IBM

eserver

iSeries

ILE 概念

バージョン 5 リリース 3

SD88-5033-07
(英文原典：SC41-5606-07)

IBM

@server

iSeries

ILE 概念

バージョン 5 リリース 3

SD88-5033-07
(英文原典：SC41-5606-07)

お願い

本書、および本書で説明している製品をご使用になる前に 227 ページの『付録 D. 特記事項』をお読みください。

| 本書は、IBM OS/400 (プロダクト番号 5722-SS1) のバージョン 5、リリース 3、モディフィケーション・レベル 0
| に適用されます。また改訂版などで特に断りのない限り、それ以降のすべてのリリースおよびモディフィケーショ
| ン・レベルにも適用されます。このバージョンは、すべての RISC モデルで稼動するとは限りません。また、CISC
| モデルでは稼動しません。

| 本書は、SD88-5033-06 の改訂版です。

本マニュアルに関するご意見やご感想は、次の URL からお送りください。今後の参考にさせていただきます。

`http://www.ibm.com/jp/manuals/main/mail.html`

なお、日本 IBM 発行のマニュアルはインターネット経由でもご購入いただけます。詳しくは

`http://www.ibm.com/jp/manuals/` の「ご注文について」をご覧ください。

(URL は、変更になる場合があります)

お客様の環境によっては、資料中の円記号がバックスラッシュと表示されたり、バックスラッシュが円記号と表示されたりする場合があります。

原　典： SC41-5606-07
iSeries
ILE Concepts
Version 5 Release 3

発　行： 日本アイ・ビー・エム株式会社

担　当： ナショナル・ランゲージ・サポート

第1刷 2004.4

この文書では、平成明朝体™W3、平成明朝体™W9、平成角ゴシック体™W3、平成角ゴシック体™W5、および平成角ゴシック体™W7を使用しています。この(書体*)は、(財）日本規格協会と使用契約を締結し使用しているものです。フォントとして無断複製することは禁止されています。

注* 平成明朝体™W3、平成明朝体™W9、平成角ゴシック体™W3、
平成角ゴシック体™W5、平成角ゴシック体™W7

目次

本書について

本書では、OS/400® ライセンス・プログラムの統合言語環境 (Integrated Language Environment® : ILE) 体系に関する概念および用語について説明しています。本書で説明するトピックには、モジュール作成、バインディング、プログラムの実行とデバッグ、および例外処理が含まれます。

本書で説明する概念は、すべての ILE 言語に関連しています。各 ILE 言語によって、ILE 体系のインプリメントの方法は、多少異なる場合があります。本書で説明する概念が各言語でどのようにインプリメントされているかについては、各 ILE 言語の「プログラマーの手引き」をご参照ください。

本書は、すべての ILE 言語に直接関連している OS/400 機能についても説明しています。特に、バインディング、メッセージ処理、およびデバッグに関する共通情報について説明しています。

本書は、既存の OS/400 言語から ILE 言語への移行方法については説明していません。この情報は、各 ILE 高水準言語 (HLL) の「プログラマーの手引き」に示されています。

本書の対象読者

本書は、次の方々を対象にしています。

- アプリケーションまたはソフトウェア・ツールを開発するソフトウェアのベンダー
- iSeries サーバーで混合言語アプリケーションの開発の経験のある方
- 他のシステムのアプリケーション・プログラミングの経験はあるが、iSeries サーバーでは初めての方
- プログラムで共通プロシージャーを共用する場合、および共通プロシージャーを更新または拡張し、共通プロシージャーを使用するプログラムを再作成する必要がある方

基本的には 1 つの言語を使用してプログラムを作成する OS/400 アプリケーション・プログラマーの場合には、本書の最初の 4 つの章を読んで、ILE とその利点の概要を理解してください。その後、該当する ILE 言語の「プログラマーの手引き」を参照することにより、アプリケーション開発を行うことができます。

前提条件および関連情報

iSeries の技術情報を得るには、まず iSeries Information Center を参照してください。

Information Center には、以下の 2 つの方法でアクセスできます。

- 以下の Web サイトから :

  `http://www.ibm.com/eserver/iseries/infocenter`

- *iSeries V5R3 Information Center*, SK88-8055-03 CD-ROM から。この CD-ROM は、新規 iSeries ハードウェアまたは IBM OS/400 ソフトウェア・アップグレードをご注文いただくと同梱されています。この CD-ROM は、下記の IBM® Publications Center からも入手することができます。

  `http://www.ibm.com/shop/publications/order`

iSeries Information Center には、ソフトウェアおよびハードウェアのインストール、Linux、WebSphere®、Java™、高可用性、データベース、論理区画、CL コマンド、およびシステム・アプリケーション・プログラミング・インターフェース (API) などの、iSeries に関する新規情報および更新情報が記載されています。また、iSeries ハードウェアおよびソフトウェアの計画、トラブルシューティング、および構成を行う上で役立つアドバイスおよび検索機能も提供されています。

新規のハードウェアをご注文いただくと、*iSeries セットアップおよびオペレーション*, SK88-8058-02 が提供されます。 この CD-ROM には、IBM @server IBM e(logo)server iSeries Access for Windows および EZ セットアップ・ウィザードが含まれています。 iSeries Access Family は、iSeries™ サーバーに PC を接続するための、クライアントとサーバーの機能のパワフルなセットを提供します。 EZ セットアップ・ウィザードは、iSeries セットアップ・タスクの多くを自動化します。

その他の関連情報については、 231 ページの『参考文献』を参照してください。

第 1 章 統合化言語環境 (ILE) の概要

本章では、統合化言語環境 (Integrated Language Environment : ILE) モデルを定義し、ILE の利点、および従来のプログラム・モデルから ILE モデルがどのように発展してきたかについて記述します。

できるかぎり、記述は RPG または COBOL のプログラマーの観点で行い、また既存の iSeries サーバーの機能の用語を用いています。

ILE とは何か ?

ILE は、iSeries システムにおけるプログラム開発の強化の目的で設計されたツールのセットおよび関連するシステム・サポートです。

この新しいモデルの機能は、新しい ILE ファミリーのコンパイラーによって生成されたプログラムによってのみ活用することができます。 このファミリーには、ILE RPG、ILE COBOL、ILE C、ILE C++、および ILE CL が含まれます。

ILE の利点は何か ?

従来のプログラム・モデルに比べて、ILE は多くの利点をもっています。これらの利点には、バインディング、モジュール性、再使用可能コンポーネント、共通実行時サービス、共存性、およびソース・デバッガーが含まれます。さらに、リソース制御機能、言語間対話制御機能、コード最適化機能、C 環境、および将来的な基礎のそれぞれが改善されています。

バインディング

バインディングの利点は、プログラムの呼び出しに関連するオーバーヘッドの削減に役立つことです。モジュールのバインディングによって、呼び出しのスピードアップをはかることができます。従来の呼び出しの手段を使用することもできますが、新しい呼び出しの手段を使用する方が速くなります。 2 つの呼び出し方式を区別するために、従来方式を動的プログラム呼び出しまたは外部プログラム呼び出しと呼び、ILE 方式を静的プロシージャー呼び出しまたはバインド・プロシージャー呼び出しと呼びます。

バインディング機能、およびその結果として得られる呼び出しのパフォーマンスの向上によって、高度なモジュラー方式のアプリケーションを開発することが以前よりもはるかに実用的となります。ILE コンパイラーは、実行可能なプログラムを生成しません。その代わりに、モジュール・オブジェクト (*MODULE) を生成します。このモジュール・オブジェクトは、他のモジュールと結合 (バインド) して 1 つの実行可能単位、つまりプログラム・オブジェクト (*PGM) を形成します。

COBOL プログラムから RPG プログラムを呼び出すことができるのと同様に、ILE を使用して、異なる言語によって作成されたモジュールをバインドすることができます。したがって、RPG、COBOL、C、C++、および CL により個別に作成されたモジュールからなる 1 つの実行可能プログラムを作成することができます。

モジュール性

アプリケーション・プログラミングでモジュラー・アプローチを使用する利点は次のとおりです。

- コンパイル時間の短縮

  コンパイルするコードの部分が小さいほど、コンパイラーによる処理は速くなります。この利点は保守時に特に重要です。なぜなら、変更する必要があるのは 1 行または 2 行である場合が多いからです。2 行を変更した場合でも、2000 行の再コンパイルが必要になることがあります。これは、リソースの効率的な使用とは言えません。

  コードをモジュール化し、ILE のバインディング機能を使用すると、100 行または 200 行の再コンパイルで済む可能性があります。バインディングのステップが組み込まれるにしても、このようなプロセスはかなり速くなります。

- 保守の簡略化

  非常に大きなプログラムを更新する場合、プログラムのロジックがどうなっているのかを正確に把握することは困難です。当初のプログラマーがユーザー自身のスタイルとは異なるスタイルでプログラムを作成している場合、特にこの把握が困難です。コードをより小さな部分に分けて単一の機能を行うようにすれば、その内部の処理の把握は、きわめて容易になります。したがって、ロジックの流れがより明確になり、変更を行う場合に、望ましくない影響が生じる可能性を大幅に減らすことができます。

- テストの簡略化

  コンパイル単位を小さくすることによって、各機能を独立してテストすることができます。この分離化によって、テスト範囲の完全性が保証されます。すなわち、可能性のあるすべての入力とロジック・パスがテストされます。

- プログラミング・リソースの有効利用

  モジュール化によって作業の分割が容易になります。大きなプログラムを作成する場合、作業の分割は (不可能ではないとしても) 困難です。プログラムのすべての部分のコーディングは、初心者のプログラマーにとっては負担が大きすぎ、一方、熟練したプログラマーを用いるのは、スキルの無駄使いになることがあります。

- 他のプラットフォームからのコードの容易な移行

  他のプラットフォーム (たとえば、UNIX® など) で作成されたプログラムがモジュラー化されていることがよくあります。これらのモジュールは OS/400 に移行し、ILE プログラムに組み込むことができます。

再使用可能なコンポーネント

ILE により、ユーザー自身のプログラムに組み込むことができるルーチンのパッケージを選択することができます。 ILE 言語のいずれかで作成されたルーチンは、iSeries ILE コンパイラーのすべてのユーザーによって使用することができます。ユーザーは選択した言語でプログラムを作成できるので、ルーチンの広範囲な選択が可能になります。

これらのパッケージをユーザーに提供するために IBM および他社が使用している同じメカニズムを、ユーザーがユーザー自身のアプリケーションで使用することができます。各システムは、独自の標準ルーチンのセットを、任意の言語で作成することができます。

アプリケーションで既製のルーチンを使用できるだけではありません。好きな ILE 言語でルーチンを開発して、それらを他の ILE 言語のユーザーに販売することもできます。

共通実行時サービス

既製のコンポーネント (**バインド可能 API**) から選択したものが ILE の一環として提供されており、ユーザーのアプリケーションに組み込むことができます。これらの API は次のサービスを提供します。

日付時刻操作

メッセージ処理

数学ルーチン

画面処理に関するより広範な制御

動的ストレージ割り振り

将来、このセットにはさらにルーチンが追加され、サード・パーティー・ベンダーのルーチンも使用できるようになるはずです。

IBM では、ILE で提供される API の詳細を記述したオンライン情報を用意しています。iSeries Information Center の**プログラミング**・カテゴリーの中の *API* セクションを参照してください。

既存のアプリケーションとの共存

ILE プログラムは、既存の OPM プログラムとの共存が可能です。ILE プログラムは、OPM プログラムや他の ILE プログラムを呼び出すことができます。同様に、OPM プログラムは、ILE プログラムや他の OPM プログラムを呼び出すことができます。したがって、綿密な計画により、ILE への漸進的な移行が可能です。

ソース・デバッガー

ソース・デバッガーを用いて、ILE のプログラムおよびサービス・プログラムのデバッグを行うことができます。ソース・デバッガーについては 145 ページの『第 10 章 デバッグに関する考慮事項』を参照してください。

リソース制御の改善

ILE の導入前は、プログラムが使用するリソース (たとえば、オープン・ファイル) の有効範囲は、以下のいずれかにしか設定できませんでした (すなわち、リソースは以下のいずれかによって所有されました)。

リソースを割り振ったプログラム

ジョブ

多くの場合、この制約によってアプリケーションの設計担当者は妥協を余儀なくされます。

ILE は別の方法を提供します。ジョブの一部がリソースを所有することができます。この方法では、ILE 構成要素である**活動化グループ**を使用します。 ILE のもとでは、リソースの有効範囲を以下のいずれかに設定することができます。

プログラム

活動化グループ

ジョブ

共用オープン・データ・パスのシナリオ

共用オープン・データ・パス (ODP) は、ILE の新しい有効範囲指定のレベルにより、より良い制御が可能なリソースの例です。

iSeries サーバーでのアプリケーションのパフォーマンスを改善するために、顧客マスター・ファイルに対して、共用 ODP を使用することをプログラマーが決定したとします。このファイルは、受注アプリケーションと請求アプリケーションの両方で使用されます。

共用 ODP の有効範囲はジョブなので、いずれかのアプリケーションで予期しない問題がもう一方のアプリケーションによって引き起こされる可能性があります。実際、このような問題を回避するには、アプリケーションの開発者間の周到な調整が必要になります。アプリケーションを種々の業者から購入した場合には、問題の回避は不可能な場合もあり得ます。

どのような問題が起こる可能性があるでしょうか? 以下のシナリオについて考えてみましょう。

1. 顧客マスター・ファイルは、顧客番号をキーとして、顧客番号 A1、A2、B1、C1、C2、D1、D2 などのレコードが入っています。
2. オペレーターは、各マスター・ファイル・レコードを調べ、必要に応じてレコードを更新して、次のレコードを要求します。現在表示されているレコードは顧客番号 B1 のレコードです。
3. 電話が鳴ります。顧客 D1 は注文したいと思っています。
4. オペレーターは「受注処理」のファンクション・キーを押し、顧客 D1 の注文を処理して、マスター・ファイル画面に戻ります。
5. プログラムはまだ B1 のレコードを正しく表示していますが、オペレーターが次のレコードを要求すると表示されるのは、どのレコードでしょうか ?

   表示されるのは D2 です。共用 ODP の有効範囲はジョブなので、受注アプリケーションがレコード D1 を読み取る時点で、現在のファイル位置が変更されています。したがって、次のレコード要求は、D1 の後の次のレコードを意味します。

ILE のもとでは、請求業務専用の活動化グループ内でマスター・ファイルの保守を実行することによってこの問題を回避することができます。同様に、受注アプリケーションも自分自身の活動化グループ内で実行することになります。各アプリケーションは、依然として共用 ODP の利点を活用できますが、関連の活動化グループにより自分自身の共用 ODP を持つことになります。このレベルの有効範囲指定を使用して、この例のような干渉を回避することができます。

リソースの有効範囲を活動化グループに設定することによって、プログラマーは、1つのジョブ内で実行中の他のアプリケーションから独立して実行するアプリケーシ

ョンを自由に開発することができます。これによって、必要な調整作業が減り、既存のアプリケーション・パッケージにドロップイン拡張を書き込む能力も向上します。

コミットメント制御のシナリオ

共用オープン・データ・パス (ODP) の有効範囲をアプリケーションに設定するこの機能は、コミットメント制御の場合に役立ちます。

コミットメント制御のもとでファイルを使用し、さらに共用 ODP を使用する必要もあると想定します。ILE を使用しない場合、1 つのプログラムがコミットメント制御のもとでファイルをオープンすると、同じジョブのプログラムはすべてコミットメント制御のもとでファイルをオープンしなければなりません。これは、コミットメント機能が 1 つまたは 2 つのプログラムでのみ必要になる場合でも当てはまります。

このような状態で起こり得る 1 つの問題は、ジョブ内のいずれかのプログラムがコミット操作を指示すると、すべての更新がコミットされてしまうことです。 処理中のアプリケーションに論理的に無関係な更新もコミットされます。

この問題は、コミットメント制御が必要なアプリケーションの各部分を個別の活動化グループで実行することによって回避することができます。

言語間対話の制御の改善

ILE を使用しない場合、iSeries サーバーでプログラムがどのように実行されるかは、以下の組み合わせによって異なります。

言語標準 (たとえば、COBOL や C の ANSI 標準)

コンパイラーの開発者

言語を混合する場合、上記の組み合わせによって問題が発生することがあります。

混合言語のシナリオ

ILE によって導入される活動化グループなしでは、OPM 言語間の対話を予測することは困難です。 ILE 活動化グループにより、この問題は解決することができます。

例として、COBOL と他の言語との混合によって起こる問題について考えてみましょう。COBOL の言語標準には、**実行単位**と呼ばれる概念が含まれています。実行単位は複数のプログラムをグループ化し、特定の状況で単一のエンティティーとして機能するようにします。これはきわめて有用な機能になり得ます。

3 つの ILE COBOL プログラム (PRGA、PRGB、および PRGC) が 1 つの小さなアプリケーションを形成していると想定します。このアプリケーションでは、PRGA が PRGB を、ついで PRGB が PRGC を呼び出します (6 ページの図 1 を参照)。ILE COBOL の規則では、これらの 3 つのプログラムは同じ実行単位に入っています。結果として、これらのいずれか 1 つのプログラムが終了した場合には、3 つのプログラムをすべて終了し、制御は PRGA の呼び出し元に戻る必要があります。

