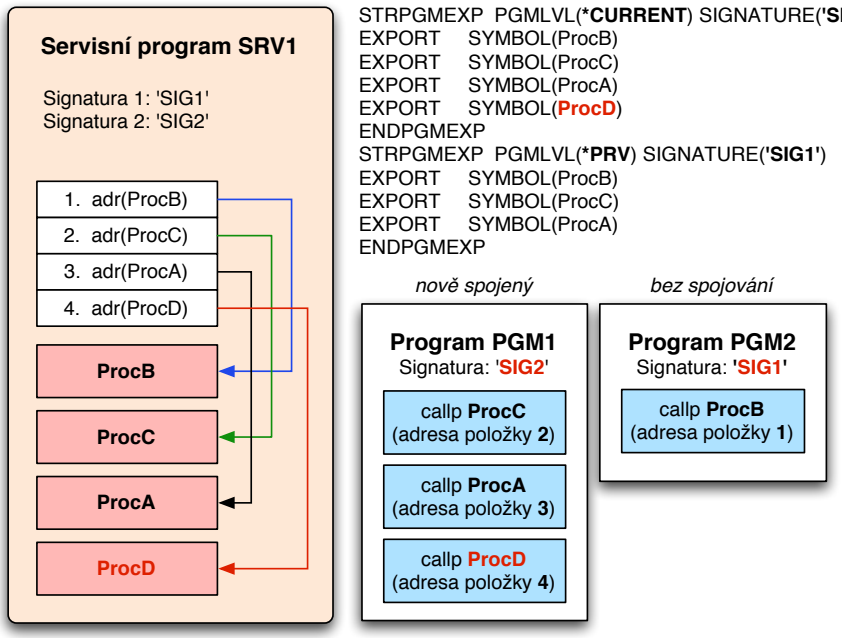


Cvičení

1. Do servisního programu SRV1 doplňte proceduru PROCD a vytvořte modul SRV1 příkazem CRTRPGMOD.
2. Napište spojovací zdrojový text SRV1B (typu BND) se dvěma bloky podle obrázku a uložte jej do souboru QSRVSRC.
3. Vytvořte novou verzi servisního programu SRV1 se spojovacím textem SRV1B.
4. Do programu PGM1 doplňte volání procedury PROCD, přeložte jej a spojte se servisním programem SRV1.
5. Ověřte funkci programů PGM1 a PGM2. Program PGM1 by měl nyní navíc volat proceduru PROCD. Program PGM2 zůstává beze změny.

6. Vypište signatury servisního programu a signatury uložené v programech.

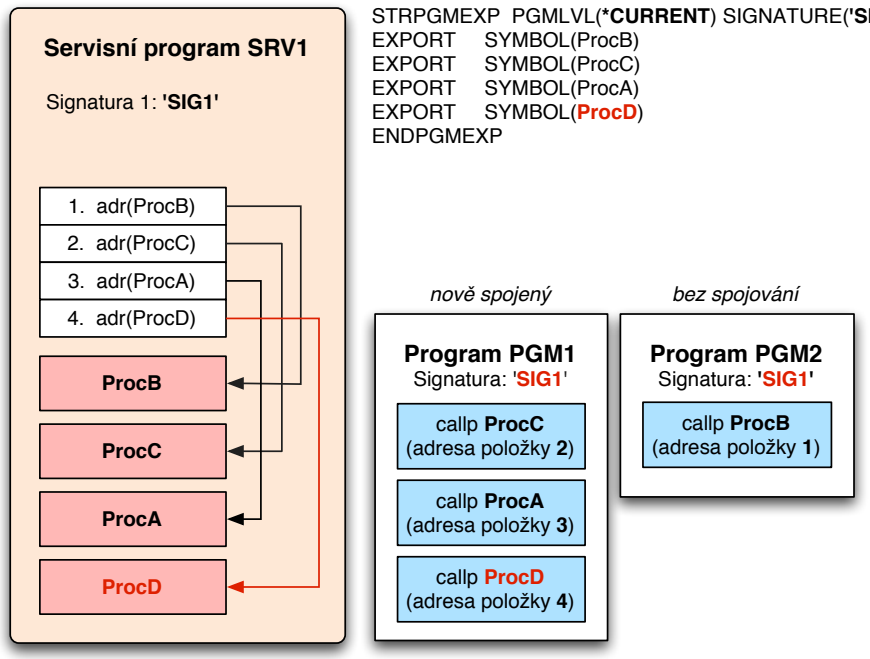
Předchozí postup můžeme ještě modifikovat, jak je patrné z dalších obrázků. Místo abychom nechali novou signaturu vygenerovat systému, určíme si ji sami, např. 'SIG2'.



Cvičení

1. Vytvořte servisní program s použitím spojovacího textu SRV1C se dvěma bloky podle obrázku.
2. Vytvořte program PGM1 novým spojením se servisním programem SRV1.
3. Ověřte funkci obou programů. Program PGM2 zůstává beze změny.

Ve spojovacím textu lze také ponechat jen jednu (původní) signaturu, přičemž na konec doplníme proceduru *ProcD*. Protože dosavadní exporty mají stále stejné místo v tabulce, staré programy mohou zůstat beze změny. Nové programy, které potřebují proceduru *ProcD*, se ovšem musí přeložit a spojit s novým servisním programem, přičemž si zapamatují starou signaturu 'SIG1'.



Cvičení

1. Vytvořte znovu servisní program SRV1 s použitím spojovacího textu SRV1D s jedním blokem podle obrázku.
2. Vytvořte program PGM1 novým spojením se servisním programem SRV1.
3. Ověřte funkci obou programů. Program PGM2 zůstává beze změny.

Export a import

Exportem obecně nazýváme *jméno procedury* nebo *jméno proměnné*, které jeden modul definuje a poskytuje proceduře v jiném modulu. Poskytující modul nemůže exportovat z programu do servisního programu nebo do jiného programu.

Importem nazýváme *jméno procedury* nebo *jméno proměnné* použité v některém modulu, které je však exportováno z jiného modulu. *Export* se také nazývá *definice* a *import* se nazývá *reference*.

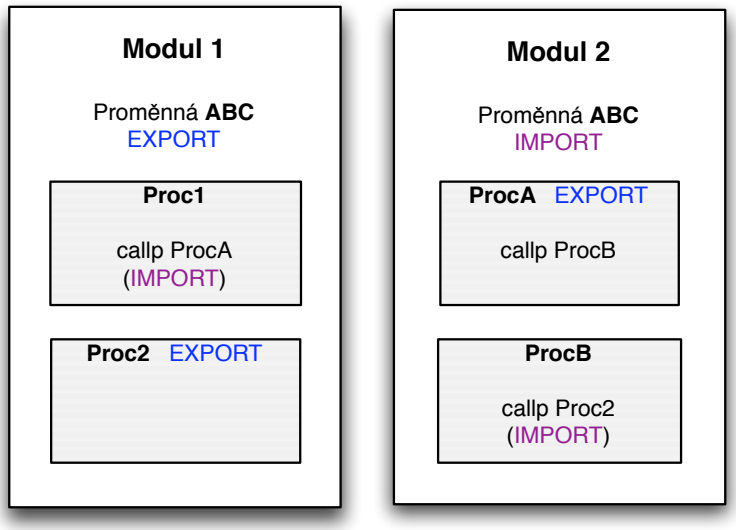
Běžný prostředek komunikace mezi programy a procedurami je předávání parametrů při jejich vzájemném volání (příkazy CALL, CALLB, CALLP nebo funkce). U volání funkce ve výrazu je to navíc hodnota funkce. *Export a import dat* představuje další prostředek komunikace mezi procedurami v různých modulech.

Export procedury vznikne zápisem klíčového slova EXPORT u definice procedury. *Export proměnné* vznikne zápisem klíčového slova EXPORT u definice proměnné. Exportujeme-li procedury nebo proměnné ze servisního programu, musíme v příkazu CRTSRVPGM zadat parametr EXPORT určující buď připojovací zdrojový text (binder source), v němž je zadán seznam exportů, nebo hodnotu *ALL, která dovolí export všech procedur a proměnných označených klíčovým slovem EXPORT.

Všechna jména procedur bez prototypu (tedy jména modulů) volaných příkazem CALLB uvnitř programu nebo servisního programu jsou automaticky k dispozici všem procedurám ve stejném programu. Nemusíme tedy u nich zapisovat klíčové slovo EXPORT. Tomu se říká *implicitní export* procedur. Netýká se to však prototypových procedur, ani jmen proměnných, které chceme exportovat. U nich musíme zapsat klíčové slovo EXPORT.

Modul 2 na obrázku exportuje proceduru *ProcA*, kterou Modul 1 importuje formou volání. Modul 1 zase exportuje proměnnou *ABC*, kterou importuje Modul 2.

Proměnná *ABC* musí být v obou modulech definována *v hlavní proceduře* naprosto shodně, jen s tím rozdílem, že v Modulu 1 má u sebe klíčové slovo EXPORT a v Modulu 2 klíčové slovo IMPORT.



Exportovat a importovat tedy můžeme

- uvnitř jednoho programu,
- uvnitř servisního programu,
- ze servisního programu do programu, k němuž je připojen,
- ze servisního programu do „vyššího“ servisního programu, k němuž je připojen.

Exportovat a importovat nelze

- z programu do připojeného servisního programu,
- mezi programy.

Příklad - export a import proměnných mezi moduly

Příklad je analogií k programu A12PGM (statické volání CALLB), jen je v něm první parametr volání - proměnná RETEZEC - přemístěn do definičního formuláře a označen jako EXPORT v modulu A16 a jako IMPORT v modulu A17. Inicializace proměnné je možná jen v exportu. U importu není povolena.

Modul **A16** - exportuje proměnnou:

```
     dcl-s Retezec    char(20) inz(*ALL'Export z A16') EXPORT;
     // Hlavní výpočet - naplní se parametr AnoNe ( '0' / '1' )
C                     CALLB     'A17'
C       *IN25         PARM                    AnoNe             1
     dump(a) 'A16';
     *inlr = *on;
```

Modul **A17** - importuje proměnnou:

```
     dcl-s Retezec   char(20) IMPORT;
C       *ENTRY        PLIST
C                     PARM      *IN01         Parametr          1
     *IN01 = (Retezec = *ALL'Export z A16');
     dump(a) 'A17';
     RETURN;
```

Vytvoření programu **A16PGM** z obou modulů:

```
CRTPGM PGM(A16PGM) MODULE(A16 A17)
```

Spojovací program (binder)

Spojovací program má za úkol spojit moduly do programu, popř. do servisního programu. Spouští se příkazem CRTPGM (Create Program) nebo CRTSRVPGM (Create Service Program).