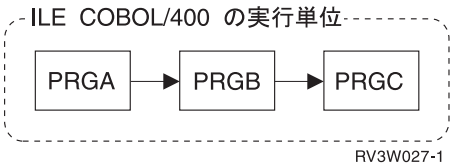

図 1. 実行単位における 3 つの ILE COBOL プログラム

RPG プログラム (RPG1) をこのアプリケーションに組み込み、その RPG1 は COBOL プログラム PRGB によって呼び出されるものとします (図 2 を参照)。RPG プログラムは、最終レコード (LR) 標識をオンにして戻るまで、その変数、ファイル、および他のリソースが変更されないものと想定します。

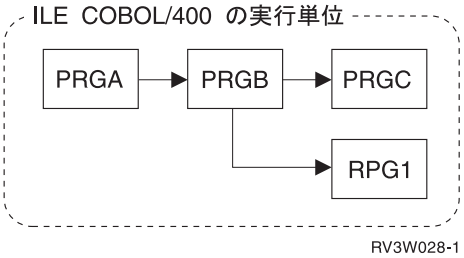

図 2. 実行単位における 3 つの ILE COBOL プログラムと 1 つの ILE RPG プログラム

ただし、RPG1 が RPG で書かれていても、RPG1 が COBOL 実行単位の一部として実行される場合には、すべての RPG のセマンティクスが適用できるとは限りません。実行単位が終了すると、RPG1 は LR 標識の設定に関係なく消失します。多くの場合、この状態は望ましい状態ではありません。たとえば、RPG1 が送り状番号の発行を制御するユーティリティー・プログラムの場合には、この状態は受け入れられません。

この状態は、COBOL 実行単位とは別の活動化グループで RPG プログラムを稼働させることによって、避けることができます (図 3 を参照)。ILE COBOL 実行単位自体が 1 つの活動化グループです。

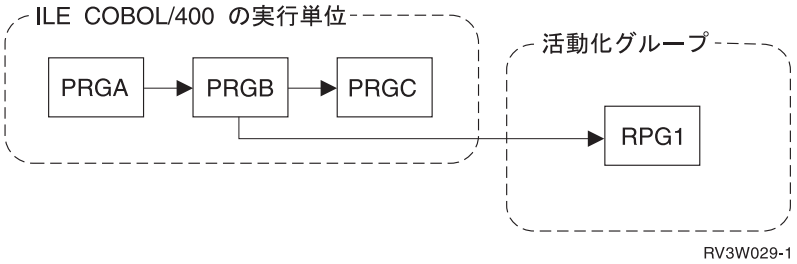

図 3. 別の活動化グループにおける ILE RPG プログラム

OPM 実行単位と ILE 実行単位の違いについては、「ILE COBOL プログラマーの手引き」 を参照してください。

コード最適化の改善

ILE 変換プログラムは、オリジナル・プログラム・モデル (OPM) よりも多くの種類の最適化を行います。各コンパイラーも何らかの最適化を行いますが、OS/400 における最適化の大部分は変換プログラムによって行われます。

ILE 使用可能コンパイラーは、モジュールを直接生成しません。最初にモジュールの中間形式を生成し、次にその中間コードを実行可能な命令に変換する ILE 変換プログラムを呼び出します。

C 言語に対する環境の改善

C 言語は、ツールのビルダーにとっては主要な言語です。このため、 C 言語を採用すれば、より多くの最新のアプリケーション開発ツールを OS/400 に移行できることになります。ユーザーにとって、このことは以下のような機能の選択の幅が広がることを意味します。

CASE ツール

第 4 世代言語 (4GL)

他のプログラミング言語

エディター

デバッガー

将来への基礎

ILE が提供する利点および機能は、将来さらに重要になるはずです。将来、ILE コンパイラーの機能が大幅に拡張されるはずです。オブジェクト指向プログラミング言語およびビジュアルなプログラミング・ツールへの移行に伴って、ILE の必要性はより明白になります。

プログラミング方式は、ますます高度にモジュール化された方法に基づくようになります。アプリケーションは、きわめて多くの小さな再使用可能なコンポーネントを結合することによって作成されます。これらのコンポーネント相互間で制御を迅速に渡せない場合、アプリケーションは機能しなくなります。

ILE の沿革

ILE は OS/400 プログラム・モデルの発展の 1 段階です。各段階は、アプリケーション・プログラマーの変化するニーズに合うように発展してきました。

AS/400® システムが最初に導入された時点で提供されたプログラミング環境は、オリジナル・プログラム・モデル (OPM) と呼ばれます。バージョン 1 リリース 2 で、拡張プログラム・モデル (EPM) が導入されました。

オリジナル・プログラム・モデルの記述

iSeries サーバー上のアプリケーションの開発者は、ソース・コードをソース・ファイルに入れ、そのソースをコンパイルします。コンパイルが成功すると、プログラム・オブジェクトが作成されます。プログラムの作成および実行のために、OS/400 によって提供される機能、処理、および規則のセットは、**オリジナル・プログラム・モデル (OPM)** として知られています。

OPM コンパイラーは、プログラム・オブジェクトを生成する時に、追加のコードを生成します。この追加のコードは、プログラム変数を初期設定し、特定言語の特殊な処理に必要なコードを提供します。特殊な処理には、プログラムによって想定される入力パラメーターの処理が含まれます。プログラムの実行開始の時点で、追加されたコンパイラー生成コードがそのプログラムの開始点 (入り口点) になります。

プログラムは一般的に、OS/400 が呼び出し要求を検出すると活動化されます。実行時における他のプログラムの呼び出しは、**動的プログラム呼び出し**と呼ばれます。動的プログラム呼び出しには、かなりのリソースが必要になることがあります。しばしば、アプリケーション開発者は、動的プログラム呼び出しの数を最小限にするために、少数の大きなプログラムからなるアプリケーションを設計します。

図 4 は、OPM とオペレーティング・システムとの間の関係を示しています。図に示されているように、RPG、COBOL、CL、BASIC、および PL/I はすべてこのモデルで作動します。

OPM の境界を示す破線は、OPM が OS/400 の統合された部分であることを示しています。この統合は、コンパイラー作成者によって通常提供される多くの機能がオペレーティング・システムに組み込まれていることを意味します。その結果の呼び出し規則の標準化によって、1 つの言語によって作成されたプログラムは、他の言語によって作成されたプログラムを自由に呼び出すことが可能になります。たとえば、RPG で作成されたアプリケーションには、一般的に、ファイル一時変更、ストリング処理、あるいはメッセージの送信などを行う多くの CL プログラムが含まれています。

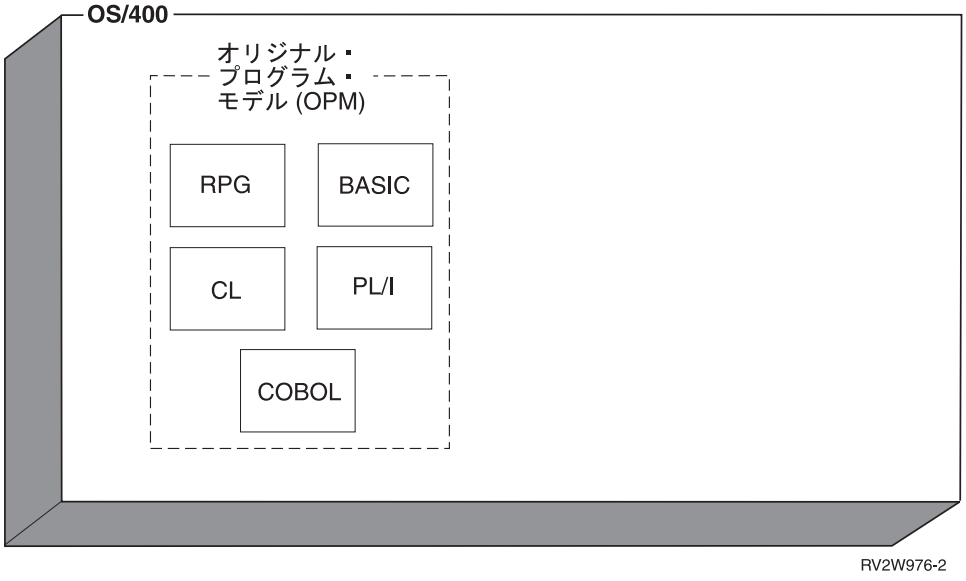

図 4. OPM と OS/400 の関係

OPM の基本特性

以下のリストは、OPM の基本特性を示しています。

- 従来の RPG プログラムおよび COBOL プログラムに対する適合性

OPM は、従来の RPG や COBOL のプログラム、つまり比較的大きな多機能プログラムのサポートに適しています。

- 動的バインディング

  プログラム A がプログラム B を呼び出す必要がある場合、ただ呼び出すだけです。この動的プログラム呼び出しは単純で強力な機能です。実行時に、オペレーティング・システムがプログラム B を見つけ、それを使用できる権限がユーザーにあることを確認します。

  OPM プログラムは入り口点を 1 つだけ持つのに対し、ILE プログラムでは各プロシージャーが 1 つの入り口点となり得ます。

- 限定されたデータ共用

  OPM において、内部プロシージャーは変数をプログラム全体と共用しなければならないのに対し、ILE では、各プロシージャーはそれぞれ独自の有効範囲がローカルで指定された変数を持つことができます。

拡張プログラム・モデルの記述

OPM は、ここでも有用です。ただし、OPM は、C などの言語で定義されているプロシージャーへの直接サポートは行いません。**プロシージャー**は、特定の作業を実行して呼び出し元に戻る自己完結型の一連の高水準言語 (HLL) ステートメントです。言語によって、プロシージャーの定義方法は異なります。C の場合、プロシージャーは関数と呼ばれます。

言語で、コンパイル単位間でのプロシージャー呼び出しを定義することができるように、または iSeries サーバーで実行するためにローカル変数を使用してプロシージャーを定義することができるように、OPM が拡張されました。拡張された OPM は、**拡張プログラム・モデル (EPM)** と呼ばれます。 ILE より前は、EPM が、Pascal および C などのプロシージャー・ベースの言語用の一時的なソリューションとして機能していました。

iSeries サーバーでは、EPM コンパイラーは提供しなくなりました。

統合化言語環境 (ILE) の記述

10 ページの図 5 に示したように、ILE は OPM と同様に、OS/400 に緊密に統合されています。 ILE は EPM と同じタイプのプロシージャー・ベースの言語のサポートを提供しますが、完全性と一貫性がはるかに向上しました。ILE は、RPG および COBOL などの従来の言語だけでなく、将来の言語開発にも対応できるように設計されています。

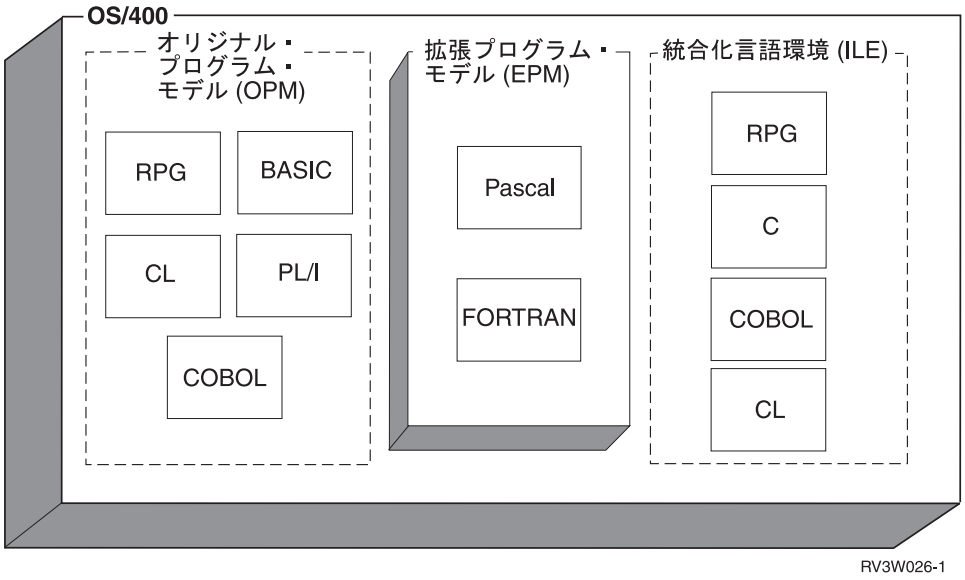

図 5. OPM、EPM、および ILE と OS/400 の関係

プロシージャー・ベースの言語の基本特性

プロシージャー・ベースの言語には以下の特性があります。

- 有効範囲がローカルの変数

  有効範囲がローカルの変数は、それを定義しているプロシージャーでのみ認識されます。有効範囲がローカルの変数の場合、同じ名前をもつ 2 つの変数を定義して、2 つの別個のデータを参照することができます。たとえば、変数 COUNT をサブルーチン CALCYR では 4 桁の長さとして、サブルーチン CALCDAY では 6 桁の長さとして定義することができます。

  複数の異なるプログラムにコピーする必要があるサブルーチンを作成する場合、有効範囲がローカルの変数を使用すると便利です。有効範囲がローカルの変数がない場合、プログラマーは、サブルーチンの名前に基づく変数の命名などの体系を使用する必要があります。

- 自動変数

  自動変数は、1 つのプロシージャーに入るたびに作成されます。自動変数は、プロシージャーから出ると破棄されます。

- 外部変数

  外部データはプログラム間でデータを共用する 1 つの方法です。プログラム A が 1 つのデータ項目を外部変数として宣言した場合、プログラム A は、そのデータを共用する必要がある他のプログラムにそのデータ項目を**エクスポート**すると言います。プログラム D は、プログラム B および C に関係なく、その項目を**インポート**することができます。インポートとエクスポートの詳細については 14 ページの『モジュール・オブジェクト』を参照してください。

- 複数の入り口点

COBOL および RPG のプログラムは、入り口点を 1 つだけ持っています。COBOL プログラムでは、入り口点は PROCEDURE DIVISION の開始点です。RPG プログラムでは、最初のページ (1P) 出力です。 これは、OPM がサポートしているモデルです。

一方、プロシージャー・ベースの言語の場合、複数の入り口点が可能です。たとえば、C プログラムは他のプログラムによって使用されるサブルーチンだけで構成することができます。このようなプロシージャーは、インポートする他のプログラムに、必要なら関連データとともに、エクスポートすることができます。

ILE では、このタイプのプログラムは、**サービス・プログラム**と呼びます。 サービス・プログラムには任意の ILE 言語のモジュールを組み込むことができます。サービス・プログラムは、概念的には、Windows® または OS/2® のダイナミック・リンク・ライブラリー (DLL) に類似しています。サービス・プログラムについては 18 ページの『サービス・プログラム』で詳細に記述します。

- 頻繁な呼び出し

  プロシージャー・ベースの言語では、性質上呼び出しが多用されます。EPM は呼び出しのオーバーヘッドを最小限にするためのいくつかの機能を提供していますが、個別にコンパイルされた単位間の呼び出しには、依然としてかなり高いオーバーヘッドが必要です。ILE では、このタイプの呼び出しが大幅に改善されました。

第 2 章 ILE の基本概念

表 1 は、オリジナル・プログラム・モデル (OPM) と統合化言語環境 (ILE) モデルを対比させたものです。本章では、この表にリストされた類似点と相違点について記述します。

表 1. OPM と ILE の類似点と相違点

OPM	ILE
プログラム	プログラム
	サービス・プログラム
コンパイルにより実行可能なプログラムが生成される	コンパイルにより実行不能なモジュール・オブジェクトが生成される
コンパイル、実行	コンパイル、バインド、実行
各言語ごとにシミュレートされた実行単位	活動化グループ
動的プログラム呼び出し	動的プログラム呼び出し
	静的プロシージャー呼び出し
単一言語フォーカス	混合言語フォーカス
言語固有エラー処理	共通エラー処理
	言語固有エラー処理
OPM デバッガー	ソース・レベル・デバッガー

ILE プログラムの構造

ILE プログラムは 1 つ以上のモジュールからなります。モジュールは、1 つ以上のプロシージャーからなります (図 6 を参照)。

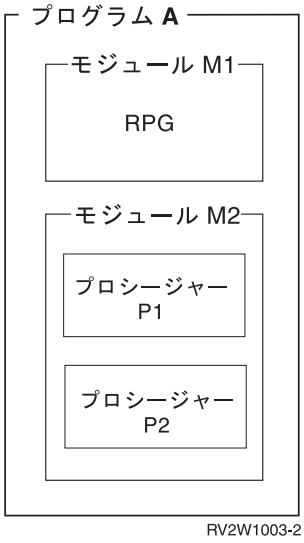

図 6. ILE プログラムの構造

プロシージャー

プロシージャーは、特定の処理を実行して呼び出し元に戻る自己完結型の一連の高水準言語ステートメントです。たとえば、ILE C 関数は ILE プロシージャーです。

モジュール・オブジェクト

モジュール・オブジェクトは、ILE コンパイラーの出力で*実行不能* なオブジェクトです。モジュール・オブジェクトは、システムに対しシンボル *MODULE によって示されます。モジュール・オブジェクトは、実行可能な ILE オブジェクトを作成するための基本的な構成ブロックです。これは、ILE と OPM の間の重要な相違点です。OPM コンパイラーの出力は、*実行可能* なプログラムです。