Při vytváření programu příkazem **CRTPGM** si spojovací program vytváří tabulku či *seznam exportů* z několika zdrojů, a to *v tomto pořadí*:

- z modulů zadaných v parametru MODULE,
- ze servisních programů zadaných v parametru BNDSRVPGM,
- z modulů a servisních programů zadaných ve spojovacích seznamech parametru BNDDIR,
- z požadavků systému pro výpočetní prostředí (run-time), které generuje kompilátor RPG (jde o systémové servisní programy a spojovací seznamy).

Při vytváření servisního programu příkazem **CRTSRVPGM** spojovací program podobně vytváří tabulku či *seznam exportů* z následujících zdrojů, a to *v tomto pořadí*:

- z modulů zadaných v parametru MODULE,
- z parametru EXPORT (spojovací zdrojový text nebo *ALL),
- ze servisních programů zadaných v parametru BNDSRVPGM,
- z modulů a servisních programů zadaných ve spojovacích seznamech parametru BNDDIR,
- z požadavků systému pro výpočetní prostředí (run-time), které generuje kompilátor (jde o systémové servisní programy a spojovací seznamy).

Případným importům pak spojovací program přiděluje exporty z těchto zdrojů podle uvedeného pořadí. *Pořadí exportů je důležité*, vyskytnou-li se v některých výše uvedených zdrojích stejná jména.

Spojovací seznam (binding directory)

Spojovací seznam - binding directory - je objekt typu ***BNDDIR**. Uvádějí se v něm *jména* modulů a servisních programů, které mohou uspokojit požadavky na import programů nebo servisních programů. Seznam je nepovinný a jeho položky se uplatní jen tehdy, jestliže se na ně některá procedura výslovně odvolává, tj. jestliže je příslušné jméno použito v některém jejím příkazu. Znamená to, že v seznamu může být uvedeno i mnohem více položek (jmen), než je v daném případě k uspokojení importů potřeba, a nevadí to. V operačním systému je mnoho takových seznamů. Jeden z nich lze využít k volání unixovských funkcí (Unix API), např. k práci se soubory IFS, ke komunikaci TCP/IP aj. Některé jsou také zdrojem automatického připojení systémových procedur potřebných pro běh RPG programu (run-time).

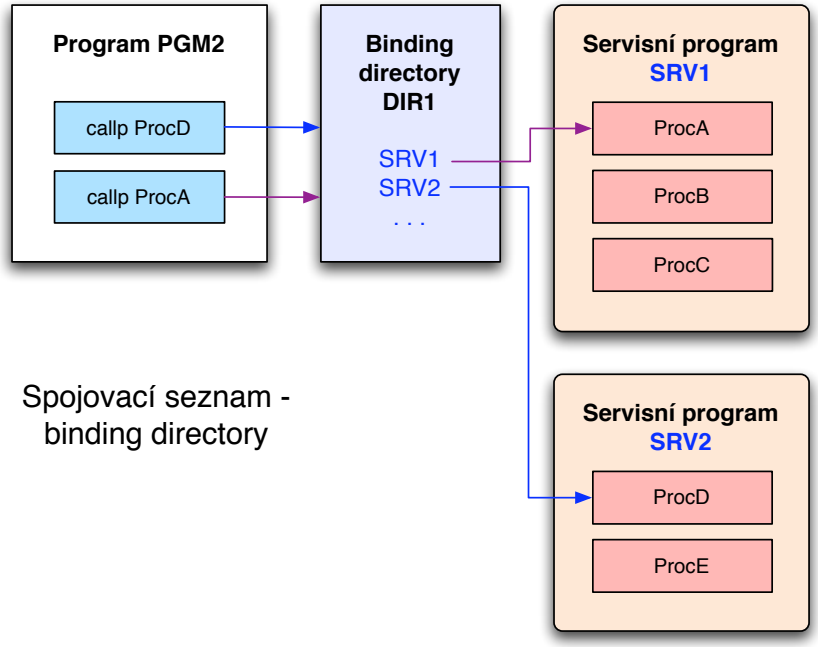
Spojovací seznam vytváříme příkazy

```
CRTBNDDIR  BNDDIR(jméno-seznamu)
ADDBNDDIRE BNDDIR(jméno-seznamu) OBJ(jméno-modulu-nebo-servisního-programu)
ADDBNDDIRE BNDDIR(jméno-seznamu) OBJ(jméno-modulu-nebo-servisního-programu)
...
```

Příkazů ADDBNDDIRE je tolik, kolik modulů a servisních programů zařazujeme do seznamu.

Jméno spojovacího seznamu pak zadáváme v příkazu CRTPGM nebo CRTSRVPGM jako hodnotu parametru BNDDIR.

CRTPGM PGM(**PGM2**) BNDDIR(**DIR1**)

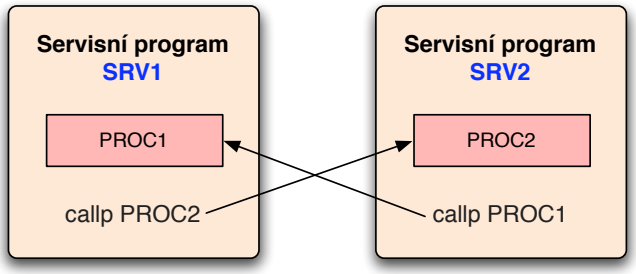


Nevyřešené reference (importy)

Při vývoji rozsáhlých aplikací je užitečné zkusit, zda některé programy fungují, i když nejsou ještě všechny procedury a funkce hotovy. Pro tento případ existuje parametr **OPTION (*UNRSLVREF)** v příkazu CRTPGM a CRTSRVPGM. Program či servisní program se vytvoří, přestože v něm zůstanou nevyřešené symboly (reference, importy). Takový program může ovšem havarovat, jestliže se nevyřešený symbol použije, např. když se vyvolá procedura, která neexistuje.

Tuto možnost lze využít také tehdy, když z nějakých důvodů dva servisní programy vzájemně používají své procedury.

CRTSRVPGM SRVPGM(SRV1) OPTION(***UNRSLVREF**)



Např. servisní program SRV1 definuje proceduru PROC1 a používá proceduru PROC2 ze servisního programu SRV2. Servisní program SRV2 naopak definuje proceduru PROC2 a používá proceduru PROC1 ze servisního programu SRV1. Tento stav se někdy nazývá vzájemná nebo kruhová reference. Obyčejné spojování by hlásilo chybu “nevyřešená reference“ v obou případech. Situaci lze vyřešit tak, že např. servisní program SRV1 vytvoříme příkazem

```
CRTSRVPGM SRVPGM(SRV1) MODULE(SRV1) EXPORT(*SRCFILE) SRCFILE(QSRVSRC) +
          SRCMBR(SRV) OPTION(*UNRSLVREF)
```

který nehlásí chybu, i když v připojovacím zdrojovém textu SRV je zapsán symbol PROC2:

```
STRPGMEXP
EXPORT SYMBOL(PROC1)
EXPORT SYMBOL(PROC2)
ENDPGMEXP
```

Teď můžeme vytvořit druhý servisní program SRV2 příkazem

```
CRTSRVPGM SRVPGM(SRV2) MODULE(SRV2) EXPORT(*SRCFILE) SRCFILE(QSRVSRC) +
          SRCMBR(SRV)
```

který se vytvoří, aniž by hlásil chybu, protože procedura PROC1 je už k dispozici (symbol PROC1 lze vyřešit). Nakonec můžeme znovu vytvořit servisní program SRV1 příkazem

```
CRTSRVPGM SRVPGM(SRV1) MODULE(SRV1) EXPORT(*SRCFILE) SRCFILE(QSRVSRC)
          SRCMBR(SRV) BNDSRVPGM(SRV2)
```

který nyní vyřeší i symbol PROC2, dříve nedostupný. Jestliže nyní vyvoláme z jednoho servisního programu proceduru druhého servisního programu, proběhne aktivace druhého servisního programu, která poněkud zdrží výpočet.

Aktualizace programů bez nového spojování

Pro případ, kdy potřebujeme z nějakého důvodu nahradit jeden nebo několik málo modulů v mnohamodulovém programu nebo servisním programu, jsou k dispozici příkazy pro jejich aktualizaci: **UPDPGM** a **UPDSRVPGM**. Příkazy se podobají příkazům pro vytváření, ale nepotřebují k činnosti všechny moduly tvořící existující objekt. Nahradí jen vyjmenované moduly. Aktualizace se používá při dodávkách programů zákazníkům, když nechceme posílat celý objekt nebo všechny moduly potřebné k jeho vytvoření, ale pošleme jen nové moduly a zákazník provede aktualizaci pomocí výše zmíněných příkazů.

Při aktualizaci může vzniknout “osamělý“ modul (nebo více modulů), a to tehdy, když nový modul obsahuje méně importů než dosavadní modul. K jejich vyřešení existoval v programu jiný modul, který pro něj poskytoval exporty. Nyní, když jeho exporty nejsou potřebné, je osamělý modul nepotřebný a zabírá v programu zbytečně místo. Osamělé moduly lze z aktualizovaného programu vynechat použitím parametru OPTION(*TRIM). Případné exporty, které osamělý program poskytoval, nadále ovšem nejsou k dispozici ani pro jiné moduly.

Obsahuje-li nový modul více importů než dosavadní a všechny se uspokojí existujícími exporty, je vše v pořádku, jinak se hlásí chyba “nevyřešené reference“, není-li ovšem zadán parametr OPTION(*UNRSLVREF).

Obsahuje-li nový modul méně exportů než dosavadní, je vše v pořádku, jestliže chybějící exporty nejsou potřeba v jiných modulech. Obsahuje-li více exportů, je výsledek ještě lepší. Při aktualizaci servisního programu je ovšem třeba dát pozor, jsou-li exporty zadány parametrem EXPORT(*ALL). Změní-li se totiž skladba exportů, změní se také signatura servisního programu.

Aktivační skupiny

Aktivační skupina (activation group) je pojem z ILE a představuje podstrukturu úlohy (job substructure). Každý ILE program se aktivuje a běží uvnitř aktivační skupiny. Aktivační skupina obsahuje všechny prostředky nutné k provozu programu. Jde o následující objekty:

- Proměnné programu (statická a automatická paměť)
- Dynamická paměť
- Dočasné prostředky pro řízení dat
- Programy pro zpracování výjimek a ukončovací procedury

V každé aktivační skupině se statická, automatická a dynamická paměť umísťuje do samostatně

adresovaných prostorů - segmentů. Tím se program izoluje od jiných a chrání se proti nahodilému přístupu zvenčí.

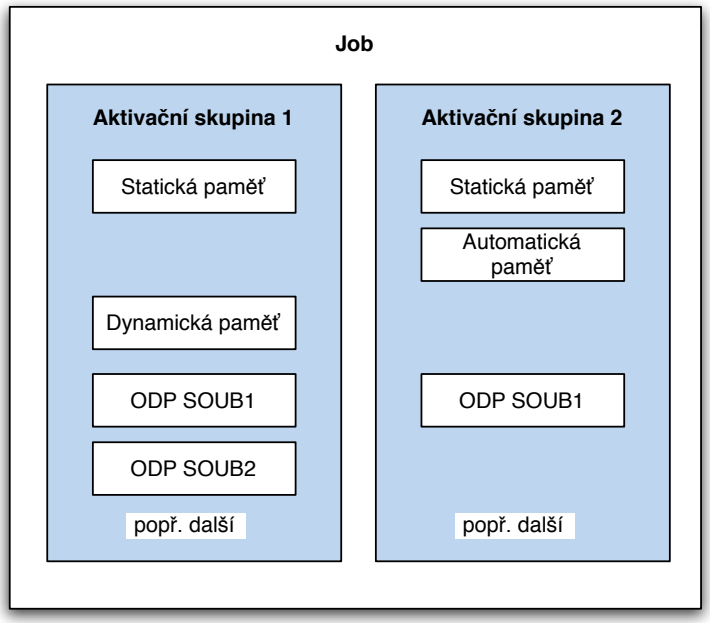
Statická paměť se vytváří při dynamickém volání programu pro všechny jeho moduly a ruší se, když program končí svůj výpočet. Statická paměť je globální paměť modulu definovaná v hlavní proceduře a je dostupná všem podprocedurám deklarovaným v tom modulu.

Automatická paměť se vytváří při vstupu výpočtu do podprocedury (v zásobníku) a při opuštění procedury se ruší.

Dynamická paměť je ta, kterou si program obstará při výpočtu, např. funkcí %ALLOC. Lze ji změnit funkcí %REALLOC nebo zrušit příkazem DEALLOC.

Nejdůležitější *dočasné prostředky* pro řízení dat jsou tyto:

- Otevřené soubory (open data paths - ODP)
- Definice potvrzování transakcí (commitment definitions)
- Lokální SQL kurzory
- Vzdálené SQL kurzory

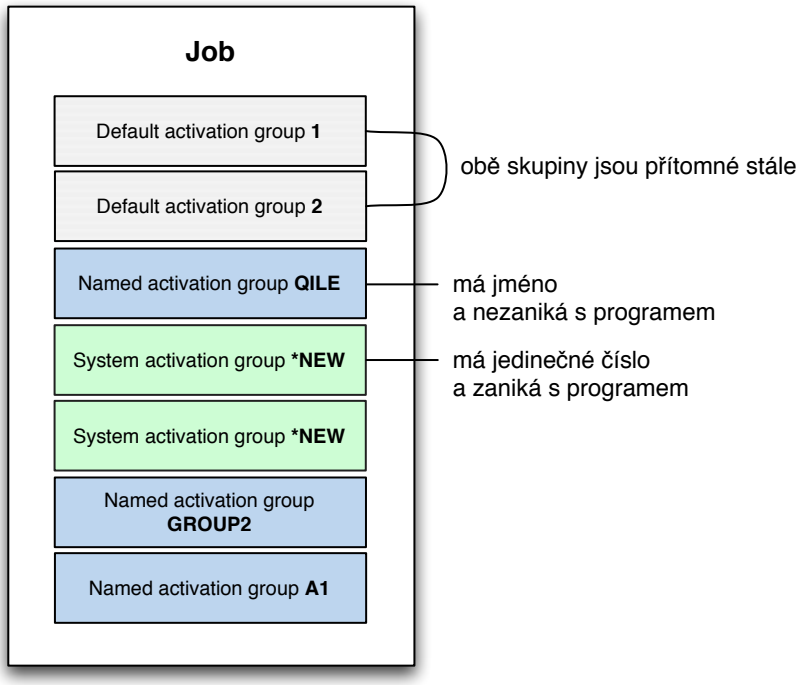


Možnost rozdělit dočasné prostředky mezi aktivační skupiny je podstatná vlastnost ILE. Lze tak vytvořit v různých aktivačních skupinách samostatné aplikace (třeba od různých dodavatelů). Programy aktivované v jedné aktivační skupině mohou např. používat stejné soubory současně s jinými programy v jiných aktivačních skupinách, a to *v rámci jedné úlohy*, aniž by se rušily. Jestliže se jedna z aplikací ukončí a její aktivační skupina se zruší, neovlivní to nijak ostatní aktivační skupiny.

Druhy aktivačních skupin

Předvolená aktivační skupina (*default activation group*) je určena k provozu systémových programů a těch uživatelských programů, které nevyužívají vlastností ILE, zejména podprocedur. Tato aktivační skupina je určena parametrem **DFTACTGRP(*YES)** při kompilaci příkazem CRTBNDRPG. Vyvolaný program běží stále v této aktivační skupině, která nikdy nezaniká, dokud existuje úloha. Ve skutečnosti existují dvě předvolené aktivační skupiny. Skupina s číslem ***1*** hostí systémové programy, např. program QCMD, kdežto aplikační programy běží ve skupině s číslem ***2***.

Pojmenovaná aktivační skupina určená jménem v parametru **ACTGRP(*jméno*)** při vytváření



programu příkazem CRTPGM, popř. servisního programu příkazem CRTSRVPGM. Při vyvolání (aktivaci) programu se tato aktivační skupina vytvoří a zůstává v úloze, i když všechny programy skončí svou činnost.

Systémová aktivační skupina určená parametrem **ACTGRP(*NEW)** při vytváření programu příkazem CRTPGM. Systém jí určí *interní jméno* tak, aby se neshodovalo s jiným. Při každém vyvolání programu se vytvoří nová aktivační skupina, která s *ukončením programu zanikne*. Této skutečnosti lze využít k rekurzivnímu volání programu (program volá sám sebe). Servisní programy nemohou tento parametr využít.

Kompatibilní režim (compatibility mode)

Tímto pojem se označuje provoz programů v předvolené aktivační skupině (default activation group). Program se vytváří nejčastěji příkazem **CRTBNDRPG** s předvolenou hodnotou parametru **DFTACTGRP(*YES)**. Podmínky provozu jsou podobné režimu RPG III (OPM - Original Program Model), ne však úplně stejné. Nelze například použít ladicí program STRISDB (Start Interactive Symbolic Debugger), ale pro ladění se používá příkaz STRDBG.

ILE program vytvořený z jediného modulu

Chceme-li využívat prototypové podprocedury v jediném modulu, můžeme ke kompilaci použít příkaz **CRTBNDRPG**, ale musíme v něm zadat parametr **DFTACTGRP(*NO)** a ponechat předvolené jméno aktivační skupiny (QILE) nebo zvolit jméno v parametru **ACTGRP**.

ILE program vytvořený z modulů a servisních programů

Jestliže spojujeme moduly a servisní programy do programu, musíme použít příkaz **CRTPGM**. V něm můžeme do parametru **ACTGRP** dosadit zvolené jméno**,** nebo *NEW, anebo ***CALLER**. Zvolíme-li *CALLER, spojený program se aktivuje a spustí v aktivační skupině *volajícího* programu.

Spojujeme-li moduly do servisního programu, musíme použít příkaz **CRTSRVPGM**. V něm zadáme parametr **ACTGRP** se zvoleným jménem nebo nebo jej nezadat a tak zvolit předvolenou hodnotu ***CALLER**. Hodnota *NEW zde není přípustná. Počítá se s tím, že servisní program funguje nejefektivněji, běží-li jeho procedury ve stejné aktivační skupině.

Podle potřeby lze pochopitelně volit u různých programů různé kombinace aktivačních skupin. Je dokonce možné kombinovat vzájemné volání ILE programů a OPM programů. OPM programy (RPG III) běží vždy v předvolené aktivační skupině. Takové kombinace se však nedoporučují. Jednak jsou pravidla chování programů při zpracování chybových stavů a při sdílení prostředků značně složitá a jednak přechody mezi aktivačními skupinami vyžadují více času při volání programů.

Otevřené soubory a jejich sdílení

Otevřený soubor je v programu představován tzv. otevřenou datovou cestou (ODP - Open Data Path). ODP je, zhruba řečeno, paměť obsahující ukazatel na data daného souboru (I/O buffer), popř. ukazatel do jeho přístupové cesty (access path). Otevřenou datovou cestu může sdílet více programů.

Poznámka: Jde však o jiné sdílení než je sdílení databázových souborů mezi různými uživateli. Takové sdílení funguje automaticky, sdílejí se při něm data souborů, ale ODP se nesdílí. Každý uživatel si vytváří samostatnou ODP.

Otevřenou datovou cestu mohou programy sdílet pouze *uvnitř jedné úlohy* (job), ať už dávkové či interakční, jestliže je v souboru aktivován parametr **SHARE(*YES)**. Parametr SHARE(*YES) lze zadat v příkazu CRTxxxF, CHGxxxF, ale zejména OVRxxxF (kde xxx může být DB, DSP, PRT, ICF, TAP, DKT, SAV, MSG). Otevřená datová cesta může být sdílena

- na úrovni volání,
- na úrovni aktivační skupiny, nebo
- na úrovni úlohy (job).

Aplikace používají sdílení ODP v hojné míře, pokud je to možné, protože se tím šetří jednak čas potřebný k otevírání a zavírání souborů, a jednak paměťový prostor pro každou novou otevřenou datovou cestu. Uvedeme příklady s databázovým souborem, ačkoliv podobné úvahy platí i pro jiné druhy souborů.