モジュール・オブジェクトは、1 つ以上のプロシージャーの指定とデータ項目の指定により構成することができます。1 つのモジュール内のプロシージャーまたはデータ項目に、別の ILE オブジェクトから直接アクセスすることができます。 他の ILE オブジェクトから直接アクセスできるプロシージャーおよびデータ項目のコーディングの詳細については、ILE HLL の「プログラマーの手引き」を参照してください。

ILE RPG、ILE COBOL、ILE C、および ILE C++ は、すべて以下の共通の概念を備えています。

- エクスポート

  エクスポートは、モジュール・オブジェクトにコーディングされたプロシージャーまたはデータ項目の名前であり、他の ILE オブジェクトが使用することができます。エクスポートは、その名前および関連するタイプ (プロシージャーまたはデータ) によって識別されます。

  エクスポートは**定義**とも呼ばれます。

- インポート

  インポートは、現行モジュール・オブジェクトで定義されていないプロシージャーまたはデータ項目の名前を使用または参照することです。インポートはその名前および関連するタイプ (プロシージャーまたはデータ) によって識別されます。

  インポートは**参照**とも呼ばれます。

モジュール・オブジェクトは ILE 実行可能オブジェクトの基本的な構成ブロックです。したがって、モジュール・オブジェクトの作成時点で、以下のデータおよびプロシージャーも生成されることがあります。

- デバッグ・データ

  デバッグ・データは、実行中の ILE オブジェクトのデバッグに必要なデータです。このデータはオプションです。

- プログラム入り口プロシージャー (PEP)

  プログラム入り口プロシージャーは、コンパイラー生成コードで、動的プログラム呼び出し時に ILE プログラムの入り口点になります。これは OPM プログラムの入り口点として提供されるコードに類似しています。

- ユーザー入り口プロシージャー (UEP)

プログラマーによって作成される**ユーザー入り口プロシージャー**は、動的プログラム呼び出しのターゲットです。これは PEP から制御を渡されるプロシージャーです。C プログラムの main() 関数は、ILE において、そのプログラムの UEP になります。

図 7 は、モジュール・オブジェクトの概念図を示しています。この例で、モジュール・オブジェクト M1 は、2 つのプロシージャー (Draw_Line と Draw_Arc) および 1 つのデータ項目 (rtn_code) をエクスポートしています。モジュール・オブジェクト M1 は、Draw_Plot というプロシージャーをインポートしています。このモジュール・オブジェクトには、PEP に対応する UEP (プロシージャー Draw_Arc)、およびデバッグ・データが含まれています。

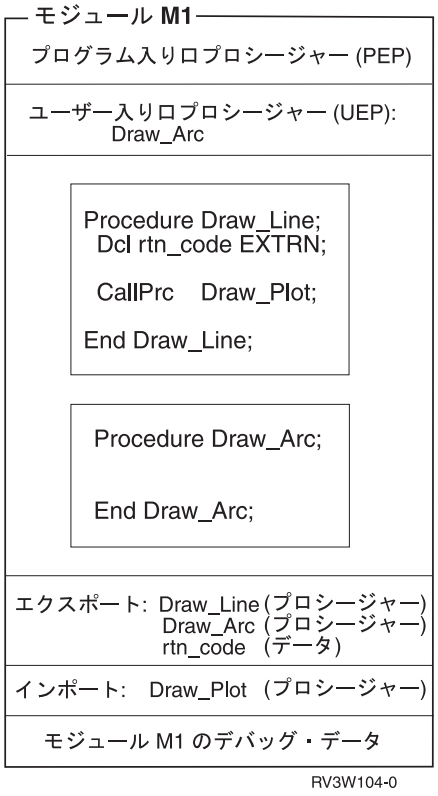

図 7. モジュールの概念図

*MODULE オブジェクトの特性

- *MODULE オブジェクトは ILE コンパイラーからの出力です。
- ILE 実行可能オブジェクトの基本的な構成ブロックです。
- 実行可能オブジェクトではありません。
- PEP を定義することができます。
- PEP を定義すると、UEP も定義されます。
- プロシージャー名およびデータ項目名をエクスポートすることができます。
- プロシージャー名およびデータ項目名をインポートすることができます。
- デバッグ・データを定義することができます。

ILE プログラム

ILE プログラムと OPM プログラムの共通の特性は次のとおりです。

- プログラムは、動的プログラム呼び出しにより制御を渡されます。
- プログラムへの入り口点はただ 1 つです。
- システムはプログラムをシンボル *PGM によって識別します。

ILE プログラムには、OPM プログラムにはない以下の特性があります。

- ILE プログラムは 1 つ以上のコピーされたモジュール・オブジェクトから作成されます。
- コピーされた 1 つ以上のモジュールは 1 つの PEP を含むことができます。
- ILE プログラム・オブジェクトの PEP として、どのモジュールの PEP を使用するかを制御することができます。

  プログラムの作成 (CRTPGM) コマンドを指定する際に、ENTMOD パラメーターによって、PEP を含むどのモジュールをプログラムの入り口点とするかを選択することができます。

  そのプログラムの入り口点として選択されないモジュールに関連する PEP は無視されます。モジュールの他のすべてのプロシージャーおよびデータ項目は、指定に従って使用されます。その PEP だけが無視されます。

動的プログラム呼び出しが ILE プログラムに対して行われると、プログラム作成時に選択されたモジュールの PEP に制御が渡されます。PEP は関連する UEP を呼び出します。

ILE プログラム・オブジェクトを作成すると、デバッグ・データを含むコピーされたモジュールに関連するプロシージャーだけが、ILE デバッガーによってデバッグ可能です。デバッグ・データは ILE プログラムの実行パフォーマンスに影響を与えません。

17 ページの図 8 は、ILE プログラム・オブジェクトの概念図を示しています。プログラム PGMEXAMP が呼び出されると、コピーされたモジュール・オブジェクト M3 に定義されているこのプログラムの PEP に制御が渡されます。コピーされたモジュール M2 にも PEP が定義されていますが、このプログラムによって無視され、使用されません。

このプログラム例では、新しい ILE デバッガーに必要なデータは、2 つのモジュール M1 と M3 だけにあります。 モジュール M2 と M4 からのプロシージャーは、新しい ILE デバッガーを使用してデバッグすることはできません。

インポートされるプロシージャー `print` と `SIN` は、それぞれサービス・プログラム PRINTS と MATHFUNC からエクスポートされるプロシージャーとして解決されます。

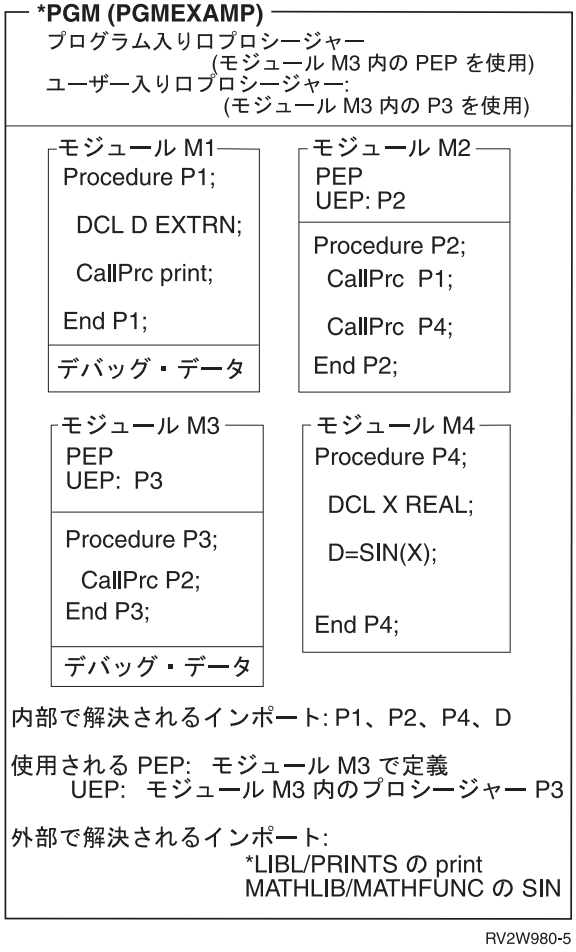

図 8. ILE プログラムの概念図

ILE *PGM オブジェクトの特性

- 任意の ILE 言語から 1 つ以上のモジュールがコピーされ、*PGM オブジェクトが作成されます。
- プログラムの作成者は、どのモジュールの PEP をプログラムの唯一の PEP にするのかを制御することができます。
- 動的プログラム呼び出し時に、プログラムの PEP として選択されたモジュールの PEP に実行の制御が渡されます。
- 選択された PEP に関連する UEP が、プログラムへのユーザーの入り口点になります。
- プロシージャー名およびデータ項目名はプログラムからエクスポートできません。
- プロシージャーやデータ項目名をインポートすることができるのは、モジュールおよびサービス・プログラムからです。プログラム・オブジェクトからのインポートはできません。サービス・プログラムについては、18 ページの『サービス・プログラム』を参照してください。
- モジュールはデバッグ・データを持つことができます。
- プログラムは実行可能なオブジェクトです。

サービス・プログラム

サービス・プログラムは、他の ILE プログラムまたはサービス・プログラムにより、容易に、しかも直接にアクセスできる実行可能プロシージャーと使用可能なデータ項目の集合です。多くの点で、サービス・プログラムはサブルーチン・ライブラリーまたはプロシージャー・ライブラリーに類似しています。

サービス・プログラムは、他の ILE オブジェクトに必要な共通サービスを提供します。そのため、サービス・プログラムと呼ばれます。OS/400 によって提供される一連のサービス・プログラムの例は、特定の言語用の実行時プロシージャーです。 これらの実行時プロシージャーには、しばしば数学プロシージャーおよび共通入出力プロシージャーなどの項目が含まれます。

サービス・プログラムの**共通インターフェース**は、他の ILE オブジェクトによってアクセス可能なエクスポートされるプロシージャーおよびデータ項目の名前からなります。サービス・プログラムからのエクスポートが可能な項目は、モジュール・オブジェクトからエクスポートされるサービス・プログラムを構成する項目だけです。

プログラマーは、どのプロシージャーまたはデータ項目が他の ILE オブジェクトによって認識可能かを指定することができます。したがって、他のどの ILE オブジェクトによっても使用できない、隠れたまたは専用のプロシージャーやデータをサービス・プログラムに入れることができます。

サービス・プログラムを更新する場合、その更新されたサービス・プログラムを使用する他の ILE プログラムまたはサービス・プログラムを再作成せずに行うことができます。サービス・プログラムに変更を行うプログラマーは、変更によって既存のサポートとの互換性が保たれるかどうかを制御します。

互換性のある変更の制御のために ILE が提供する方法は、**バインド・プログラム (バインダー) 言語**の使用です。 バインド・プログラム言語によって、エクスポート可能なプロシージャー名とデータ項目名のリストを定義することができます。プロシージャーとデータ項目の名前、およびそれらの名前のバインド・プログラム言語での指定順序から、**シグニチャー**が生成されます。サービス・プログラムに対して互換性のある変更を行うためには、新しいプロシージャーまたはデータ項目の名前を、エクスポート・リストの末尾に追加しなければなりません。シグニチャー、バインド・プログラム言語、およびサービス・プログラムへの投資の保護の詳細については 90 ページの『バインド・プログラム言語』を参照してください。

19 ページの図 9 はサービス・プログラムの概念図を示しています。サービス・プログラムを構成しているモジュールは、 17 ページの図 8 の ILE プログラム・オブジェクト PGMEXAMP を構成しているモジュールのセットと同じである点に注意してください。 サービス・プログラム SPGMEXAMP に関する前のシグニチャー Sigyy には、プロシージャー P3 と P4 の名前が入っています。サービス・プログラムへの上方への互換性がある変更が行われた後、現行シグニチャー Sigxx には、プロシージャー P3 と P4 の名前だけでなく、データ項目 D の名前も含まれます。プロシージャー P3 または P4 を使用する他の ILE プログラムまたはサービス・プログラムは、再作成する必要はありません。

サービス・プログラムのモジュールに PEP があっても、これらの PEP は無視されます。サービス・プログラム自体に PEP はありません。したがって、プログラム・オブジェクトと異なり、サービス・プログラムを動的に呼び出すことはできません。

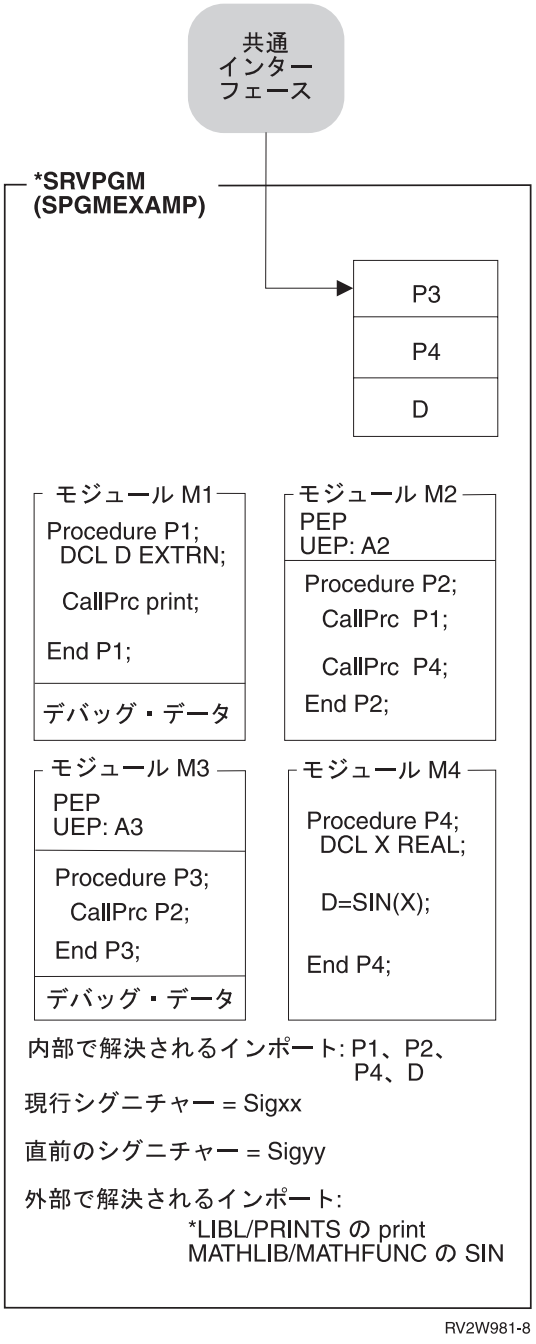

図 9. ILE サービス・プログラムの概念図

ILE *SRVPGM オブジェクトの特性

- ILE 言語から 1 つ以上のモジュールがコピーされ、*SRVPGM オブジェクトが作成されます。

- PEP はサービス・プログラムには関連付けられません。PEP がないので、サービス・プログラムの動的プログラム呼び出しは無効です。モジュールの PEP は無視されます。
- 他の ILE プログラムまたはサービス・プログラムは、共通インターフェースによって識別されるこのサービス・プログラムのエクスポートを使用することができます。
- シグニチャーが、サービス・プログラムからエクスポートされるプロシージャーおよびデータ項目の名前から生成されます。
- サービス・プログラムは、前のシグニチャーがサポートされている限り、そのサービス・プログラムを使用する ILE プログラムまたはサービス・プログラムに影響を与えずに置き換えることができます。
- モジュールはデバッグ・データを持つことができます。
- サービス・プログラムは実行可能なプロシージャーおよびデータ項目の集合です。
- ウイーク・データは、活動化グループへのみエクスポートすることができます。それは、サービス・プログラムからエクスポートされる共通インターフェースにはなれません。ウィーク・データについては 88 ページの『インポートおよびエクスポートの概念』のエクスポートを参照してください。

バインディング・ディレクトリー

バインディング・ディレクトリーには、ILE プログラムまたはサービス・プログラムの作成に必要なモジュールおよびサービス・プログラムの名前が入っています。バインディング・ディレクトリーにリストされているモジュールまたはサービス・プログラムは、現在未解決であるインポート要求を満足するエクスポートを提供する場合にのみ使用されます。バインディング・ディレクトリーは、シンボル *BNDDIR によってシステムが識別するシステム・オブジェクトです。

バインディング・ディレクトリーはオプションです。バインディング・ディレクトリーを使用する理由は、利便性とプログラム・サイズです。

- バインディング・ディレクトリーは、ユーザー自身の ILE プログラムまたはサービス・プログラムの作成に必要なモジュールまたはサービス・プログラムをパッケージ化する便利な方法を提供します。たとえば、1 つのバインディング・ディレクトリーに、数学関数を提供するすべてのモジュールおよびサービス・プログラムを入れることができます。これらの関数のいくつかを使用する必要がある場合、使用する各モジュールまたはサービス・プログラムではなく、1 つのバインディング・ディレクトリーを指定するだけで済みます。

  注: バインディング・ディレクトリーに入っているモジュールまたはサービス・プログラムが多いほど、プログラムのバインドに時間がかかります。 したがって、バインディング・ディレクトリーには必要なモジュールまたはサービス・プログラムのみを入れる必要があります。
- バインディング・ディレクトリーによって、使用しないモジュールまたはサービス・プログラムを指定せずに済むので、プログラム・サイズを縮小することができます。