Na *úrovni volání (call level)* mohou otevřenou datovou cestu sdílet programy, které byly volány z této úrovně příkazem CALL. Například program A volá program B, program B volá program C. Datová cesta souboru F1 otevřená v CL programu A příkazy

```
OVRDBF FILE(F1) SHARE(*YES) OVRSCOPE(*CALLLVL)
OPNDBF FILE(F1)
```

je sdílena programy B a C (které také otevírají soubor F1).

Na *úrovni úlohy (job level)* mohou otevřenou datovou cestu sdílet všechny programy spuštěné na kterékoliv úrovni příkazem CALL v rámci celé úlohy. Takové sdílení lze zadat příkazem

```
OVRDBF FILE(F1) SHARE(*YES) OVRSCOPE(*JOB)
```

Sdílení na úrovni úlohy se nedoporučuje, protože je nepružné. Zejména tehdy, když v jedné úloze běží aplikace od různých dodavatelů, každá v jiné aktivační skupině. Aplikace by pak sdílely s datovými záznamy souboru i ODP, což by mohlo způsobit nežádoucí chování programů.

Na *úrovni aktivační skupiny (activation group level)* mohou otevřenou cestu sdílet všechny programy volané příkazem CALL v rámci stejné aktivační skupiny. Sdílení lze zadat následujícím příkazem, je-li vydán *v jiné než předvolené* aktivační skupině:

```
OVRDBF FILE(F1) SHARE(*YES) OVRSCOPE(*ACTGRPDFN)
```

Předvolený parametr OVRSCOPE(*ACTGRPDFN) je pružný. Je-li příkaz OVRDBF vydán *v předvolené* aktivační skupině, sdílení působí na úrovni volání (*CALLLVL), a to i tehdy, když vnořené volání mění aktivační skupinu. Běží-li program *v pojmenované* aktivační skupině, působí sdílení na úrovni aktivační skupiny.

V příkazu OVRDBF lze zadat také parametr **OPNSCOPE**, který může mít hodnotu *ACTGRPDFN (předvolenou) nebo *JOB s podobným významem jako mají v parametru OVRSCOPE. Příkaz

```
OVRDBF FILE(F1) SHARE(*YES) OVRSCOPE(*CALLVL) OPNSCOPE(*ACTGRPDFN)
```

určuje, že soubor F1 lze sdílet na úrovni současné aktivační skupiny, ale jen těmi programy, které „vidí" tento příkaz OVRDBF, tj. ty, které běží ve stejné nebo nižší úrovni volání.

Otevřenou datovou cestu lze přímo vytvořit příkazem **OPNDBF** (Open Database File) s parametrem OPNSCOPE. Ten může mít hodnoty *ACTGRPDFN (předvolená), *JOB nebo *ACTGRP. Hodnota *ACTGRPDFN vytvoří ODP sdílitelnou na úrovni aktivační skupiny nebo na úrovní volání, hodnota *JOB vytvoří ODP sdílitelnou všude na úrovni úlohy, kdežto hodnota *ACTGRP, znamená, že vytvořenou ODP lze sdílet pouze na úrovni aktivační skupiny, v níž byl příkaz OPNDBF vydán.

```
OVRDBF FILE(F1) SHARE(*YES)
OPNDBF FILE(F1) OPNSCOPE(*ACTGRP)
```

Otevřená datová cesta souboru F1 může být sdílena jen v rámci aktivační skupiny, v níž byly tyto dva příkazy vydány.

Ukončování procedur, programů a aktivačních skupin

V RPG III, tedy v modelu OPM se programy ukončují buď slabě, příkazem RETRN, nebo silně, zapnutím indikátoru LR nebo automaticky skončením výpočtu podle RPG cyklu, když jsou vyčerpány vstupní soubory. Slabé ukončení zachová data a otevřené soubory. Silné ukončení způsobí zrušení dat a uzavření souborů.

V ILE RPG je situace složitější, protože je třeba brát v úvahu existenci podprocedur a aktivačních skupin. Pro *normální ukončování* platí zhruba tato pravidla:

- *Podrocedura* se ukončuje příkazem RETURN, který vrátí výpočet do volající procedury (totéž se stane po provedení posledního příkazu podprocedury bez příkazu RETURN).
- *Program* (hlavní procedura) se ukončuje příkazem RETURN, s případně zapnutým indikátorem LR. Zapnutý indikátor LR sice uzavře soubory, ale neodstraní statickou paměť. Pouze ji označí pro novou inicializaci.
- *Aktivační skupina* se ukončuje (ruší) podle toho, o jakou běží.
- *Předvolená* aktivační skupina (parametr DFTACTGRP(*YES) v příkazu CRTBNDRPG) se neukončuje, končí se jen program.
- *Nová*, nepojmenovaná aktivační skupina (parametr ACTGRP(*NEW) v příkazu CRTPGM) se *vždy* ukončuje zároveň s programem. Všechny programové prostředky zaniknou.
- *Pojmenovaná* aktivační skupina se *neukončuje* s koncem programu. Ukončit ji může jen:
  - Statické volání programu **CEETREC** (příkazem CALLB nebo CALLP bez parametrů). V tom případě se z paměti odstraní současná aktivační skupina a ukončí se (normálně) všechny programy této aktivační skupiny.
  - CL příkaz **RCLACTGRP** (Reclaim Activation Group). Ruší všechny pojmenované aktivační skupiny, které právě nejsou používány žádným programem.
  - Abnormální konec programu (neošetřená chyba), je-li to poslední aktivní program (nejstarší položka) této aktivační skupiny v zásobníku volání.
  - Pochopitelně, že všechny pojmenované aktivační skupiny končí nejpozději s koncem úlohy (job).

Zpracování chybových stavů

Zpracování chybových stavů je v prostředí ILE složitější než v OPM, zejména proto, že bere v úvahu existenci aktivačních skupin a přechodů mezi nimi, ale i existenci procedur. Hlavní

procedury i procedury uvnitř programů mají své vlastní položky v zásobníku volání (call stack).

Poznámka: Zásobník volání se v OPM nazýval zásobník programů (program stack), protože v něm byly obsaženy jen položky programů.

V zásobníku volání rozeznáváme určité hranice mezi položkami, zvané *kontrolní hranice (control boundaries)*. Položka je kontrolní hranicí, jestliže je vyvolána buď programem z OPM nebo programem či procedurou z jiné aktivační skupiny. Často je kontrolní hranicí položka představující hlavní proceduru programu, a to tehdy, když ILE program byl vyvolán dynamicky, tj. příkazem CALL, a to i z OPM programu. Kontrolní hranice se nazývá tvrdá (hard control boundary), jestliže je v dané aktivační skupině první. Další se nazývají měkké (soft control boundaries).

Hlavní procedura (main procedure) ILE programu je v zásobníku označena jménem

_QRNP_PEP_*jméno-programu*.

PEP je zkratka pro *Program Entry Procedure*, což je vlastně hlavní procedura programu. Až za PEP je v zásobníku zapsáno jméno vstupního modulu. Servisní program nemá žádnou hlavní proceduru, a tedy ani PEP. OPM program také nemá hlavní proceduru, a tudíž ani PEP; jeho první položka v zásobníku se jmenuje stejně jako program.

Ostatní procedury a podprocedury vstupního modulu a připojených modulů se do zásobníku zapisují tak, jak se postupně vyvolávají a ruší se v obráceném pořadí (LIFO - Last In, First Out).

Vznikne-li v některé RPG proceduře chyba, zpracuje se následujícími prostředky v uvedeném pořadí:

- modifikátor E příkazu, např. CHAIN(E), s následným testem pomocí funkce %ERROR,
- příkaz MONITOR spolu s příkazy ON-ERROR a ENDMON.
- podprogram INFSR pro chybu v souboru (file information subroutine)
- chybový ILE program (ILE condition handler)
- podprogram *PSSR pro chybu v programu (program status subroutine)
- chybová rutina RPG pro jinak neošetřené chyby

Poznámka: Příkaz MONITOR zhruba odpovídá CL příkazu MONMSG.

První čtyři prostředky označí chybovou zprávu (typu *ESCAPE) za ošetřenou, takže zpráva se dále nešíří. Poslední prostředek však neošetřenou zprávu pošle výše do další položky zásobníku, kde se zpracuje stejným způsobem. Tento proces *prosakování (percolation)* pokračuje tak dlouho, dokud není zpráva označena jako ošetřená nebo dokud nedosáhne kontrolní hranice. V proceduře na kontrolní hranici, není-li chyba ošetřena v programu, systém zprávu ošetří podle typu zprávy:

- Zprávu typu *ESCAPE zapíše do protokolu úlohy (job log) a vygeneruje zprávu Function Check. Pak pokračuje ve zpracování zprávy Function Check znovu od místa, kde vznikla zmíněná zpráva typu *ESCAPE.
- Zprávu Function Check nezapisuje do protokolu úlohy, pošle výpočet do volající položky zásobníku (která vyvolala položku kontrolní hranice), přičemž jí pošle zprávu CEE9901 typu *ESCAPE. Tam se opět postupuje dále v obsluze této chyby podle dosud popsaných pravidel.

Příklady – servisní programy

Kombinace servisních programů

Vytvoříme tři jednoduché procedury - funkce, každou v samostatném modulu a zakomponujeme je do dvou servisních programů, které pak připojíme k hlavnímu programu.

První funkce *DynEdit* v modulu M11A provádí dynamickou editaci – edituje 15ciferné pakované číslo s nulovým počtem desetinných míst tak, aby výsledek měl jiný počet desetinných míst, který zadáme jako parametr.

Druhá funkce *NonDigit* v modulu M11B kontroluje znakovou proměnnou, zda obsahuje samé číslice nebo mezery.

Třetí funkce *EdtChrNbr* v modulu M11 používá předchozí dvě k editaci znakově kódovaného čísla bez vyznačené desetinné čárky k jeho převodu (editaci) s desetinnou čárkou umístěnou podle toho, kolik zadáme desetinných míst.

Hlavní program v modulu M10 používá funkci *EdtChrNbr* k zadávání znakových čísel a zobrazení výsledných editací.