バインディング・ディレクトリーの項目に対する制約はほとんどありません。モジュールまたはサービス・プログラムの名前は、そのオブジェクトがまだ存在しない場合にも、バインディング・ディレクトリーに追加することができます。

バインディング・ディレクトリーに使用できる CL コマンドのリストについては 223 ページの『付録 C. ILE オブジェクトに使用される CL コマンド』を参照してください。

図 10 は、バインディング・ディレクトリーの概念図を示しています。

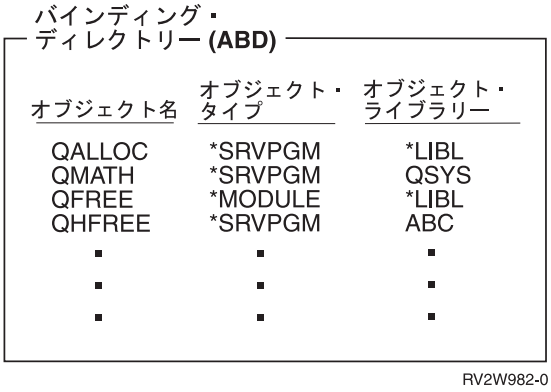

図 10. バインディング・ディレクトリーの概念図

*BNDDIR オブジェクトの特性

- ILE プログラムまたはサービス・プログラムの作成に必要なサービス・プログラムおよびモジュールの名前をグループ化するのに便利な方法です。
- バインディング・ディレクトリーの項目は名前だけなので、リストされているオブジェクトがシステムに存在している必要はありません。
- 有効なライブラリー名は *LIBL または特定のライブラリー名だけです。
- リスト内のオブジェクトはオプションです。指定されたオブジェクトは、未解決のインポートが存在し、かつ、指定されたオブジェクトがその未解決のインポート要求を満たすエクスポートを提供する場合にのみ使用されます。

バインディング・ディレクトリーの処理

バインディングでは、以下の順序で処理が行われます。

1. MODULE パラメーターで指定されたすべてのモジュールが検査されます。バインド・プログラムは、そのオブジェクトによってインポートおよびエクスポートされた記号のリストを判別します。検査されたモジュールは、リストされた順に、作成中のプログラムにバインドされます。
2. BNDSRVPGM パラメーターで指定されたすべてのサービス・プログラムが、リストされた順に検査されます。サービス・プログラムは、インポートを解決するために必要な場合にのみバインドされます。
3. BNDDIR パラメーターで指定されたすべてのバインディング・ディレクトリーが、リストされた順に処理されます。これらのバインディング・ディレクトリーでリストされたすべてのオブジェクトは、リストされた順に検査されますが、インポートを解決するために必要な場合にのみバインドされます。バインディング・ディレクトリー内の重複した項目は、無視されます。

4. 各モジュールには、**参照システム・オブジェクト**のリストがあります。このリストは、バインディング・ディレクトリーをそのままリストしたものです。バインドされたモジュールから取得した参照システム・オブジェクトの処理は、最初のモジュールから取得したすべての参照システム・オブジェクトが最初に処理され、次に 2 番目のモジュールから取得したオブジェクトが処理され、さらにその次のモジュールから取得したオブジェクトが処理される、という順序で行われます。これらのバインディング・ディレクトリーでリストされたオブジェクトは、必要なものについてのみ、リストされた順に検査され、必要な場合にのみバインドされます。この処理は、OPTION(*UNRSLVREF) が使用されている場合であっても、未解決のインポートが存在する間のみ継続されます。つまり、すべてのインポートが解決されると、オブジェクトの処理は停止します。

オブジェクトの検査時、作成中のプログラムにそのオブジェクトが最終的にバインドされない場合でも、メッセージ CPD5D03「記号の定義が重複しています」が出力されることがあります。

モジュールには通常、そのモジュールのソース・コードにはないインポートが含まれていることに注意してください。これらは、コンパイラーによって追加され、サービス・プログラムからのランタイム・サポートを必要とするさまざまな言語機能をインプリメントします。このようなインポートを見るためには、DSPMOD DETAIL(*IMPORT) を使用してください。

あるモジュールまたはサービス・プログラムに関してインポートまたはエクスポートされた記号のリストを見るためには、CRTPGM または CRTSRVPGM DETAIL(*EXTENDED) リストの「Binder Information Listing (バインダー情報リスト)」セクションを参照してください。このセクションには、バインディング中に検査されたオブジェクトがリストされています。

作成中のプログラムまたはサービス・プログラムにバインドされたモジュールまたはサービス・プログラム・オブジェクトは、CRTPGM または CRTSRVPGM DETAIL(*EXTENDED) リストの「Binder Information Listing (バインダー情報リスト)」セクションで表示されます。また、オブジェクトが作成された後で、DSPPGM または DSPSRVPGM コマンドの DETAIL(*MODULE) を使用して、バインドされた *MODULE オブジェクトを見たり、DETAIL(*SRVPGM) を使用して、バインドされた *SRVPGM オブジェクトのリストを見たりすることもできます。

DSPMOD DETAIL(*REFSYSOBJ) を使用すると、バインディング・ディレクトリーである、参照システム・オブジェクトのリストを見ることができます。これらのバインディング・ディレクトリーには、一般的に、オペレーティング・システムまたは言語のランタイム・サポートによって提供されるサービス・プログラム API の名前が含まれています。これにより、プログラマーがコマンドで特別な指定を行わなくても、モジュールをその言語ランタイム・サポートやシステム API にバインドできるようになります。

バインド・プログラム機能

バインド・プログラムの機能は、多少異なりますが、リンケージ・エディターによって提供される機能に類似しています。**バインド・プログラム**は、指定されたモジュールからのプロシージャー名およびデータ項目名に関するインポート要求を処理します。次にバインド・プログラムは、指定されたモジュール、サービス・プログラム、およびバインディング・ディレクトリーを調べて一致するエクスポートを探します。

ILE プログラムまたはサービス・プログラムの作成時に、バインド・プログラムは以下のタイプのバインディングを行います。

- コピーによるバインド

  ILE プログラムまたはサービス・プログラムを作成するために、以下のモジュールがコピーされます。

  モジュール・パラメーターに指定されたモジュール

  未解決のインポートに対するエクスポートを提供するバインディング・ディレクトリーから選択されたモジュール

  コピーされたモジュールで使用される必要なプロシージャーおよびデータ項目の物理アドレスは、ILE プログラムまたはサービス・プログラムの作成時に確立されます。

  たとえば 19 ページの図 9 で、モジュール M3 のプロシージャー P3 がモジュール M2 のプロシージャー P2 を呼び出します。 モジュール M2 のプロシージャー P2 の物理アドレスは、プロシージャー P3 に知らされるので、このアドレスに直接アクセスすることができます。

- 参照によるバインド

  未解決のインポート要求に対してエクスポートを提供するサービス・プログラムへのシンボリック・リンクは、作成されたプログラムまたはサービス・プログラムに保管されます。シンボリック・リンクは、エクスポートを提供するサービス・プログラムを参照します。 リンクは、そのサービス・プログラムがバインドされるプログラム・オブジェクトが活動化された時点で、物理アドレスに変換されます。

  19 ページの図 9 は、サービス・プログラム *MATHLIB/MATHFUNC の SIN へのシンボリック・リンクの例を示しています。SIN へのシンボリック・リンクが物理アドレスに変換されるのは、サービス・プログラム SPGMEXAMP がバインドされているプログラム・オブジェクトが活動化される時点です。

実行時に、使用されるプロシージャーおよびデータ項目に対して確立される物理リンクによって、以下のアクセス間のパフォーマンスの差はほとんどなくなります。

- ローカルのプロシージャーまたはデータ項目へのアクセス
- 同じプログラムにバインドされた別のモジュールまたはサービス・プログラム中のプロシージャーまたはデータ項目へのアクセス

24 ページの図 11 および 24 ページの図 12 は、ILE プログラム PGMEXAMP およびサービス・プログラム SPGMEXAMP の作成方法に関する概念を示しています。バインド・プログラムは、モジュール M1、M2、M3、M4 およびサービス・プログ

ラム PRINTS と MATHFUNC を使用して、ILE プログラム PGMEXAMP とサービス・プログラム SPGMEXAMP を作成しています。

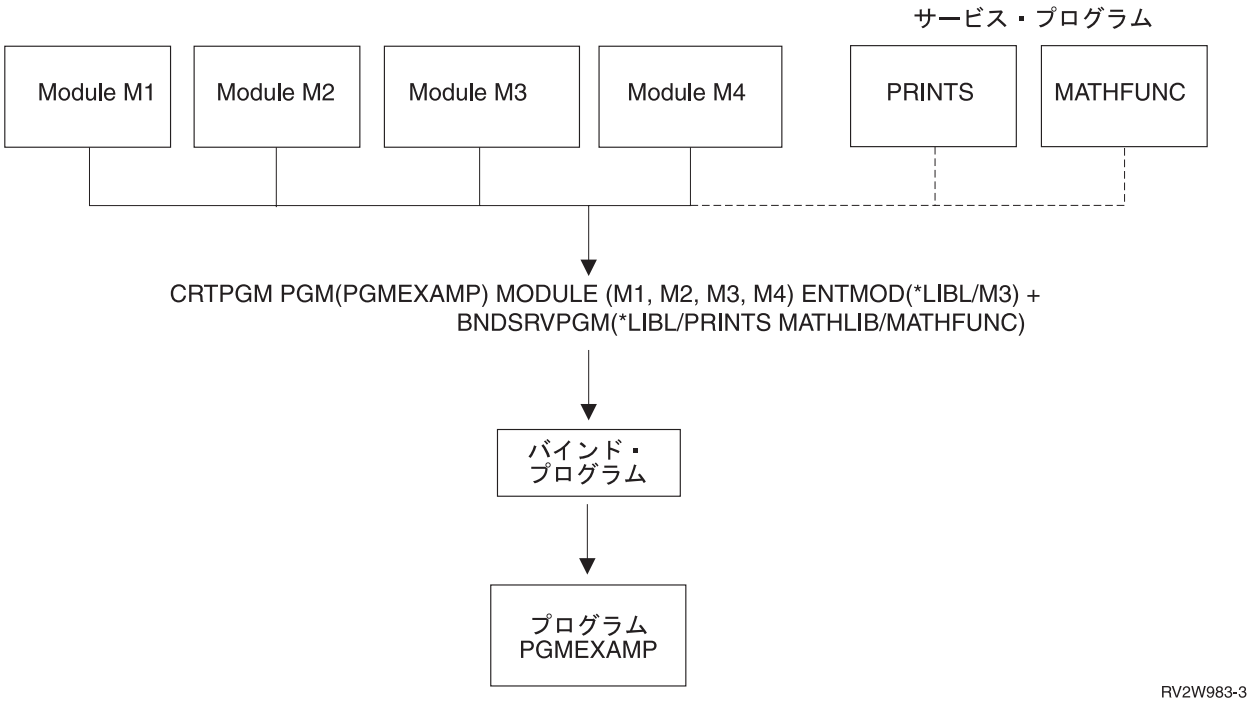

図 11. ILE プログラムの作成： (破線は、サービス・プログラムがコピーによってではなく、参照によってバインドされることを示しています。)

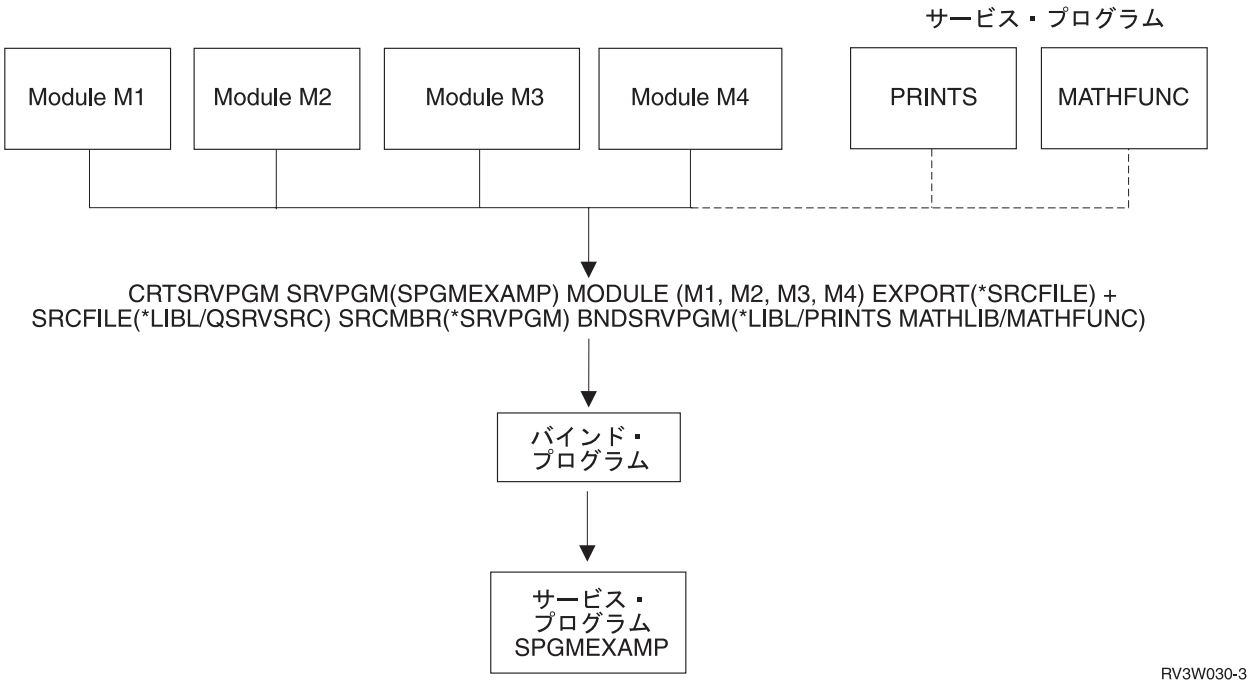

図 12. サービス・プログラムの作成： (破線は、サービス・プログラムがコピーによってではなく、参照によってバインドされることを示しています。)

ILE プログラムまたはサービス・プログラムの作成の詳細については 75 ページの『第 5 章 プログラム作成の概念』を参照してください。

プログラムおよびプロシージャーの呼び出し

ILE では、プログラムまたはプロシージャーのいずれでも呼び出すことができます。 ILE では、呼び出し元は、呼び出しステートメントのターゲットがプログラムであるか、プロシージャーであるかを識別しなければなりません。ILE 言語では、この要求を、プログラムとプロシージャーに対して別個の呼び出しステートメントを使用することによって通知します。したがって、ILE プログラムの作成時に、プログラムを呼び出すのか、プロシージャーを呼び出すのかを認識しなければなりません。

各 ILE 言語には、動的プログラム呼び出しと静的プロシージャー呼び出しを区別するするための固有の構文があります。各 ILE 言語の標準の呼び出しステートメントは、デフォルトによって、動的プログラム呼び出しまたは静的プロシージャー呼び出しのいずれかになります。RPG および COBOL の場合、デフォルトは動的プログラム呼び出しです。したがって、標準言語呼び出しは、OPM でも ILE でも同じタイプの機能を実行します。この規則により、OPM 言語から ILE 言語への移行は比較的容易になります。

バインド・プログラムは、長さが 256 文字までのプロシージャー名を処理することができます。プロシージャー名で使用できる長さについては、ILE HLL の「プログラマーの手引き」を参照してください。

動的プログラム呼び出し

動的プログラム呼び出しは、ILE プログラム・オブジェクトまたは OPM プログラム・オブジェクトのいずれかに制御を渡します。動的プログラム呼び出しには以下の事項が含まれます。

- OPM プログラムは、他の OPM プログラムまたは ILE プログラムを呼び出すことができますが、サービス・プログラムを呼び出すことはできません。
- ILE プログラムは OPM プログラムまたは他の ILE プログラムを呼び出すことはできますが、サービス・プログラムを呼び出すことはできません。
- サービス・プログラムは OPM プログラムまたは ILE プログラムを呼び出すことはできますが、他のサービス・プログラムを呼び出すことはできません。

静的プロシージャー呼び出し

静的プロシージャー呼び出しは ILE プロシージャーに制御を渡します。静的プロシージャー呼び出しは ILE 言語でのみコーディングすることができます。静的プロシージャー呼び出しは、以下のいずれかのプロシージャーの呼び出しに使用することができます。

- 同じモジュール内のプロシージャー
- 同じ ILE プログラムまたはサービス・プログラム内の別のモジュールのプロシージャー
- 別の ILE サービス・プログラム内のプロシージャー

図 13 は、静的プロシージャー呼び出しの例を示しています。この図は、以下のようなプロシージャー呼び出しを示しています。

- ILE プログラムのプロシージャーは、同じプログラムまたはサービス・プログラムのエクスポートされたプロシージャーを呼び出すことができます。プログラム A のプロシージャー P1 は、コピーされた別のモジュールのプロシージャー P2 を呼び出しています。プログラム C のプロシージャー P3 は、サービス・プログラム D のプロシージャー P4 を呼び出しています。
- サービス・プログラムのプロシージャーは、同じサービス・プログラムまたは別のサービス・プログラムのエクスポートされたプロシージャーを呼び出すことができます。サービス・プログラム B のプロシージャー P6 は、コピーされた別のモジュールのプロシージャー P7 を呼び出しています。サービス・プログラム E のプロシージャー P5 は、サービス・プログラム F のプロシージャー P4 を呼び出しています。