Funkce DynEdit, NonDigit a EdtChrNbr

```
**free
// ****************************************************************
// *
// *   Modul M11A - Dynamická editace čísla
// *                Kompiluje se volbou 15 - CRTRPGMOD,
// *                např. CRTRPGMOD MODULE(VZILE/M11A)
// *
// ****************************************************************

ctl-opt nomain decedit('0,');

// Definice procedury

dcl-proc DynEdit export;

    dcl-pi DynEdit  varchar(100);
        Number      packed(15) value;
        DecPos      packed (3) value;
    end-pi DynEdit;

    dcl-s EdtNbr      varchar(100);
    dcl-s WorkNbr     packed(30: 15);
    dcl-s Correction  packed(1);
    dcl-s Pos         packed(3);

    // Je-li počet žádaných des. míst 0 - bude korekce -1

    if DecPos = 0;
    Correction = -1;

    // Je-li počet žádaných des. míst > 0 - bude korekce 0

    else;
    Correction = 0;
    endif;

    // Posunu vstupní číslo (15 0) vpravo o počet desetinných míst
    // (je-li kladný nebo nula) a umístím je do pracovní proměnné
    // (30 15). Např. pro DecPos = 2:
    //          1                   1         2         3
    // 1...5....0....5     1...5....0....5....0....5....0
    // 111111111111111 ==> 001111111111111110000000000000

    WorkNbr = Number * 10 ** -DecPos;
```

```
    // Edituji číslo kódem 3 a výsledek umístím do znakové proměnné
    // s proměnlivou délkou:
    // '  1111111111111.110000000000000'

    EdtNbr  = %Editc(WorkNbr :'3');

    // Vyjmu z textu číslo bez koncových číslic:
    // Text od pozice 1 do konce minus 15,
    //                           plus des. místa,
    // '  1111111111111.11'      plus korekce

    EdtNbr = %subst(EdtNbr : 1 :
                    %len(EdtNbr) -15 +DecPos
                    +Correction );

    // Vrátím editované číslo jako znakovou proměnnou s proměnnou délkou

    return EdtNbr;

end-proc DynEdit;

**free
// ****************************************************************
// *
// *   Modul M11B - Kontrola řetězce na nečíslicové znaky
// *                Kompiluje se volbou 15 - CRTRPGMOD,
// *                např. CRTRPGMOD MODULE(VZILE/M11B)
// *
// ****************************************************************
ctl-opt nomain;

// Definice procedury

dcl-proc NonDigit export;

    dcl-pi NonDigit ind;
        String   varchar(100);
        Position packed(3);
    end-pi;

    // Kontrola, zda se ve vstupním řetězci najde chybný znak
    // (tj. jiný než číslice nebo mezera)

    Position = %check(String: ' 0123456789': 1);

    // Jestliže se našel, nahradím vstupní řetězec samými nulami
    // a vrátím *ON

    if Position = 0;
        String = *all'0';
        return *on;

    // Jestliže jsou to samé číslice nebo mezery, nahradím mezery nulami
    // a vrátím *OFF

    else;
        String = %xlate(' ': '0': String: 1);
        return *off;
    endif;

end-proc NonDigit;

**free
// ****************************************************************
// *
// *   Modul M11 - Editace znakově kódovaných čísel
// *               Kompiluje se volbou 15 - CRTRPGMOD,
// *               např. CRTRPGMOD MODULE(VZILE/M11)
// *
// ****************************************************************

ctl-opt nomain;
```

```
// Prototypy volaných procedur - nutné: pro volané procedury

/copy QRPGLESRC,CpyS10
/copy QRPGLESRC,CpyS11

// Definice procedury

dcl-proc EdtChrNbr export;

    dcl-pi EdtChrNbr ind;
        CharNbr   varchar(100);
        DecPos    packed(3) value;
        EditedNbr varchar(100);
    end-pi;
    dcl-s Pos     packed(3);
    dcl-s Number  packed(15);
    dcl-c NumberDigits const(%len(Number));

    // Kontrola, zda znaky zahrnují jen číslice nebo mezery

    if  NonDigit(CharNbr : Pos);

        // Nejsou-li samé číslice nebo mezery - vrátím hodnotu *On (chyba)

        return *on;
    endif;

    // Jinak (samé číslice nebo mezery) - Převod znaků na číslo

    Number = %dec(CharNbr: NumberDigits: 0);
    // c                   move(p)   CharNbr       Number

    // Editace čísla s požadovaným počtem des. míst (ediční kód 3)

    EditedNbr = DynEdit(Number : DecPos);

    // Vracím *off (OK)

    return *off;

end-proc EdtChrNbr;
```

Modul M10 pro program P10 – ukázka použití funkcí

```
**free
//*****************************************************************
//*
//*   Modul M10 - Ukázka editace znakově kódovaných čísel
//*
//*****************************************************************

// Popis souborů - Obrazovkový soubor

dcl-f CHRNUMW workstn;

// Definice dat

dcl-s EditNbr    varchar(100);
dcl-s RC         ind;
dcl-s CharNbr    varchar(100);

// Prototypy volaných procedur

/copy QRPGLESRC,CpyS10

dow 0 = 0;

    // Zobrazit první formát k zadání znakově kódovaného čísla
    // a požadovaného počtu desetinných míst

    exfmt CHRNUMWR;
```

```
    if *in03;
        leave;
    endif;

    // Editace znakově kódovaného čísla s požadovaným počtem des. míst

    CharNbr = CHRNBR;
    RC = EdtChrNbr(CharNbr: DecPos: EditNbr);

    // Po chybě - Dosadit samé mezery

    if RC;
        EDTNBR = *Blanks;
    else;

        // Jinak (OK) - Odříznout koncové mezery a posunout vpravo

        evalr EDTNBR = %trimr(EditNbr);
    endif;

    // Zobrazit výsledek v druhém obrazovkovém formátu

    exfmt CHRNUMWR2;
    if *in03;
        leave;
    endif;

enddo;

*inlr = *on;
```

Zdrojový člen CpyS11 s prototypy funkcí DynEdit a NonDigit

```
**free
// ****************************************************************
// *
// *   Prototypy funkcí DynEdit a NonDigit
// *
// ****************************************************************

// Prototyp funkce DynEdit - Dynamická editace

dcl-pr DynEdit   varchar(100);
    Number  packed(15) Value;
    DecPos  packed(3)  Value;
end-pr;

// Prototyp funkce NonDigit - Kontrola nečíslicových a nemezerových
//                            znaků ve znakové proměnné

dcl-pr NonDigit   ind;
    String    varchar(100);
    Position  packed(3);
end-pr;
```

Zdrojový člen CpyS10 pro funkci EdtChrNbr

```
**free
// ****************************************************************
// *
// *   Prototyp funkce EdtChrNbr
// *
// ****************************************************************

dcl-pr EdtChrNbr   ind;
    CharNbr    varchar(100);
    DecPos     packed(3) Value;
    EditedNbr  varchar(100);
end-pr;
```

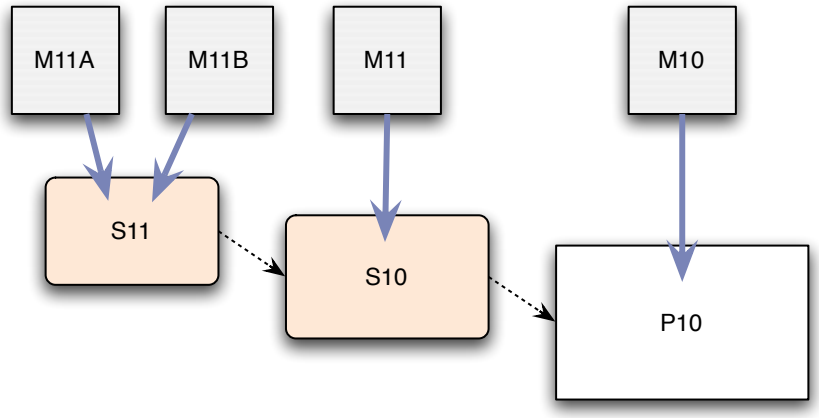
Obrazovkový soubor pro modul M10

```
     ************************************************************
     *
     *   CHRNUMW - Zobrazení znakově kódovaných čísel
     *
     ************************************************************
A                                      DSPSIZ(24 80 *DS3)
A                                      CA03(03 'End')
     *   Formát k zadání znakového čísla a počtu desetinných míst
A          R CHRNUMWR
A                                   5  4'Zadejte číslo bez speciálních -
A                                      znaků:'
A                                      DSPATR(HI)
A            CHRNBR        15A  B  6  5
A                                   8  4'Zadejte počet desetinných míst, -
A                                      které chcete mít v editovaném -
A                                      čísle:'
A                                      DSPATR(HI)
A            DECPOS         3  0B  9  5EDTCDE(4)

     *   Formát k zobrazení výsledného editovaného čísla
A          R CHRNUMWR2
A                                   4  4'Znakově kódované číslo:'
A            CHRNBR        15A  O  6  5
A                                   8  4'Žádaný počet des. míst:'
A            DECPOS         3Y 0O  9  5EDTCDE(3)
A                                  12  4'Výsledné editované číslo:'
A            EDTNBR        16A  O 13  5DSPATR(RI)
A                                  16  5'Stiskněte Enter.'
A                                      COLOR(BLU)
A                                  23  2'F3=Konec'
A                                      COLOR(BLU)
```

Kompilace modulů, vytvoření servisních programů a programu

Moduly M10, M11, M11A, M11B vytvoříme ze zdrojů (zkompilujeme) volbou 15 v PDM nebo příkazy CRTRPGMOD.

Vytvoříme dva servisní programy, S11 a S10 (v tomto pořadí). V příkazech CRTSRVPGM uvedeme parametr EXPORT(*ALL), který znamená, že spojovací program si vytvoří signatury servisních programů ze všech exportovaných procedur. Program P10 pak vytvoříme spojením modulu M10 a servisního programu S10 (spojeného z modulu M11 a servisního programu S11).




```
/*   Vytvoření servisního programu S11 z modulů M11A a M11B         */

             CRTSRVPGM  SRVPGM(*CURLIB/S11) MODULE(*LIBL/M11A M11B) +
                          EXPORT(*ALL)

/*   Vytvoření servisního programu S10 z modulu M11                 */

             CRTSRVPGM  SRVPGM(*CURLIB/S10) MODULE(*LIBL/M11) +
                          EXPORT(*ALL) BNDSRVPGM(*LIBL/S11)

/*   Vytvoření programu P10 z modulu M10                            */
/*                       a servisního programu S10                  */

             CRTPGM     PGM(*CURLIB/P10) MODULE(*LIBL/M10) +
                          BNDSRVPGM(*LIBL/S10) ACTGRP(QILE)
```

Ze servisních programů můžeme získat spojovací zdrojové texty typu BND příkazem RTVBNDSRC.

Příkaz

```
RTVBNDSRC MODULE(M11A M11B) SRCMBR(S11)
```

vytvoří člen S11 ve zdrojovém souboru QSRVSRC:

```
  STRPGMEXP PGMLVL(*CURRENT)
/****************************************************************/
/*   *MODULE      M11A         VZILE        05/05/20  16:47:39      */
/****************************************************************/
  EXPORT SYMBOL("DYNEDIT")
/****************************************************************/
/*   *MODULE      M11B         VZILE        05/05/20  16:47:39      */
/****************************************************************/
  EXPORT SYMBOL("NONDIGIT")
```

```
  ENDPGMEXP
```

Příkaz

```
  RTVBNDSRC MODULE(M11) SRCMBR(S10)
```

vytvoří člen S10 ve zdrojovém souboru QSRVSRC:

```
 STRPGMEXP PGMLVL(*CURRENT)
 /****************************************************************/
 /*   *MODULE      M11          VZILE        05/05/20  16:51:25      */
 /****************************************************************/
   EXPORT SYMBOL("EDTCHRNBR")
 ENDPGMEXP
```

Cvičení

Spusťe program P10. Vypište a prohlédněte údaje o obou servisních programech S10, S11 příkazem DSPSRVPGM.

Vypište a prohlédněte údaje o programu P10 příkazem DSPPGM.

Vytvořte jeden servisní program ze všech tří modulů M11, M11A, M11B a připojte k modulu M10. Výsledný program (např. P10A) spusťte a pak opět vypište údaje o servisním programu a o programu.

Datumové funkce + cvičení

V následujícím příkladu uvidíme, jakým způsobem lze využít spojovací zdrojový text (binder source) k údržbě servisních programů.

Sestrojíme jeden servisní program SDATUM se dvěma funkcemi: funkce *dnesni_datum* dodá systémové datum, funkce *pricist_dny* přičte k zadanému datu zadaný počet dnů a vrátí výsledek. Data jsou číselná a mají tvar CYYMMDD, kde

C = 0 pro roky 1900-1999,
C = 1 pro roky 2000-2099,
. . .
C = 9 pro roky 2800-2899.

Člen MDATUM obsahuje zdrojové texty funkcí. Úkolem je vytvořit modul MDATUM a z něj servisní program SDATUM. Před kompilací modulu vytvoříme zdrojový člen CPYSDATUM. Před vytvořením servisního programu také napíšeme spojovací zdrojový text (binder source) se seznamem exportů (jmen funkcí) a dodat jeho jméno spojovacímu programu.

Pak napíšeme a vytvoříme program PDAT01 z modulu MDAT01 a servisního programu SDATUM a vyzkoušíme, zda funkce fungují.

Jako další krok a cvičení doplníme do servisního programu další funkci *odecist_data* a vytvoříme nový program PDAT02 z modulu MDAT02 a nově doplněného servisního programu SDATUM. Musíme to zařídit tak, aby první program PDAT01 fungoval beze změny, tj. bez nového spojování s novým servisním programem.

Modul MDATUM pro servisní program SDATUM

```
**FREE
// ********************************************************************
// *
// *   MDATUM - Modul pro servisní program SDATUM
// *
// ********************************************************************
ctl-opt nomain;
```

```
// Prototypy volaných procedur servisního programu SDATUM
// - volitelné: procedury nemusí mít vlastní prototypy
// /Copy QRPGLESRC,CpySDATUM

// *====================================================================
// *   Funkce dnesni_datum
// *   vrací systémový čas ve tvaru CYYMMDD jako číslo (7 0)
// *====================================================================

dcl-proc  dnesni_datum  export;
    dcl-pi dnesni_datum packed(7);
    end-pi;

    dcl-s ISOtime       timestamp;
    dcl-s dnesni_datum  packed(7);
    dcl-s znakove_datum char(7);
    dcl-s prac_datum    date(*ymd);

    ISOtime = %timestamp;
    prac_datum = %date(ISOtime);
    dnesni_datum = %dec(prac_datum: *cymd);
    return dnesni_datum;
end-proc dnesni_datum;

// *====================================================================
// *   Funkce pricist_dny
// *   vrací datum zvýšené o počet dnů
// *====================================================================

dcl-proc pricist_dny  export;
    dcl-pi pricist_dny     packed(7);
        vstupni_datum   packed(7) value;
        pocet_dni       int(10) value;
    end-pi;
    dcl-s vysledne_datum  packed(7);
    dcl-s prac_datum      date(*ymd);

    prac_datum = %date(vstupni_datum: *cymd);
    prac_datum += %days(pocet_dni);
    vysledne_datum = %dec(prac_datum: *cymd);
    return vysledne_datum;
end-proc pricist_dny;
```

Modul MDAT01 pro program PDAT01

Abychom servisní program vyzkoušeli, vytvoříme modul MDAT01 z následujícího zdrojového členu. Modul pak spojíme do programu PDAT01 se servisním programem SDATUM.

```
**free
// ************************************************************
// *
// *   Modul MDAT01 - Test datumových funkcí servis.prog. SDATUM
// *
// ************************************************************

dcl-s nove_datum  packed(7);

// Prototypy volaných procedur
/Copy QRPGLESRC,CpySDATUM

nove_datum = dnesni_datum;
dsply nove_datum;

nove_datum = pricist_dny(dnesni_datum: 7);
```

```
dsply nove_datum;

*inlr = *on;
```

Doplnění servisního programu o další funkci - cvičení

Doplňte servisní program SDATUM o novou funkci *odecist_data* se dvěma parametry (první a druhé datum), která vrátí jejich rozdíl v počtu dnů.

```
**free
// *====================================================================
// *   Funkce odecist_data
// *   vrací rozdíl dvou datumů v počtu dnů
// *====================================================================

dcl-proc odecist_data  export;
    dcl-pi odecist_data int(10); // vrací počet dnů
         datum1 packed(7) value; //
         datum2 packed(7) value;
    end-pi;

    dcl-s prac_dat1  date(*ymd);
    dcl-s prac_dat2  date(*ymd);
    dcl-s dny        int(10);

    prac_dat1 = %date(datum1: *cymd);
    prac_dat2 = %date(datum2: *cymd);
    dny = %diff(prac_dat2: prac_dat1: *days);
    return dny;
end-proc odecist_data;
```

K vyzkoušení nové funkce vytvořte program PDAT02, který tu funkci otestuje. Předchozí program PDAT01 musí fungovat beze změny, aniž bychom ho znovu spojovali s novým servisním programem.

Zopakujte celý postup s tím, že ve spojovacím zdrojovém textu nahradíte hodnotu *GEN v parametru SIGNATURE hodnotami “V.01“ a “V.02“.

Příklady - aktivační skupiny

V následujících příkladech ukážeme účinky, jaké mají aktivační skupiny na provoz programů v rámci úlohy (jobu). Dvojice RPG programů, z nichž jeden volá druhý, čte a tiskne záznamy ze stejného databázového souboru. Soubor je definován pro sdílení s parametrem SHARE(*YES). Příklady jsou určeny ke zkoušení, jak se programy chovají, běží-li ve stejné aktivační skupině nebo ve dvou různých aktivačních skupinách.

Je vhodné programy modifikovat a pozorovat jejich činnost na tiskových sestavách a na zásobníku volání. Například lze ponechat zapínání indikátoru LR nebo je zrušit, zařadit do CL programu nebo RPG programu rušení aktivačních skupin apod. Některé navrhované modifikace jsou v programech označeny otazníky.

Soubory použité v příkladech

V následujících příkladech jsou použity dva jednoduché soubory – databázový soubor ITEMS a tiskový soubor REPORT. Jejich popisy DDS uvádíme předem.

Definice databázového souboru ITEMS

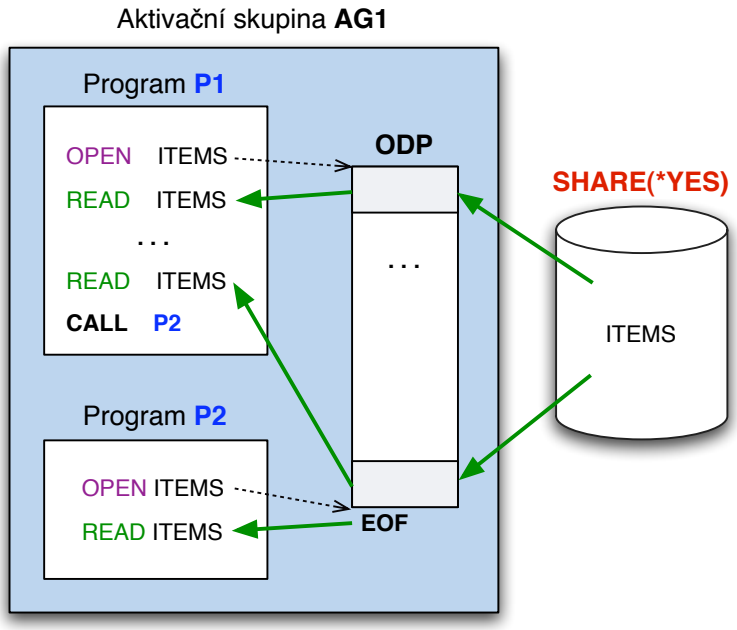
```
     ************************************************************
     *
     *   ITEMS - Soubor položek
     *
     ************************************************************
    A                                     UNIQUE
    A           R ITEMSR
     *   Číslo položky
    A             ITEMNBR         5       COLHDG('Číslo' 'položky')
     *   Jednotková cena
    A             UNITPR          9  2    COLHDG('Jedn.' 'cena')
     *   Popis položky
    A             ITEMDESC       50       COLHDG('Popis položky')

     *   Klíčové pole
    A           K ITEMNBR
```

Definice tiskového souboru REPORT

```
     ************************************************************
     *
     *   REPORT - Seznam položek
     *
     ************************************************************
    A                                     REF(ITEMS)

    A           R ITEMDETAIL              SPACEA(1)
    A             ITEMNBR    R           2
    A             UNITPR     R          +2 EDTCDE(Q)
    A             ITEMDESC   R          +2

    A           R EOFLINE                 SPACEA(1)
    A             EOFTEXT        50      2
```

Programy ve stejné aktivační skupině

Dva programy, P1 a P2 mají společnou aktivační skupinu pojmenovaou AG1. Oba programy čtou sekvenčně databázový soubor ITEMS, jehož přístupová cesta se otevírá jako sdílená, tedy s parametrem SHARE (*YES) na úrovni aktivační skupiny. Program P1 přečte celý soubor a vyvolá program P2, který tentýž soubor čte znovu. Program P2 ale nepřečte žádný záznam, protože program P1 nechal soubor nastavený na konci. Je to tím, že oba programy sdílejí stejnou otevřenou cestu k souboru a stejnou aktivační skupinu.