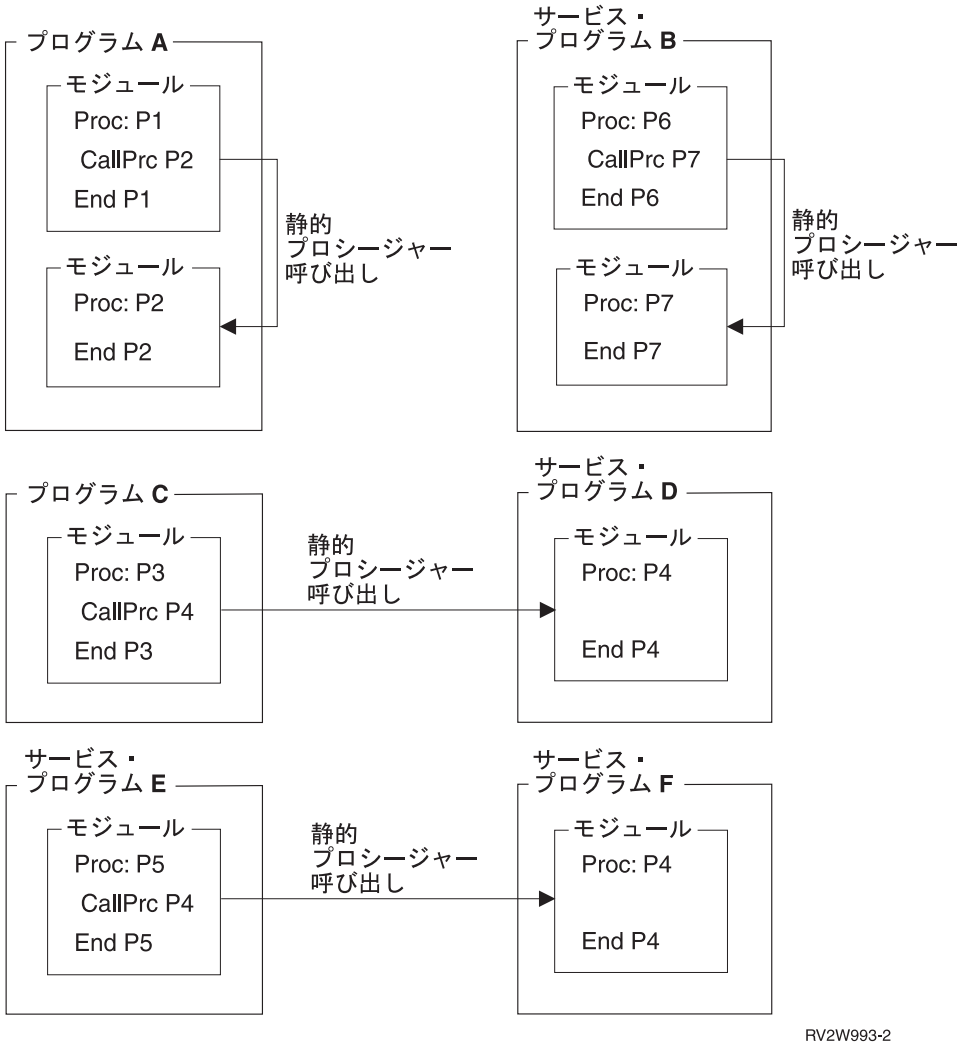

図 13. 静的プロシージャー呼び出し

活動化

ILE プログラムを正常に作成した後、そのコードを実行することになります。プログラムまたはサービス・プログラムを実行可能にするプロセスを**活動化**と呼びます。プログラムを活動化するためにコマンドを出す必要はありません。プログラムが呼び出される時点で、システムにより活動化が行われます。サービス・プログラムは呼び出されないので、このようなサービスを直接または間接に必要とするプログラムの呼び出しの過程で活動化されます。

活動化により以下の機能が行われます。

- プログラムまたはサービス・プログラムに必要な静的データを一意的に割り振ります。
- エクスポートするサービス・プログラムへのシンボリック・リンクを、物理アドレスへのリンクに変更します。

1 つのプログラムまたはサービス・プログラムを実行するジョブの数には関係なく、ストレージに存在するそのオブジェクトの一連の命令のコピーは 1 つだけです。 ただし、各プログラムの活動化はそれぞれの静的ストレージを持っています。したがって 1 つのプログラム・オブジェクトが、多くのジョブによって並行して使用された場合でも、静的変数は活動化ごとに分離されています。同じジョブでも、1 つのプログラムを複数の活動化グループで活動化することができますが、1 つの活動化は特定の活動化グループに限定されます。

次のいずれかに該当する場合、

- 活動化が必要なサービス・プログラムを見つけ出せない場合。
- サービス・プログラムが、シグニチャーによって示されているプロシージャーまたはデータ項目をもはやサポートしていない場合。

エラーが発生し、アプリケーションを実行することはできません。

プログラム活動化の詳細については 32 ページの『プログラム活動化の作成』を参照してください。

活動化により、プログラムで使用される静的変数に必要なストレージが割り振られる場合、スペースは活動化グループから割り振られます。プログラムまたはサービス・プログラムの作成時に、実行時に使用する活動化グループを指定することができます。

活動化グループの詳細については 33 ページの『活動化グループ』を参照してください。

エラー処理

28 ページの図 14 は、OPM プログラムおよび ILE プログラムのエラー処理構造全体を示しています。この図は、本書で以後、拡張エラー処理機能を記述する場合に使用します。ここでは、標準言語エラー処理機能について概説します。エラー処理の詳細については 45 ページの『エラー処理』を参照してください。

図は、例外メッセージ体系と呼ばれる基本的な層を示しています。例外メッセージは、OPM プログラムまたは ILE プログラムがエラーを検出するたびに、システム

によって生成されます。例外メッセージは、プログラム・エラーとは考えられない状況の情報を伝えるためにも使用されます。たとえば、データベース・レコードが見つからないという状況は、状況例外メッセージによって伝えられます。

それぞれの高水準言語では、言語に固有のエラー処理機能を定義しています。この機能は言語によって異なってはいますが、一般に、各 HLL ユーザーは特定のエラー状態を処理する意図を宣言することができます。この意図の宣言には、エラー処理ルーチンの識別が含まれます。例外が発生すると、システムは該当するエラー処理ルーチンを見つけて、ユーザーが作成した一連の命令に制御を渡します。ユーザーは、プログラムの終了またはエラーからの回復と続行を含む種々のアクションを行うことができます。

図 14 は、OPM プログラムが使用する例外メッセージ体系と同じ体系を、ILE が使用することを示しています。システムが生成する例外メッセージは、OPM プログラムと同様に ILE プログラムでも、言語固有のエラー処理を開始します。図の最下部の層には、例外メッセージを送受するための機能が含まれています。この機能は、メッセージ・ハンドラー API またはコマンドによって実行することができます。例外メッセージは ILE プログラムと OPM プログラムとの間で送受することができます。

図 14. OPM および ILE のエラー処理

言語固有のエラー処理は、OPM プログラム、および ILE プログラムの双方に同様に機能しますが、基本的な相違点があります。

- システムが例外メッセージを ILE プログラムに送る場合、プロシージャー名およびモジュール名を使用して例外メッセージを修飾します。ユーザーが例外メッセージを送る場合、これらの同じ修飾を指定することができます。例外メッセージが ILE プログラムのジョブ・ログに現れる場合、システムによって、通常、プログラム名、モジュール名、およびプロシージャー名が示されます。
- ILE プログラムに対する拡張最適化によって、生成される 1 つの命令群に複数の HLL ステートメント番号が関連づけられることがあります。したがって、ジョブ・ログに現れる例外メッセージには、最適化の結果として複数の HLL ステートメント番号が含まれることがあります。

他のエラー処理機能については 45 ページの『エラー処理』を参照してください。

最適化変換プログラム

OS/400 では、**最適化**とは、オブジェクトの実行時のパフォーマンスを最高にすることを意味します。すべての ILE 言語は、ILE 最適化変換プログラムによって提供される最適化手法にアクセスします。一般に、最適化の要求が高度になると、オブジェクトの作成に必要な時間が長くなります。実行時に、高度に最適化されたプログラムまたはサービス・プログラムは、低レベルの最適化によって作成された対応するプログラムまたはサービス・プログラムよりも速くなるはずです。

最適化はモジュール、プログラム・オブジェクト、およびサービス・プログラムごとに指定できますが、最適化手法は各モジュールに適用されます。最適化のレベルを次に示します。

10 または *NONE

20 または *BASIC

30 または *FULL

40 (レベル 30 以上の最適化)

モジュールを実稼働環境で使用する場合には、パフォーマンス上の理由で、高レベルの最適化が必要になるはずです。モジュールに対して設定したい最適化レベルで、コードをテストしてください。コードがすべて期待したとおりに機能することを確認して、ユーザーに提供してください。

レベル 30 (*FULL) またはレベル 40 の最適化は、プログラム命令にかなりの影響を与えることがあるので、アドレッシングに関するさまざまな例外の検出およびデバッグ上の制約について注意する必要があります。デバッグの考慮事項に関しては 145 ページの『第 10 章 デバッグに関する考慮事項』を参照してください。アドレッシング・エラーの考慮事項に関しては 221 ページの『付録 B. 最適化プログラムにおける例外』を参照してください。

デバッガー

ILE は、ソース・レベルのデバッグが可能なデバッガーを提供します。デバッガーはリスト・ファイルを処理することができ、停止点の設定、変数の表示、およびあるステップからの実行またはステップのとび越しを行うことができます。これらの機能は、コマンド行からコマンドを入力せずに実行することができます。デバッガーを使用して作業しているときは、コマンド行を使用することもできます。

ソース・レベルのデバッガーは、システム提供の API を使用して、ユーザーがプログラムまたはサービス・プログラムをデバッグできるようにします。これらの API は誰でも使用でき、ユーザー固有のデバッガーを作成することもできます。

OPM プログラムに対するデバッガーも継続して iSeries サーバーに存在しますが、OPM プログラムのデバッグにのみ使用することができます。

最適化されたモジュールをデバッグする場合、混乱が生じることがあります。 ILE デバッガーを使用して、実行中のプログラムまたはプロシージャーによって使用されている変数を表示または変更すると、以下のような状態になります。デバッガー

は、該当の変数のストレージのデータの検索または更新を行います。レベル 20 (*BASIC)、30 (*FULL)、または 40 の最適化では、データ変数の現行値がハードウェア・レジスターに存在し、デバッガーがアクセスできない場合があります。 (データ変数がハードウェア・レジスターにあるかどうかは、いくつかの要因によって決まります。これらの要因には、変数の使用方法、そのサイズ、およびデータ変数を検査または変更するためにコード内で停止した位置が含まれます。) したがって、変数に関して表示される値は現在値でない可能性があります。このため、開発の過程では、最適化レベル 10 (*NONE) を使用する必要があります。その後、稼動環境で最高のパフォーマンスを得るために、最適化レベル 30 (*FULL) または 40 に変更してください。

ILE デバッガーについての詳細は 145 ページの『第 10 章 デバッグに関する考慮事項』を参照してください。

第 3 章 ILE の拡張概念

本章では、ILE モデルの拡張された概念について記述します。本章を読む前に 13 ページの『第 2 章 ILE の基本概念』で記述した概念について理解しておくことが必要です。

プログラムの活動化

活動化とは、プログラム実行の準備を行うプロセスです。 ILE プログラムおよび ILE サービス・プログラムではいずれも、その実行に先立って、システムによる活動化が必要になります。

プログラムの活動化には、以下の 2 つの主なステップがあります。

1. プログラムの静的ストレージの割り振りと初期設定を行う。
2. サービス・プログラムへのプログラムのバインディングを完了する。

このトピックでは、ステップ 1 について記述します。ステップ 2 については 40 ページの『サービス・プログラムの活動化』で記述します。

32 ページの図 15 は、永続ディスク装置に保管されている 2 つの ILE プログラム・オブジェクトを示しています。すべての OS/400 オブジェクトと同様に、これらのプログラム・オブジェクトも、異なる OS/400 ジョブを実行している複数の並行ユーザーによって共用することができます。プログラムのコードのコピーは 1 つだけ存在します。ただし、これらの ILE プログラムの 1 つが呼び出された場合、そのプログラムで宣言されているいくつかの変数は、プログラムの活動化ごとに割り振りと初期設定が必要になります。

図 15 に示すように、それぞれのプログラム活動化は、これらの変数のうち少なくとも 1 つの固有のコピーをサポートします。変数の複数のコピーが同じ名前で 1 つのプログラム活動化内に存在する可能性があります。これは、その有効範囲が個々のプロシージャーになる静的変数を HLL で宣言できる場合に起こります。

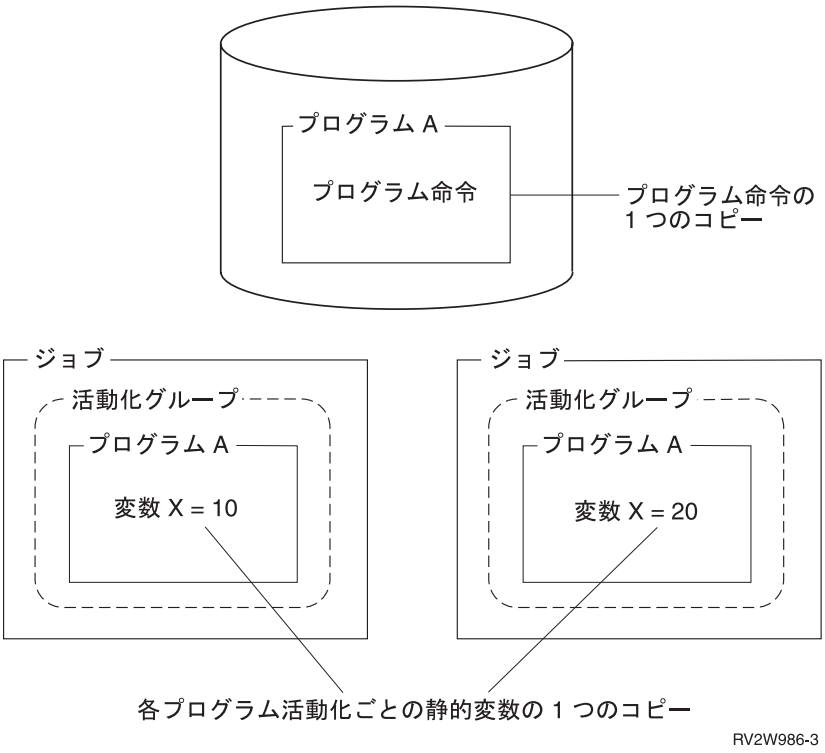

図 15. 各プログラム活動化ごとの静的変数の 1 つのコピー

プログラム活動化の作成

ILE は、活動化グループ内のプログラム活動化を追跡することによって、プログラム活動化のプロセスを管理します。活動化グループの定義については 33 ページの『活動化グループ』を参照してください。 1 つの活動化グループには、特定のプログラム・オブジェクトに関する活動化は 1 つしかありません。この規則を適用すると、異なる OS/400 ライブラリーにある同じ名前のプログラムは、異なるプログラム・オブジェクトと見なされます。

HLL プログラムで動的プログラム呼び出しステートメントを使用すると、ILE は、プログラムの作成時に指定された活動化グループを使用します。この属性は、プログラム作成 (CRTPGM) コマンドまたはサービス・プログラムの作成 (CRTSRVPGM) コマンドのいずれかで活動化グループ (ACTGRP) パラメーターを使用して指定します。このパラメーターで指定された活動化グループ内にプログラム活動化がすでに存在していれば、その活動化が使用されます。この活動化グループ内でプログラムが活動化されていない場合には、まずプログラムが活動化されてから実行されます。名前をもった活動化グループがあれば、その名前は、UPDPGM および UPDSRVPGM コマンドの ACTGRP パラメーターを使用して変更できます。

プログラムは、一度活動化されると、その活動化グループが削除されるまでは活動化されたままです。この規則の結果、活動化グループ内の呼び出しスタックにない活動プログラムが存在する可能性があります。 33 ページの図 16 は、1 つの活動化グループ内の 3 つの活動プログラムの例を示していますが、3 つのうち 2 つのプログラムのみが呼び出しスタックにプロシージャーを持っています。この例では、

プログラム A がプログラム B を呼び出すことによって、プログラム B が活動化されます。次にプログラム B は、プログラム A に戻ります。その次にプログラム A は、プログラム C を呼び出します。結果としての呼び出しスタックには、プログラム A およびプログラム C のプロシージャーが含まれますが、プログラム B のプロシージャーは含まれません。呼び出しスタックについては 119 ページの『呼び出しスタック』を参照してください。

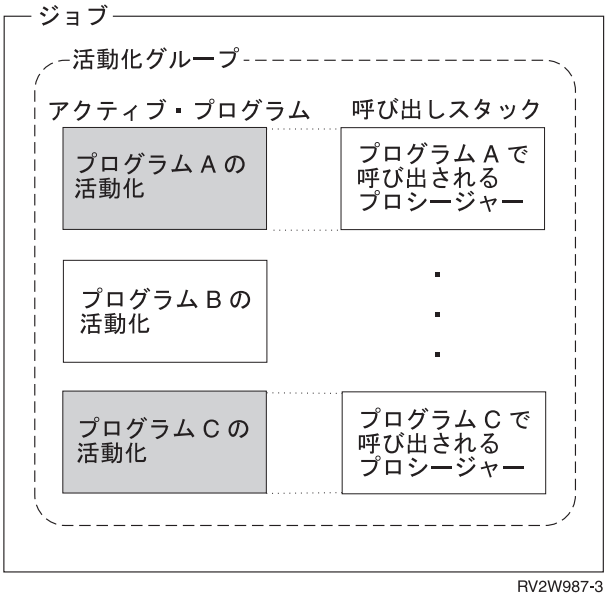

図 16. 呼び出しスタックにはないが活動化されている可能性があるプログラム

活動化グループ

すべての ILE プログラムおよびサービス・プログラムは、**活動化グループ**と呼ばれるジョブのサブストラクチャー内で活動化されます。このサブストラクチャーには、そのプログラムの実行に必要なリソースが含まれます。これらのリソースは以下のカテゴリーに分けることができます。

静的プログラム変数

動的ストレージ

一時データ管理機能リソース

特定のタイプの例外ハンドラーおよび終了プロシージャー

活動化グループは、静的プログラム変数のストレージを提供するために、単一レベル・ストアまたはテラスペースを使用します。詳細については 57 ページの『第 4 章 テラスペースおよび単一レベル・ストア』を参照してください。単一レベル・ストアを使用した場合、静的プログラム変数および動的ストレージが、各活動化グループごとの個別のアドレス・スペースに割り当てられます。これによって、プログラムのある程度の分離および不測のアクセスからの保護が可能になります。テラスペースを使用した場合、静的プログラム変数および動的ストレージは、テラスペー

ス内の個別のアドレス範囲に割り当てられます。これによって、プログラムの分離および不測のアクセスからの保護の程度は、単一レベル・ストアと比較すると低くなります。

一時データ管理機能リソースには、以下のものが含まれます。

オープン・ファイル (オープン・データ・パス (ODP))

コミットメント定義

ローカル SQL カーソル

リモート SQL カーソル

階層ファイル・システム (HFS)

ユーザー・インターフェース・マネージャー

QUERY 管理機能インスタンス

オープン通信リンク

共通プログラミング・インターフェース (CPI) コミュニケーション

これらのリソースが活動化グループ間で分離しているということが、1 つの基本概念を支えています。つまり、1 つの活動化グループ内で活動化されるすべてのプログラムは、1 つの連携アプリケーションとして開発されるという概念です。

ソフトウェア・ベンダーは、同じジョブ内で実行する他社のアプリケーションと自社のプログラムを分離するために、別個の活動化グループを選択することができます。このようなベンダー別の分離を 35 ページの図 17 に示しています。 この図では、 4 つの異なるベンダーからのソフトウェア・パッケージを統合することによって、お客様の全体のソリューションが得られることになります。活動化グループによって、各ベンダーのパッケージに関連するリソースを分離できるので、統合がより容易になります。

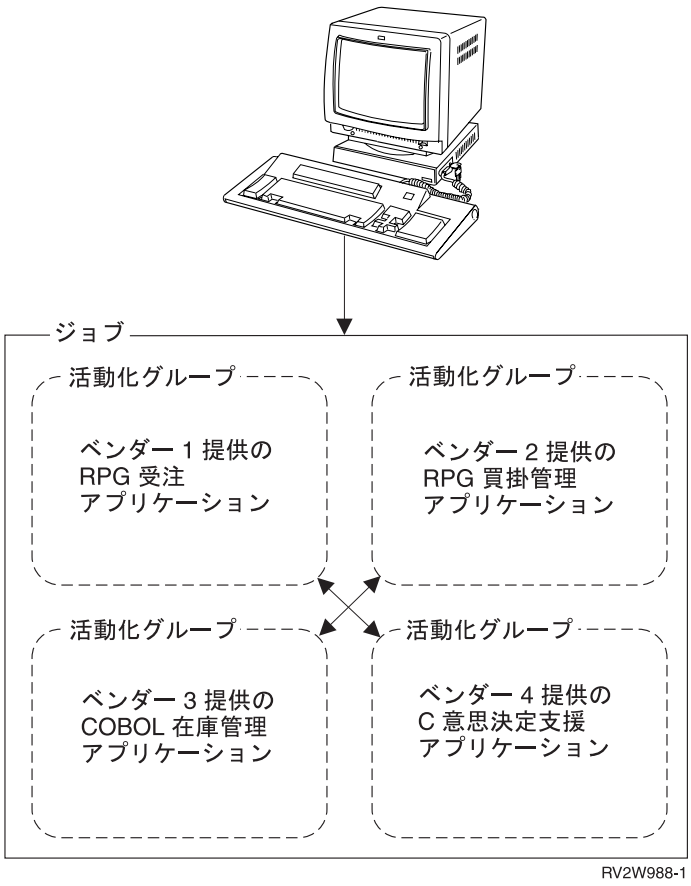

図 17. 活動化グループによる各ベンダーのアプリケーションの分離

活動化グループに対して上記のリソースを割り当てると、重要な結果をもたらします。つまり、活動化グループを削除すると、上記のすべてのリソースはシステムに戻される結果になるということです。活動化グループの削除時点でオープンのままの一時データ管理機能リソースは、システムによってクローズされます。 割り振り解除されていない静的プログラム変数と自動プログラム変数のストレージ、および動的ストレージは、システムに戻されます。

活動化グループの作成

ILE 活動化グループの実行時の作成は、プログラムまたはサービス・プログラムの作成時点で、活動化グループ属性を指定することによって制御することができます。この属性は、CRTPGM または CRTSRVPGM コマンドの ACTGRP パラメーターを使用して指定します。活動化グループを作成するコマンドはありません。

すべての ILE プログラムは、以下のいずれかの活動化グループ属性をもっています。

- ユーザー指定活動化グループ

  ACTGRP (名前) パラメーターによって指定された属性。この属性によって、ILE プログラムおよび ILE サービス・プログラムの集合を 1 つのアプリケーションとして管理することができます。この活動化グループは、最初に必要になった時点で作成されます。その後は、同じ活動化グループ名を指定しているすべてのプログラムおよびサービス・プログラムによって使用されます。

- システム指定の活動化グループ

  CRTPGM コマンドの ACTGRP(*NEW) パラメーターを使用して指定します。この属性によって、プログラムの呼び出し時に新しい活動化グループを作成することができます。ILE はこの活動化グループに対する名前を選択します。ILE によって割り当てられる名前は、ジョブ内で固有です。システム指定活動化グループに割り当てられる名前は、ユーザー指定活動化グループとしてユーザーが選択する名前のいずれにも一致しません。ILE サービス・プログラムは、この属性をサポートしていません。

- 呼び出し側プログラムの活動化グループを使用するための属性

  ACTGRP(*CALLER) パラメーターを使用して指定します。この属性によって、呼び出し側プログラムの活動化グループ内で活動化される ILE プログラムまたは ILE サービス・プログラムを作成することができます。この属性を指定すると、プログラムまたはサービス・プログラムが活動化される時点で、新しい活動化グループは作成されません。

- プログラム言語およびストレージ・モデルに適した活動化グループを選択するための属性

  CRTPGM コマンドの ACTGRP(*ENTMOD) パラメーターを使用して指定します。 ACTGRP(*ENTMOD) を指定すると、ENTMOD パラメーターで指定されたプログラム入り口プロシージャー・モジュールが検査されます。この場合、次のいずれかのことが行われます。

  – モジュール属性が RPGLE または CBLLE である場合には、QILE が活動化グループとして使用されます。

  – モジュール属性が CLLE である場合、

    - STGMDL(*SNGLVL) が指定されているときには、QILE が活動化グループとして使用されます。

    - STGMDL(*TERASPACE) が指定されているときには、QILETS が活動化グループとして使用されます。

  – モジュール属性が RPGLE、CBLLE、または CLLE ではない場合には、*NEW が活動化グループとして使用されます。

  ACTGRP(*ENTMOD) は、CRTPGM コマンドのこのパラメーターのデフォルト値です。

ジョブ内のすべての活動化グループには名前があります。ジョブに活動化グループが存在する場合、その名前を指定しているプログラムおよびサービス・プログラムを活動化するために、その名前を ILE が使用します。このような設計の結果として、重複する活動化グループ名が、1 つのジョブ内に存在することはありません。ただし、UPDPGM および UPDSRVPGM 上で ACTGRP パラメーターを使用することによって、活動化グループの名前を変更することができます。

デフォルトの活動化グループ

OS/400 ジョブが開始される時点で、システムは他のすべての OPM プログラムによって使用される 2 つの活動化グループを作成します。デフォルトの活動化グループは、静的プログラム変数に単一レベル・ストアを使用します。この OPM デフォルトの活動化グループを削除することはできません。OPM デフォルトの活動化グループは、ジョブの終了時にシステムによって削除されます。

以下の条件に該当する場合、ILE プログラムおよび ILE サービス・プログラムは、この OPM デフォルトの活動化グループで活動化することができます。

- ILE プログラムまたは ILE サービス・プログラムが、活動化グループ *CALLER オプションを指定して作成された場合。
- その ILE プログラムまたは ILE サービス・プログラムの呼び出しが、 OPM デフォルトの活動化グループで行われている場合。
- ILE プログラムまたはサービス・プログラムが、テラスペース・ストレージ・モデルを使用していない場合。

デフォルトの活動化グループは削除できないため、ILE HLL の終了 verb によって、完全な終了処理を行うことはできません。オープン・ファイルは、ジョブが終了するまでシステムでクローズすることはできません。ILE プログラムによって使用された静的およびヒープ・ストレージは、ジョブが終了するまでシステムに戻すことはできません。

図 18 は、ILE 活動化グループと OPM デフォルトの活動化グループが存在する典型的な OS/400 ジョブを示しています。2 つの OPM デフォルトの活動化グループは、両方のグループを表すために特殊値 *DFTACTGRP が使用されるので、結合されています。各活動化グループ内のボックスは、プログラムの活動化を示しています。

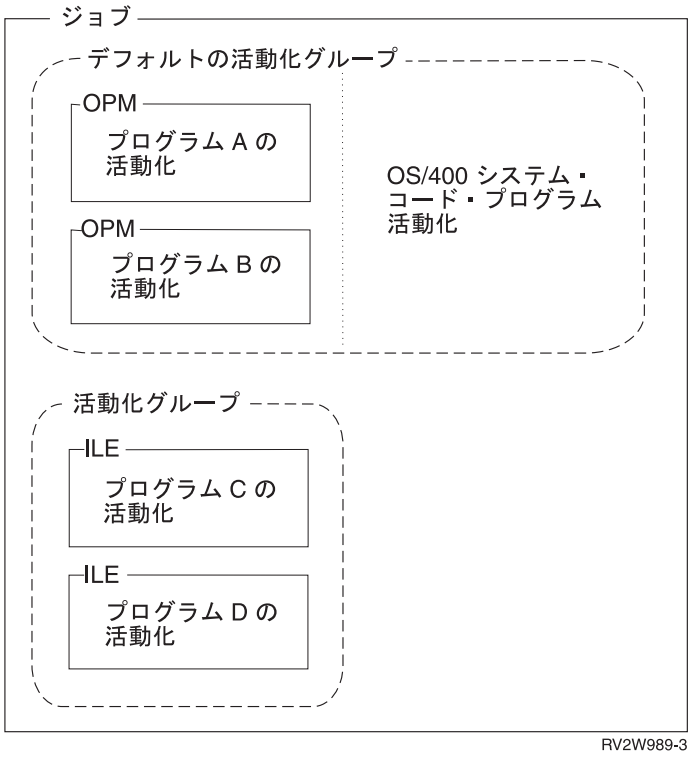

図 18. デフォルトの活動化グループおよび ILE 活動化グループ

ILE 活動化グループの削除

活動化グループは、ジョブ内でのリソースの作成を要求します。アプリケーションで活動化グループを再使用できる場合は、処理時間を節約することができます。ILE は、活動化グループを終了または削除せずに活動化グループから戻れるようなオプションをいくつか用意しています。活動化グループが削除されるかどうかは、活動化グループのタイプとアプリケーションの終了方法によって決まります。

アプリケーションは以下の方法により、1 つの活動化グループをそのまま存続させ、別の活動化グループで実行中の呼び出しスタック項目 (119 ページの『呼び出しスタック』を参照) に戻ることができます。

- HLL の終了 verb

  たとえば、COBOL の STOP RUN または C の exit()。
- 未処理の例外

  未処理の例外は、システムによって、別の活動化グループ内の呼び出しスタック項目に移されることがあります。
- 言語固有の HLL 戻りステートメント

  たとえば、C の return ステートメント、 COBOL の EXIT PROGRAM ステートメント、または RPG の RETURN ステートメント。
- スキップ操作

  たとえば、例外メッセージの送信、または現在の活動化グループにない呼び出しスタック項目へのブランチ。

HLL 終了 verb を使用することによって、アプリケーションから活動化グループを削除することができます。未処理の例外によっても、活動化グループが削除されます。 最も近い制御境界が活動化グループ内の最も古い呼び出しスタック項目である場合、上記の操作によって、活動化グループ (ハード制御境界とも呼ばれる) は常に削除されます。最も近い制御境界が最も古い呼び出しスタック項目 (ソフト制御境界とも呼ばれる) でない場合、制御境界の前の呼び出しスタック項目に制御が渡されます。ただし、活動化グループは削除されません。

制御境界は、アプリケーションの境界を示す呼び出しスタック項目です。活動化グループ間で呼び出しを行うたびに、ILE は制御境界を定義します。制御境界の定義については 42 ページの『制御境界』を参照してください。

ユーザー指定活動化グループは、後で使用するためにジョブ内に残すことができます。このタイプの活動化グループの場合、通常の戻りまたはハード制御境界を超えるスキップ操作では、活動化グループは削除されません。同じ操作をシステム指定活動化グループ内で使用すると、この活動化グループは削除されます。システム指定活動化グループは、システムが生成した名前を指定して再使用することができないので、常に削除されます。活動化グループの最も古い呼び出しスタック項目からの正常な戻りに関する言語依存の規則については、 ILE HLL の「プログラマーの手引き」を参照してください。

39 ページの図 19 は活動化グループを残す方法の例を示しています。この図で、プロシージャー P1 は最も古い呼び出しスタック項目です。システム指定活動化グループ (ACTGRP(*NEW) オプションを指定して作成) の場合、P1 からの通常の戻りによってこの活動化グループは削除されます。ユーザー指定活動化グループ

(ACTGRP(名前) オプションを指定して作成) の場合、P1 からの通常の戻りによってこの活動化グループが削除されることはありません。

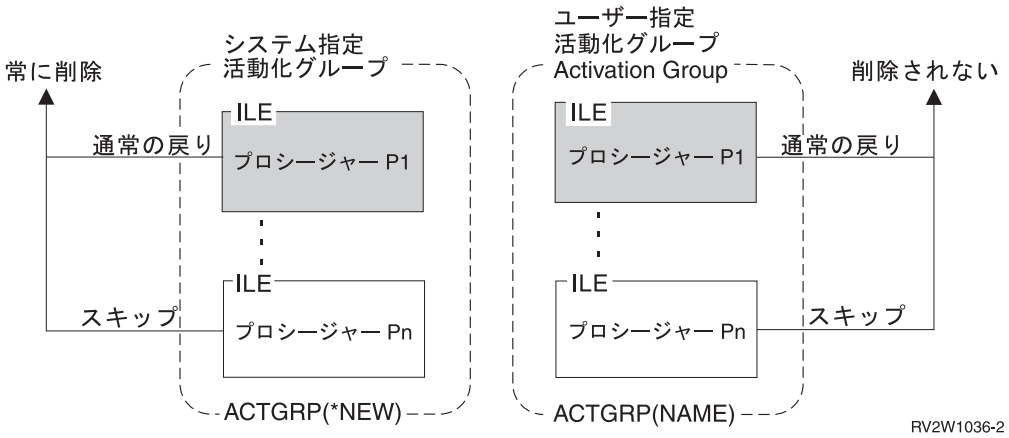

図 19. ユーザー指定活動化グループとシステム指定活動化グループの存続

ユーザー指定活動化グループがジョブ内に残っている場合は、活動化グループの再利用 (RCLACTGRP) コマンドを用いて削除することができます。このコマンドで、アプリケーションから戻った後に、指定の活動化グループを削除することができます。このコマンドで削除できるのは、使用中でない活動化グループだけです。

40 ページの図 20 は、使用されていない 1 つの活動化グループと、現在使用中の 1 つの活動化グループが存在する OS/400 ジョブを示しています。活動化グループは、その活動化グループ内で活動化されている ILE プロシージャーに関する呼び出しスタック項目が存在する場合に、使用中であると見なされます。プログラム A またはプログラム B で RCLACTGRP コマンドを使用すると、プログラム C とプログラム D の活動化グループが削除されます。