RPG program P1 – volá program P2

```
     CTL-OPT DFTACTGRP(*NO) ACTGRP('AG1');

    *   Popis souborů

     DCL-F ITEMS  USAGE(*UPDATE);
     DCL-F REPORT PRINTER;

    *   Popis dat

     DCL-S Odpoved CHAR(1);
     DCL-S Delitel PACKED(1);

    *   Výpočet

     Read(N)  ITEMS;

     If        %EoF;
         EOFTEXT = 'P1 - konec souboru';
     Else;
         EOFTEXT = 'P1 - začátek souboru';
     EndIf;
     Write     EOFLINE;

     DoW       Not %EoF;
         Write     ITEMDETAIL;
         Read(N)  ITEMS;
     EndDo;

   C                   Call(E)   'P2'

     Dsply 'zadejte Y' '*EXT ' Odpoved;
     If        Odpoved = 'Y' or Odpoved = 'y';
         Delitel = 0;
         UNITPR = UNITPR / Delitel;
     EndIf;

     *InLR = *On;
     Return;
```

V programu P2 je zadána stejná aktivační skupina AG1 jako v programu P1. Alternativně by mohl být v příkazu H zadán parametr ACTGRP(*CALLER).

RPG program P2

```
     CTL-OPT DFTACTGRP(*NO) ACTGRP('AG1');

    *   Popis souborů

     DCL-F ITEMS USAGE(*UPDATE);
     DCL-F REPORT  PRINTER;

    *   Popis dat

     DCL-S Odpoved CHAR(1);
     DCL-S Delitel PACKED(1);

    *   Výpočet

     Read      ITEMS;

     If        %EoF;
         EOFTEXT = 'P2 - konec souboru';
     Else;
         EOFTEXT = 'P2 - začátek souboru';
     EndIf;
     Write     EOFLINE;

     DoW       Not %EoF;

         UNITPR = UNITPR * 1,01;
         Update    ITEMSR;

         Write     ITEMDETAIL;

     Read      ITEMS;
     EndDo;

    *   Vyvolání chyby dělení nulou na žádost uživatele
     Dsply 'zadejte X' '*EXT' Odpoved;
     If        Odpoved = 'X' or Odpoved = 'x';
         Delitel = 0;
         UNITPR = UNITPR / Delitel;
     EndIf;

     *InLR = *On;
     Return;
```

V obou programech je příkaz DSPLY, který dovoluje uživateli vynutit chybu v programu, jejíž postupné řešení lze pozorovat v zásobníku volání (call stack).

Program P1 vyvoláme v CL programu P1CALL, kde určíme souboru ITEMS sdílení v rámci aktivační skupiny. Kromě souboru ITEMS sdílí otevřenou cestu také tiskový soubor REPORT. Znamená to, že oba programy tisknou do stejného spool souboru. Programy se zde chovají jako by byly napsány v RPG III.

CL program P1CALL – připraví sdílení v rámci aktivační skupiny

```
/*****************************************************************/
/*                                                               */
/*   Volání programu P1, který dále volá program P2 ve stejné    */
/*   aktivační skupině AG1.                                      */
/*   Program P1 vytiskne obsah souboru ITEMS a zůstane na konci. */
/*   Program P2 začne od konce souboru a nepřečte nic.           */
/*                                                               */
```

```
/*****************************************************************/

/*   Sdílení databázového souboru ITEMS na úrovni společné           */
/*   aktivační skupiny.                                              */
/*   Budoucí přístupová cesta bude otevřena a sdílena                */
/*   v rámci společné aktivační skupiny AG1 obou programů P1 a P2    */

             OVRDBF     FILE(ITEMS) OVRSCOPE(*ACTGRPDFN) SHARE(*YES) +
                          OPNSCOPE(*ACTGRPDFN)

/*   Sdílení tiskového souboru REPORT na úrovni společné             */
/*   aktivační skupiny.                                              */
/*   Budoucí přístupová cesta bude otevřena a sdílena                */
/*   v rámci společné aktivační skupiny AG1 obou programů P1 a P2    */

             OVRPRTF    FILE(REPORT) OVRSCOPE(*ACTGRPDFN) +
                          SHARE(*YES) OPNSCOPE(*ACTGRPDFN)

/*   Volání programu P1, který dále volá program P2                  */

             CALL       PGM(P1)

/*   Zrušení aktivační skupiny AG1. Tím uzavřu soubory a paměť.      */

/*????       RCLACTGRP  ACTGRP(AG1) OPTION(*NORMAL)               ***/
```

Výsledkem spuštění programu P1CALL je tato tisková sestava (částky se v programu P2 nezměnily, protože se záznamy vůbec nečetly) :

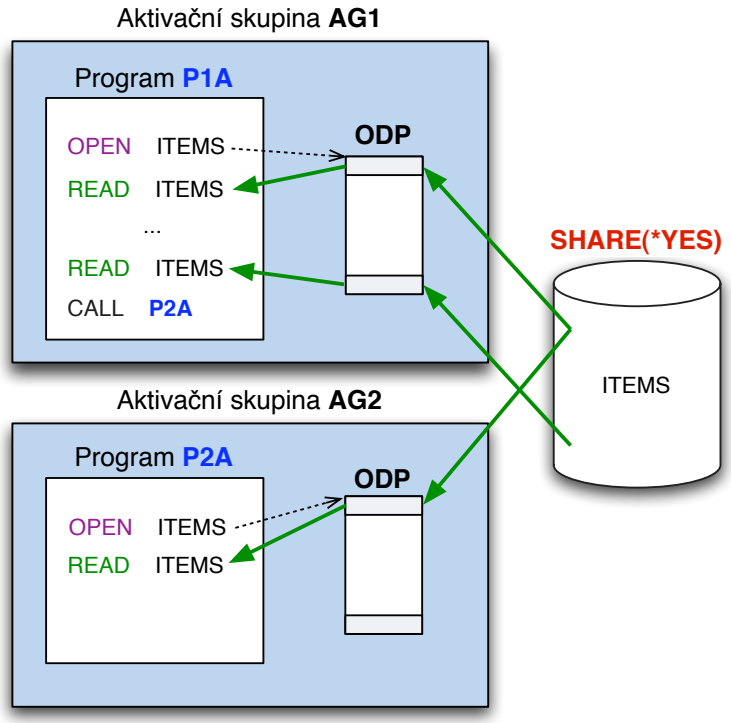
```
P1 - začátek souboru
00001         1,00  Položka 1
00002         2,00  Položka 2
00003         3,00  Položka 3
P2 - konec souboru
```

Tisková sestava je jen jedna, protože druhý program sdílí datovou paměť (buffer) souboru REPORT s prvním programem.

Programy v rozdílných aktivačních skupinách

Program P1A má aktivační skupinu AG1, program P2A má aktivační skupinu AG2.

Oba, stejně jako předchozí programy, čtou soubor ITEMS od začátku do konce. Otevřená cesta k souboru ITEMS je opět sdílená na úrovni aktivační skupiny. Tentokrát má však každý program k dispozici svou vlastní otevřenou cestu ODP, takže po otevření přečte každý program první záznam.



RPG program P1A – volá program P2A

```
     CTL-OPT DftActGrp(*NO) ActGrp('AG1');

    *   Popis souborů

     DCL-F ITEMS;
     DCL-F REPORT Printer UsrOpn;

    *   Popis dat

     DCL-S Odpoved CHAR(1);
     DCL-S Delitel PACKED(1);

    *   Výpočet

     Read       ITEMS;
     Open       REPORT;

     If         %EoF;
         EOFTEXT = 'P1A - konec souboru';
     Else;
         EOFTEXT = 'P1A - začátek souboru';
     EndIf;
     Write      EOFLINE;

     DoW        Not %EoF;
         Write      ITEMDETAIL;
         Read       ITEMS;
```

```
  EndDo;

 *   Volám program P2A s aktivační skupinou AG2
C                   Call(E)   'P2A'

 *   Vyvolání chyby dělení nulou na žádost uživatele
  Dsply 'zadejte Y' Odpoved;
  If        Odpoved = 'Y' or Odpoved = 'y';
      Delitel = 0;
      UNITPR = UNITPR / Delitel;
  EndIf;

  *InLR = *On;
C*????              CallB     'CEETREC'

  Return;
```

Program P2A má aktivační skupinu AG2, jinou než program P1A, který ho vyvolává.

RPG program P2A

```
  CTL-OPT DFTACTGRP(*NO) ACTGRP('AG2');

 *   Popis souborů

  DCL-F ITEMS   USAGE(*UPDATE);
  DCL-F REPORT  Printer;

 *   Popis dat

  DCL-S Odpoved CHAR(1);
  DCL-S Delitel PACKED(1);

 *   Výpočet

  Read      ITEMS;

  If        %EoF;
      EOFTEXT = 'P2A - konec souboru';
  Else;
      EOFTEXT = 'P2A - začátek souboru';
  EndIf;
  Write     EOFLINE;

  DoW       Not %EoF;

      UNITPR = UNITPR * 1,01;
      Update    ITEMSR;

      Write     ITEMDETAIL;

      Read      ITEMS;
  EndDo;

 *   Vyvolání chyby dělení nulou na žádost uživatele
  Dsply 'zadejte X' '*EXT' Odpoved;
  If        Odpoved = 'X' or Odpoved = 'x';
      Delitel = 0;
      UNITPR = UNITPR / Delitel;
  EndIf;

  *InLR = *On;
  Return;
```

Program P1A je volán z následujícího CL programu P1CALLA, který určuje souboru ITEMS

sdílení otevřené přístupové cesty v rámci aktivační skupiny. Protože však každý program běží v jiné aktivační skupině, tato vlastnost se nemůže uplatnit. Výsledek je, že oba programy, P1A i P2A zpracují celý soubor od začátku až do konce. Oba soubory se nakonec uzavřou, protože se zapnul indikátor LR.