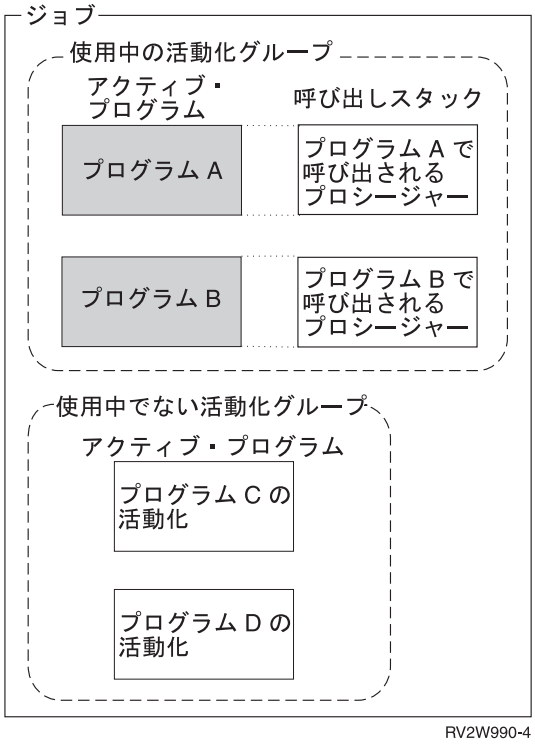

図 20. 呼び出しスタックに項目が存在する使用中の活動化グループ

活動化グループが ILE によって削除されると、ある終了操作処理が行われます。この処理には、ユーザー登録出口プロシージャー、データ管理機能クリーンアップ、および言語クリーンアップ (ファイルのクローズなど) の呼び出しが含まれます。活動化グループが削除される時点で行われるデータ管理機能処理の詳細については 53 ページの『データ管理機能の有効範囲指定の規則』を参照してください。

サービス・プログラムの活動化

このトピックでは、サービス・プログラムの活動化にシステムが使用する特有のステップについて記述します。プログラムとサービス・プログラムに使用される共通のステップについては 31 ページの『プログラムの活動化』を参照してください。以下の活動化処理は、サービス・プログラムに特有です。

- サービス・プログラムの活動化は、ILE プログラムに対する動的プログラム呼び出しの一環として間接的に開始されます。
- サービス・プログラムの活動化には、物理リンクへのシンボリック・リンクのマッピングによるプログラム間のバインディング接続の完了が含まれます。
- サービス・プログラムの活動化には、シグニチャー・チェック処理が含まれます。

活動化グループ内で最初に活動化される ILE プログラムは、ILE サービス・プログラムへのバインディングが行われているか否かが検査されます。活動化されるプログラムにサービス・プログラムがすでにバインドされている場合、サービス・プログラムも同じ動的呼び出し処理の一環として活動化されます。このプロセスは、必要なすべてのサービス・プログラムが活動化されるまで繰り返されます。

図 21 は、ILE プログラム A が B、C、および D にバインドされることを示しています。ILE サービス・プログラム B および C は、 ILE サービス・プログラム E にもバインドされます。各プログラムおよびサービス・プログラムの活動化グループ属性が示されています。

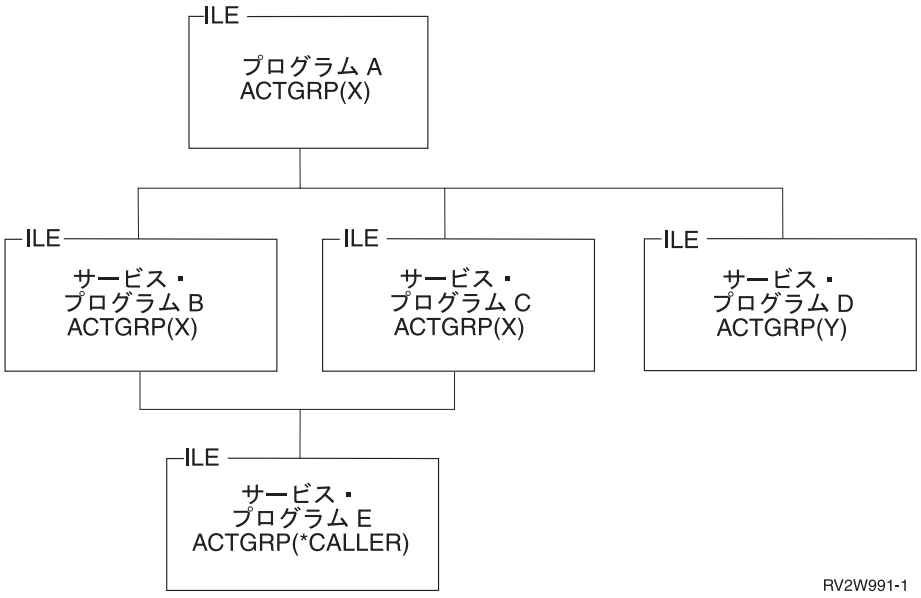

図 21. サービス・プログラムの活動化

ILE プログラム A が活動化されると、以下の処理が行われます。

- そのサービス・プログラムは、明示的なライブラリー名または現行ライブラリー・リストを使用して検索されます。このオプションは、プログラムおよびサービス・プログラムの作成時にユーザーが制御します。
- プログラムと同様に、サービス・プログラムの活動化は、1 つの活動化グループ内で 1 回だけ行われます。図 21 で、サービス・プログラム E は、サービス・プログラム B と C によって使用されていますが、活動化は 1 回だけです。
- 2 番目の活動化グループ (Y) が、サービス・プログラム D に対して作成されます。
- すべてのプログラムおよびサービス・プログラムの間でシグニチャー・チェックが行われます。

概念上、このプロセスは、プログラムおよびサービス・プログラムの作成時点で開始されたバインディング・プロセスの完了処理と見なすことができます。
CRTPGM、および CRTSRVPGM コマンドは、参照された各サービス・プログラムの名前とライブラリーを保管しています。エクスポートされるプロシージャーとデータ項目に関するテーブルへの索引も、プログラム作成時にクライアント・プログラムまたはサービス・プログラムで保管されます。サービス・プログラム活動化のプロセスは、これらの記号による参照を、実行時に使用可能なアドレスに変更することによって、バインディング・ステップを完了します。

サービス・プログラムが活動化された後は、別のサービス・プログラムのモジュールに対して、静的プロシージャーによる呼び出しおよび静的データ項目参照が処理されます。処理の量は、コピーによってモジュールを同じプログラムにバインドし

た場合と同じです。ただし、コピーによってバインドされるモジュールに必要な活動化時間の処理の量は、サービス・プログラムより少なくなります。

プログラムおよびサービス・プログラムの活動化を行うには、ILE プログラム・オブジェクトおよびすべての ILE サービス・プログラム・オブジェクトに対する実行権限が必要です。 41 ページの図 21 で、プログラム A およびすべてのサービス・プログラムに対する権限の検査には、プログラム A の呼び出し元の現行の権限が使用されます。また、プログラム A の権限は、すべてのサービス・プログラムに対する権限の検査にも使われます。サービス・プログラム B、C、または D の権限は、サービス・プログラム E に対する権限の検査には使用されない点に注意してください。

制御境界

ILE は、未処理の機能チェックが発生するか、または HLL 終了 verb が使用されると、以下のアクションを行います。ILE は、アプリケーションの境界を示している呼び出しスタック項目の呼び出し元に制御を渡します。このような呼び出しスタック項目は、**制御境界**と呼ばれます。

制御境界には、2 つの定義があります。『ILE 活動化グループの制御境界』および 43 ページの『OPM デフォルトの活動化グループの制御境界』では、次のような定義を図示しています。

制御境界は以下のいずれかです。

- 直前の呼び出しスタック項目が別のデフォルト以外の活動化グループ内にある ILE 呼び出しスタック項目。
- 直前の呼び出しスタック項目が OPM プログラムである ILE 呼び出しスタック項目。

ILE 活動化グループの制御境界

この例は、ILE 活動化グループ間での制御境界の定義方法を示しています。

43 ページの図 22 は、2 つの ILE 活動化グループ、および様々な呼び出しによって設定される制御境界を示しています。プロシージャー P2、P3、および P6 は、制御境界になる可能性があります。たとえば、プロシージャー P7 を実行中の場合、プロシージャー P6 が制御境界になります。プロシージャー P4 または P5 を実行中の場合、プロシージャー P3 が制御境界になります。

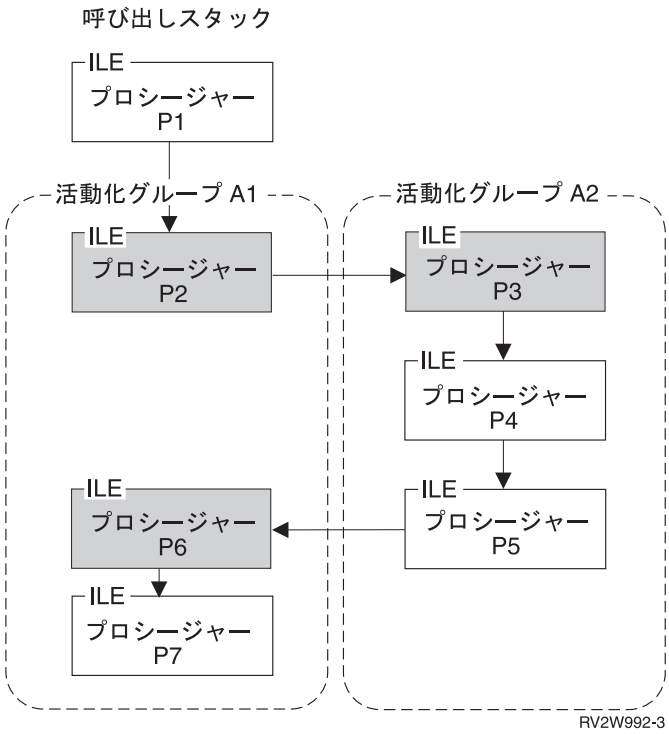

図 22. 制御境界： (陰影の付いたプロシージャーが制御境界です。)

OPM デフォルトの活動化グループの制御境界

この例は、OPM デフォルトの活動化グループ内での ILE プログラムの実行時に制御境界が定義される方法を示しています。

44 ページの図 23は、OPM デフォルトの活動化グループ内で実行される 3 つの ILE プロシージャー (P1、P2、および P3) を示しています。この例は、ACTGRP(*CALLER) パラメーター値を指定した CRTPGM コマンドまたは CRTSRVPGM コマンドを使用して作成されています。プロシージャー P1 と P3 は、その前の呼び出しスタック項目がそれぞれ OPM プログラム A と B なので、制御境界になる可能性があります。

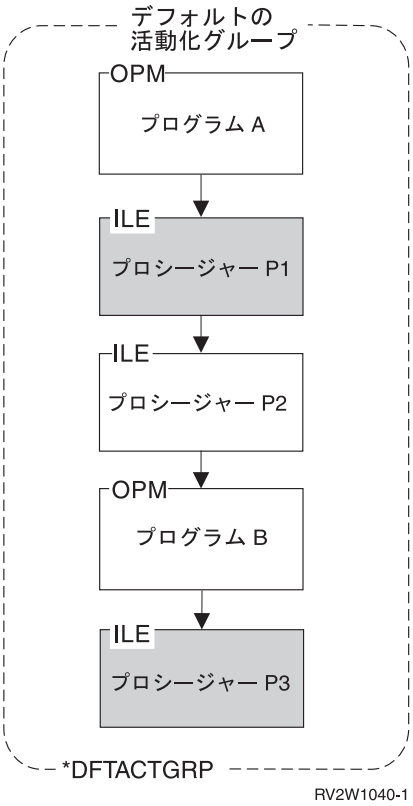

図 23. デフォルトの活動化グループ内の制御境界： (陰影の付いたプロシージャーが制御境界です。)

制御境界の使用

ILE HLL 終了 verb を使用すると、ILE は呼び出しスタック上の最新の制御境界を使用して、制御を渡す先を決めます。ILE がすべての終了処理を完了すると、制御境界の直前の呼び出しスタック項目に制御が渡ります。

ILE プロシージャー内で未処理の機能チェックが発生した場合も、制御境界が使用されます。制御境界は、未処理の機能チェックを総称 ILE 障害条件に**プロモート**する呼び出しスタック上の場所を定義します。詳細は 45 ページの『エラー処理』を参照してください。

最も近い制御境界が ILE 活動化グループ内の最も古い呼び出しスタック項目の場合、HLL 終了 verb または未処理の機能チェックによって、活動化グループが削除されます。最も近い制御境界が ILE 活動化グループ内の最も古い呼び出しスタック項目ではない場合、制御境界の直前の呼び出しスタック項目に制御が戻されます。同じ活動化グループ内に、より古い呼び出しスタック項目があるので、活動化グループは削除されません。

43 ページの図 22 は、各活動化グループ内の最も古い呼び出しスタック項目として、プロシージャー P2 と P3 を示しています。プロシージャー P2、P3、P4、または P5 (P6 または P7 ではない) で HLL 終了 verb を使用すると、活動化グループ A2 が削除されます。

エラー処理

このトピックでは、OPM プログラムと ILE プログラムに関する拡張エラー処理機能について記述します。これらの機能が例外メッセージ体系で占める位置付けについては 図 24 を参照してください。個々の参照情報およびその他の概念については 135 ページの『第 9 章 例外および条件管理』を参照してください。図 24 は、エラー処理の概要を示しています。このトピックでは、この図の最下部の層から最上部への層の順に記述します。最上部の層は、OPM プログラムまたは ILE プログラムでエラーの処理に使用できる機能を示しています。

図 24. ILE と OPM のエラー処理

ジョブ・メッセージ待ち行列

メッセージ待ち行列は、各 OS/400 ジョブ内のすべてのスタック呼び出し項目ごとに存在します。このメッセージ待ち行列によって、各呼び出しスタックで実行されるプログラムおよびプロシージャーの間の通知メッセージと例外メッセージの送受が容易になります。このメッセージ待ち行列は、**呼び出しメッセージ待ち行列**と呼ばれます。

呼び出しメッセージ待ち行列は、呼び出しスタック上の OPM プログラムまたは ILE プロシージャーの名前によって識別されます。プロシージャー名またはプログラム名は、送信するメッセージのターゲット呼び出しスタック項目を指定するのに使用されます。ILE プロシージャー名は固有ではないので、ILE モジュール名、ILE プログラム名、または ILE サービス・プログラム名を必要に応じて指定することができます。同じプログラムまたはプロシージャーに複数の呼び出しスタック項目がある場合、最も近い呼び出しメッセージ待ち行列が使用されます。

呼び出しメッセージ待ち行列に加えて、各 OS/400 ジョブ内には 1 つの**外部メッセージ待ち行列**が含まれます。ジョブ内で実行中のすべてのプログラムおよびプロシージャーは、このキューを使用することによって、対話式ジョブとワークステーション・ユーザーの間でメッセージをやりとりすることができます。

IBM では、メッセージ処理 API を使用して例外メッセージを送受信する方法に関するオンライン情報を用意しています。iSeries Information Center の**プログラミング**・カテゴリーの中の *API* セクションを参照してください。

例外メッセージとその送信方法

ここでは、種々の例外メッセージのタイプと、例外メッセージの送信方法について記述します。

ILE と OPM のエラー処理は、例外メッセージのタイプによって異なります。特に限定しない限り、**例外メッセージ**という用語は、以下のメッセージ・タイプのいずれかを示します。

エスケープ (*ESCAPE)
処理の完了前に、プログラムの異常終了を引き起こすエラーを示します。エスケープ例外メッセージの送信後、制御はユーザーに渡りません。

状況 (*STATUS)
プログラムにより行われている処理の状況を示します。このメッセージ・タイプの送信後、制御がユーザーに渡ることがあります。制御がユーザーに渡るかどうかは、受信プログラムがその状況メッセージを処理する方法によって決まります。

通知 (*NOTIFY)
訂正アクションを必要とする状態または呼び出し側プログラムからの応答が必要な状態を示します。このメッセージ・タイプの送信後、制御がユーザーに渡ることがあります。制御がユーザーに渡るかどうかは、受信プログラムがその通知メッセージを処理する方法によって決まります。

機能チェック
プログラムによって予期されていない終了状態を示します。ILE 機能チェック CEE9901 は、システムによってのみ送信される特殊なメッセージ・タイプです。OPM の機能チェックは、メッセージ ID が CPF9999 のエスケープ・メッセージ・タイプです。

IBM では、これらのメッセージ・タイプおよび他の OS/400 メッセージ・タイプに関するオンライン情報を用意しています。 iSeries Information Center の**プログラミング**・カテゴリーの中の *API* セクションを参照してください。

例外メッセージは以下の方法で送信されます。

- システムによる生成

  OS/400 (HLL を含む) は、プログラミング・エラーまたは状況情報を示すために例外メッセージを生成します。
- メッセージ・ハンドラー API

  例外メッセージを特定の呼び出しメッセージ待ち行列に送信するために、プログラム・メッセージ送信 (QMHSNDPM) API を使用することができます。
- ILE API

  条件シグナル (CEESGL) バインド可能 API を使用して、ILE 条件を発生させることができます。この条件によって、エスケープ例外メッセージまたは状況例外メッセージが送信されます。
- 言語に固有の verb