CL program P1CALLA – připraví sdílení v rámci aktivační skupiny

```
/*   Sdílení databázového souboru ITEMS v rámci aktivační skupiny.   */
/*   Budoucí otevřené cesty k souboru ITEMS budou sdíleny v každé    */
/*   aktivační skupině samostatně.                                   */

             OVRDBF     FILE(ITEMS) OVRSCOPE(*ACTGRPDFN) SHARE(*YES)

/*   Volání programu P1A, který dále volá program P2A.               */
/*   Program P1A vytváří aktivační skupinu AG1,                      */
/*   program P2A vytváří aktivační skupinu AG2                       */

             CALL       PGM(P1A)

/*   Zabráním havárii po chybném ukončení programu P1A               */

             MONMSG     MSGID(CPF0000)

/*   Zrušení aktivační skupiny AG1 a AG2                             */

     /*????  RCLACTGRP  ACTGRP(AG1) OPTION(*NORMAL) **/
     /*????  RCLACTGRP  ACTGRP(AG2) OPTION(*NORMAL) **/

/*   Alternativně bych mohl zrušit aktivační skupiny statickým       */
/*   voláním programu CEETREC na konci programu P1A a P2A.           */

/*   Nebo je nemusím rušit vůbec, ale pak musím zabezpečit           */
/*   uzavření souborů                                                */
```

Výsledkem spuštění programu P1CALLA jsou dvě samostatné sestavy (soubor REPORT již nesdílí otevřenou cestu). V obou sestavách jsou vytištěny všechny záznamy souboru ITEMS, přičemž program P2A zvýšil částky o 1 procento, jak je v něm stanoveno.

```
P1A - začátek souboru
00001         1,00  Položka 1
00002         2,00  Položka 2
00003         3,00  Položka 3

P2A - začátek souboru
00001         1,01  Položka 1
00002         2,02  Položka 2
00003         3,03  Položka 3
```

Pro ilustraci, jak je komplikované sdílet otevřenou cestu v rámci celé úlohy při různých aktivačních skupinách, uvádíme program P1CALLJ, který toto uspořádání vytvoří.

CL program P1CALLJ – připraví sdílení v rámci úlohy

```
/*   Sdílení databázového souboru ITEMS v rámci celé úlohy           */
             OVRDBF     FILE(ITEMS) OVRSCOPE(*JOB) SHARE(*YES)

/*   Otevření přístupové cesty k souboru ITEMS                       */
/*   s platností v celé úloze                                        */
             OPNDBF     FILE(ITEMS) OPTION(*ALL) OPNSCOPE(*JOB)

/*   Vyvolání programu P1A s aktivační skupinou AG1, který dále      */
/*   volá program P2A s aktivační skupinou AG2                       */
             CALL       PGM(P1A)
```

```
/*   Zabráním havárii, když P1A skončí chybou *ESCAPE               */
             MONMSG     MSGID(CPF0000)

/*   Zrušení obou aktivačních skupin                                */
             RCLACTGRP  ACTGRP(AG1) OPTION(*NORMAL)
             RCLACTGRP  ACTGRP(AG2) OPTION(*NORMAL)

/*   Uzavření souboru ITEMS                                         */
/*   Kdybych jej neuzavřel, zůstal by otevřený až do konce úlohy.   */
/*   Zde, na úrovni *JOB, na to neplatí ani RCLRSC, ani RCLACTGRP,  */
/*   ani volání 'CEETREC'.                                          */
             CLOF       OPNID(ITEMS)

/*   Alternativně lze zrušit aktivační skupiny voláním programu     */
/*   CEETREC v programu P2A a P1A.                                  */
```

K souboru ITEMS se v důsledku sdílení v rámci úlohy otevřou dvě cesty, jedna v programu P1A a druhá v progamu P2A. Ty zůstanou otevřené stále, i po zrušení obou aktivačních skupin. Kdybychom neuzavřeli soubor příkazem CLOF, zůstaly by cesty otevřené až do skončení úlohy.

Statická paměť servisního programu

Tento příklad ukazuje aplikaci, v níž dva ILE programy, PGMA, PGMB volají proceduru ProcP1 ze servisníno programu SPGMX. Přitom programy i servisní program běží *každý v jiné aktivační skupině*. Příklad demonstruje, jak se projeví *statická paměť* servisního programu, jestliže se používá k produkování návratové hodnoty pro volající programy. Je-li návratová hodnota čerpána z jedné statické proměnné, bez ohledu na to, který program proceduru volal, může dostat každý z programů neočekávaný výsledek.

V proceduře se statická proměnná zvyšuje při každém volání o 1. Když se volající programy střídají v nahodilém pořadí, získávají z procedury ProcP1 nahodilé výsledky – číslo zvětšené o nahodilou hodnotu.

Jestliže však procedura vezme v úvahu parametr, v němž volající program předá svou identifikaci (‘PGMA‘, ‘PGMB‘), může každému z nich vrátit správný, očekávaný výsledek (číslo zvětšené o 1 po každém volání).

Servisní program s volanou procedurou

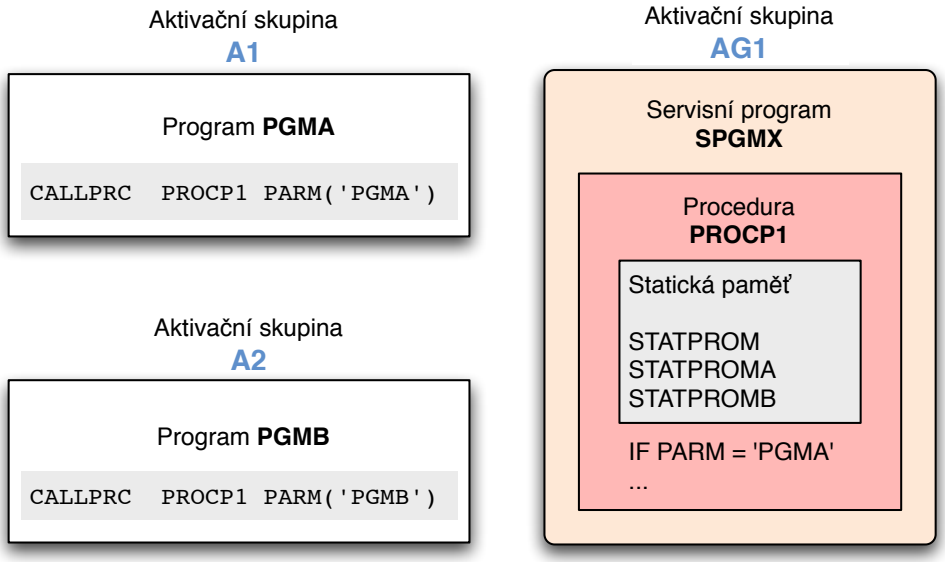
Modul SPGMX pro servisní program SPGMX. Obsahuje statické proměnné a definici procedury PROCP1.

```
ctl-opt nomain;

 // Statické proměnné
 dcl-s StatProm   packed(5: 0);
 dcl-s StatPromA  packed(5: 0);
 dcl-s StatPromB  packed(5: 0);

 // Procedura
 dcl-proc ProcP1  export;

     dcl-pi ProcP1  packed(5);
         Parm1 char(10);
     end-pi;

     if *on;  // podruhé zaměnit na *off
        // Varianta 1 - bez ohledu na to, kdo volal
        StatProm = StatProm + 1;
     else;
        // Varianta 2 - rozlišuje, kdo volal
        if Parm1 = 'PGMA';  // Volal PGMA
            StatPromA = StatPromA + 1;
            StatProm = StatPromA;
        endif;
        if Parm1 = 'PGMB';  // Volal PGMB
            StatPromB = StatPromB + 1;
            StatProm = StatPromB;
        endif;
     endif;

     dsply Parm1 ' ' StatProm;

     return StatProm;
 end-proc ProcP1;
```

Servisní program SPGMX bude mít aktivační skupinu AG1.

```
CRTRPGMOD  MODULE(SPGMX) SRCFILE(QCLSRC) SRCMBR(SPGMX)
CRTSRVPGM  SRVPGM(SPGMX) MODULE(SPGMX) EXPORT(*ALL) ACTGRP(AG1)
```

Volající programy v různých aktivačních skupinách



Program **PGMA** – volá proceduru PROCP1 s identifikačním parametrem ‘PGMA‘ a získá návratovou hodnotu STATPROM.

```
DCL         VAR(&PARM1) TYPE(*CHAR) LEN(10) VALUE(PGMA)
DCL         VAR(&STATPROM) TYPE(*DEC) LEN(5 0)

CALLPRC     PRC(PROCP1) PARM(&PARM1) +
            RTNVAL(&STATPROM)

DMPCLPGM
```

Program **PGMB** – volá proceduru PROCP1 s identifikačním parametrem ‘PGMB‘ a získá návratovou hodnotu STATPROM.

```
DCL         VAR(&PARM1) TYPE(*CHAR) LEN(10) VALUE(PGMB)
DCL         VAR(&STATPROM) TYPE(*DEC) LEN(5 0)

CALLPRC     PRC(PROCP1) PARM(&PARM1) +
            RTNVAL(&STATPROM)

DMPCLPGM
```

Program PGMA bude mít aktivační skupinu A1.

```
CRTCLMOD    MODULE(PGMA) SRCFILE(QCLSRC) SRCMBR(PGMA)
CRTPGM      PGM(PGMA) BNDSRVPGM(SPGMX) ACTGRP(A1)
```

Program PGMB bude mít aktivační skupinu A2.

```
CRTCLMOD    MODULE(PGMB) SRCFILE(QCLSRC) SRCMBR(PGMB)
CRTPGM      PGM(PGMB) BNDSRVPGM(SPGMX) ACTGRP(A2)
```

Zkouška aplikace

Z příkazového řádku voláme v nahodilém pořadí programy PGMA, PGMB:

```
call pgmb
call pgmb
call pgma
call pgmb
...
```

Na obrazovce se vypíše vždy o 1 vyšší číslo bez ohledu na to, který program voláme.

Nyní modifikujeme proceduru PROCP1 v modulu SPGMX tak, že konstantu *on v podmínce zaměníme na *off a vytvoříme znovu servisní program SPGMX. Zkoušku opakujeme ve druhé variantě. Tentokrát by se mělo přičítat číslo 1 pro každé volání samostatně: zvlášť pro

```
call pgma
```

a zvlášť pro

```
call pgmb
```