ILE C および ILE C++ の場合、raise() 関数が C シグナルを生成します。ILE RPG および ILE COBOL には同様の関数はありません。

例外メッセージの処理方法

ユーザーまたはシステムが例外メッセージを送信すると、例外処理が開始されます。この処理は、例外が処理されるまで、つまり例外メッセージが処理されたことを示すために例外メッセージが変更されるまで続行されます。

システムは、OPM 呼び出しメッセージ待ち行列に対する例外ハンドラーを呼び出す時点で、例外メッセージが処理されたことを示すために例外メッセージを変更します。ILE HLL は、ILE 呼び出しメッセージ待ち行列に対して例外ハンドラーが呼び出される前に例外メッセージを変更します。結果として、HLL 固有のエラー処理は、ハンドラーが呼び出される時点で、例外メッセージが処理済みであると見なします。HLL 固有のエラー処理を使用しない場合、ILE HLL は例外メッセージを処理することも、例外処理を続行させることもできます。未処理の例外メッセージに関する各 HLL のデフォルトのアクションについては、各 ILE HLL の資料を参照してください。

ILE では、言語固有のエラー処理を回避できるような機能がさらに定義されています。この機能には、直接モニター・ハンドラーと ILE 条件ハンドラーがあります。この機能を使用する場合、例外が処理されたことを示すように例外メッセージを変更するのは、ユーザーの責任になります。ユーザーが例外メッセージを変更しない場合は、システムは、別の例外ハンドラーの検索を試行することにより、例外処理を継続します。 49 ページの『例外ハンドラーのタイプ』では、直接モニター・ハンドラーおよび ILE 条件ハンドラーについて詳細に記述しています。 IBM では、例外メッセージを変更する方法について記述したオンライン情報を提供しています。 iSeries Information Center の**プログラミング**・カテゴリーの中の *API* セクションにある例外メッセージ変更 (QMHCHGEM) API を参照してください。

例外からの回復

例外が送られた後で、処理を続行したい場合があります。エラーからの回復は、エラーに対応できるアプリケーションを作成するための便利なツールです。ILE および OPM のプログラムの場合、システムは**再開点**の概念を定義しています。再開点は、最初は、例外を起こした命令の直後の命令に設定されます。例外を処理した後、再開点から処理を続行することができます。再開点の使用および変更の方法についての詳細は 135 ページの『第 9 章 例外および条件管理』を参照してください。

未処理例外に関するデフォルト・アクション

HLL で例外メッセージを処理しない場合、システムは、未処理の例外に関してデフォルト・アクションを行います。

45 ページの図 24 は、例外の送信先が OPM プログラムであるか、または ILE プログラムであるかに基づいて異なる未処理例外のデフォルト・アクションを示しています。OPM および ILE の異なるデフォルト・アクションにより、エラー処理機能に基本的な相違が生じます。

OPM の場合、未処理の例外によって、機能チェック・メッセージとして知られる特殊なエスケープ・メッセージが生成されます。このメッセージには特殊なメッセージ ID である CPF9999 が与えられます。このメッセージは、元の例外メッセージを引き起こした呼び出しスタック項目の呼び出しメッセージ待ち行列に送られます。機能チェック・メッセージが処理されない場合、システムは、その呼び出しスタック項目を除去します。次に、機能チェック・メッセージを前の呼び出しスタック項目に送ります。このプロセスは、機能チェック・メッセージが処理されるまで、続行されます。機能チェック・メッセージが最後まで処理されない場合、ジョブが終了します。

ILE の場合、未処理の例外メッセージは、前の呼び出しスタック項目のメッセージ待ち行列へパーコレートされます。 **パーコレーション**は、例外メッセージが前の呼び出しメッセージ待ち行列に移されると行われます。パーコレーションには、同じ例外メッセージを前の呼び出しメッセージ待ち行列に送信する効果があります。パーコレーションが行われると、例外処理は、前の呼び出しスタック項目で続行されます。

49 ページの図 25 は、ILE 内の未処理の例外メッセージを示しています。この例で、プロシージャー P1 は制御境界です。プロシージャー P1 は活動化グループ内の最も古い呼び出しスタック項目でもあります。プロシージャー P4 で未処理の例外メッセージが発生しました。未処理例外のパーコレーションは、制御境界に到達するまで、または例外メッセージが処理されるまで続行されます。未処理例外は、制御境界にパーコレートされるとき機能チェックに変換されます。例外がエスケープの場合、機能チェックが生成されます。通知例外の場合には、デフォルト応答が送られ、例外が処理され、通知の送信側は続行を許可されます。状況例外の場合には、例外が処理され、状況の送信側は続行を許可されます。再開点 (プロシージャー P3 に示されている) は、機能チェックの例外処理を続行すべき場所にある呼び出しスタック項目を定義するために使用されます。ILE の場合、次の処理ステップは、この呼び出しスタック項目への特殊な機能チェック例外メッセージの送信になります。この例では、この呼び出しスタック項目はプロシージャー P3 です。

機能チェック例外メッセージは、そのとき処理されるかまたは制御境界にパーコレートされます。処理されると、通常の処理が続行され、例外処理は終了します。 機能チェック・メッセージが制御境界にパーコレートされると、ILE はこのアプリケーションが予期しないエラーによって終了したと見なします。総称障害例外メッセージは、ILE によってすべての言語に対して定義されています。このメッセージは CEE9901 であり、ILE によって制御境界の呼び出し側に送信されます。

ILE に定義されている未処理例外メッセージのデフォルトのアクションによって、混合言語アプリケーションで発生するエラー条件から回復することができます。予期しないエラーの場合、ILE は、すべての言語に関して整合性のある障害メッセージを出します。これによって、種々のソースからのアプリケーションを統合する機能を改善することができます。

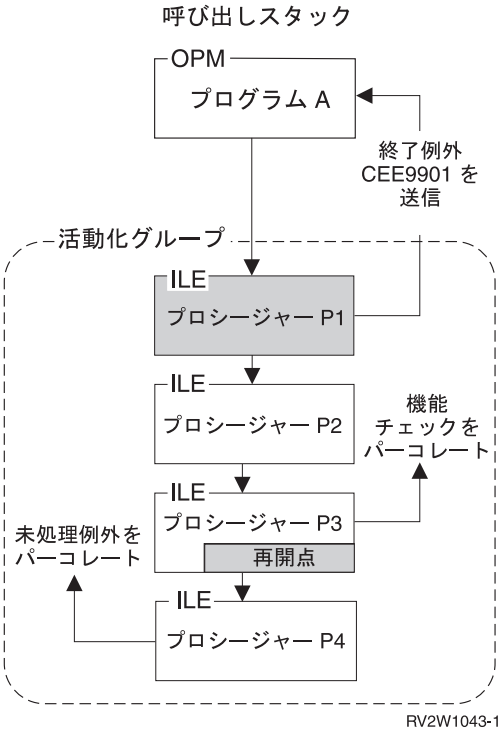

図 25. 未処理例外のデフォルト・アクション

例外ハンドラーのタイプ

この項では、OPM プログラムと ILE プログラムの両者に対し提供されている例外ハンドラーのタイプの概要を記述します。 45 ページの図 24 に示したように、これは、例外メッセージ体系の最上部の層です。ILE には、OPM より多くの例外処理能力があります。

OPM プログラムの場合、HLL 固有のエラー処理は、各呼び出しスタック項目に 1 つ以上の処理ルーチンを提供します。例外が OPM プログラムに送信されると、システムによって適切なルーチンが呼び出されます。

ILE の HLL 固有のエラー処理は同じ機能を提供します。しかし、ILE には他のタイプの例外ハンドラーもあります。これらのタイプのハンドラーによって、例外メッセージ体系の直接制御および HLL 固有エラー処理のバイパスが可能になります。 以下のタイプのハンドラーが ILE に追加されています。

直接モニター・ハンドラー
ILE 条件ハンドラー

これらのタイプのハンドラーが各 HLL によってサポートされているかどうかを調べるには、該当の ILE HLL の「プログラマーの手引き」を参照してください。

直接モニター・ハンドラーによって、HLL ソース・ステートメントの限定された範囲で例外モニターを直接宣言することができます。ILE C の場合、この機能は、#pragma ディレクティブにより使用可能になります。ILE COBOL は、ILE C と同様の意味での限定された HLL ソース・ステートメントの範囲での例外モニターを

直接宣言することはありません。ILE COBOL プログラムは、任意のソース・コードの範囲でのハンドラーの使用可能性と使用不能性を直接コーディングすることはできません。ただし、

```
ADD a TO b ON SIZE ERROR imperative
```

といったステートメントは、同様のメカニズムを使用するために内部的にマップされます。このようにして、どのハンドラーが最初に制御を取得するかという優先順位の意味で、このようなステートメントの範囲の条件付き命令が、ILE 条件ハンドラー (CEEHDLR 経由で登録された) の前に制御を取得します。次に、制御は COBOL の USE 宣言部分に移ります。

ILE 条件ハンドラーによって、実行時に例外ハンドラーを**登録**することができます。ILE 条件ハンドラーは特定の呼び出しスタック項目に登録されます。ILE 条件ハンドラーを登録するには、 ユーザー作成条件ハンドラー登録 (CEEHDLR) バインド可能 API を使います。この API によって、例外が発生したときに制御が渡るプロシージャーを実行時に識別することができます。CEEHDLR API には、言語の中でプロシージャー・ポインターを宣言し設定する機能が必要です。 CEEHDLR は、組み込み関数としてインプリメントされています。したがって、そのアドレスを指定することや、プロシージャー・ポインターを通して呼び出すことはできません。ILE 条件ハンドラーを**抹消する**には、ユーザー作成条件ハンドラー抹消 (CEEHDLU) バインド可能 API を呼び出します。

OPM および ILE は、HLL 固有のハンドラーをサポートします。 **HLL 固有ハンドラー**は、エラー処理のために定義された言語機能です。たとえば、ILE C シグナル関数は、例外メッセージの処理に使用することができます。 RPG における HLL 固有のエラー処理には、単一ステートメントに関する例外を処理する機能 (E エクステンダー)、ステートメントのグループに関する例外を処理する機能 (MONITOR)、およびプロシージャー全体に関する例外を処理する機能 (*PSSR および INFSR サブルーチン) が含まれます。 COBOL の HLL 固有のエラー処理には、入出力エラー処理のための USE 宣言や、ON SIZE ERROR や AT INVALID KEY などのステートメント有効範囲条件句が含まれます。

HLL 固有のエラー処理と ILE に追加された例外ハンドラー・タイプの両方を使用する場合、例外ハンドラーの優先順位が重要になります。

52 ページの図 26 は、プロシージャー P2 に関する呼び出しスタック項目を示しています。この例では、単一の呼び出しスタック項目に対して、3 つのタイプのハンドラーがすべて定義されています。これは一般的な例ではありませんが、 3 つのタイプをすべて定義することは可能です。3 つのタイプがすべて定義されているので、例外ハンドラーの優先順位が定義されています。図は、この優先順位を示しています。例外メッセージが送信されると、例外ハンドラーは以下の順に呼び出されます。

1. 直接モニター・ハンドラー

   最初に呼び出し順で、次にその呼び出しの中の相対的順序でハンドラーが選択されます。呼び出しの中で、すべての直接モニター・ハンドラーと COBOL ステートメント範囲条件付き命令が、ILE 条件ハンドラーの前に制御を獲得します。同様に、ILE 条件ハンドラーが、他の HLL 固有ハンドラーの前に制御を獲得します。

例外を引き起こしたステートメントの周囲で直接モニター・ハンドラーが宣言されている場合、これらのハンドラーが HLL 固有ハンドラーの前に呼び出されます。たとえば 52 ページの図 26 のプロシージャー P2 が HLL 固有モニターを持ち、プロシージャー P1 が直接モニター・ハンドラーを持っている場合には、P2 のハンドラーが P1 の直接ハンドラーの前と考えられます。

直接モニターは、字句単位でネストされます。最深部にネストされた直接モニターにより指定されたハンドラーは、同じ優先順位を持つ複数のネストされたモニターの中で最初に選択されます。

2. ILE 条件ハンドラー

   ILE 条件ハンドラーが呼び出しスタック項目に対して登録されている場合、このハンドラーが 2 番目に呼び出されます。複数の ILE 条件ハンドラーを登録することができます。例では、プロシージャー P5 とプロシージャー P6 が ILE 条件ハンドラーです。1 つの呼び出しスタック項目に対して複数の ILE 条件ハンドラーが登録されている場合、システムは後入れ先出し (LIFO) の順にこれらのハンドラーを呼び出します。COBOL ステートメント有効範囲の条件付き命令を HLL 固有のハンドラーとすると、これらの命令が ILE 条件ハンドラーに優先します。一般的に、HLL 固有ハンドラーの優先順位は、ダイレクト・モニター・ハンドラーおよび異常事態処理ルーチンに続いて最も低くなります。1 つの例外は HLL 固有ハンドラーである COBOL ステートメント有効範囲の条件命令 (condition imperatives) であり、ダイレクト・モニター・ハンドラーと同じ優先順位を持っています。

3. HLL 固有ハンドラー

   HLL 固有のハンドラーは最後に呼び出されます。

例外メッセージの処理が行われたことを示すように例外メッセージが変更されると、システムは例外処理を終了します。直接モニター・ハンドラーまたは ILE 条件ハンドラーを使用している場合、例外メッセージの変更はユーザーの責任になります。いくつかの制御アクションが選択可能です。たとえば、ハンドルを制御アクションとして指定できます。例外メッセージが未処理のまま残っている限り、システムは、前に定義された優先順位に従って例外ハンドラーの検索を続行します。 現行呼び出しスタック項目内で例外が処理されない場合、前の呼び出しスタック項目へのパーコレーションが行われます。HLL 固有エラー処理を使用していない場合、ILE HLL は、前の呼び出しスタック項目で例外処理が続行するのを許すよう選択できます。

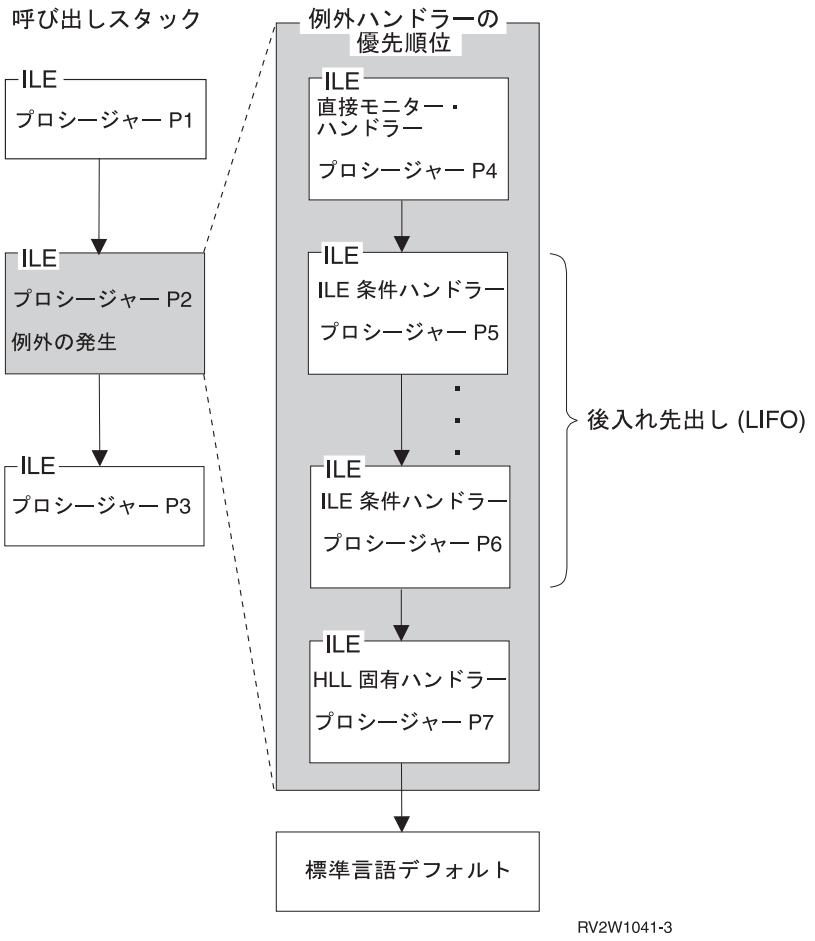

図 26. 例外ハンドラーの優先順位

ILE 条件

システム間の整合性の向上のために、ILE は、エラー条件を処理するための機能を定義しています。ILE の**条件**は、HLL 内のエラー条件に関する、システムから独立した表示です。OS/400 の場合、各 ILE 条件には対応する例外メッセージがあります。ILE の特定の条件は、特定の条件トークンによって示されます。**条件トークン**は、複数のシステム間での整合性をもつ 12 バイトのデータ構造です。このデータ構造には、条件と、その基礎になる例外メッセージとを関連付けるための情報が入っています。

システム間での整合性をもつプログラムを作成するには、ILE 条件ハンドラーおよび ILE 条件トークンを使用する必要があります。ILE 条件の詳細については 135 ページの『第 9 章 例外および条件管理』 を参照してください。

データ管理機能の有効範囲指定の規則

データ管理機能の有効範囲指定の規則は、データ管理機能リソースの使用を制御します。これらのリソースは、プログラムがデータ管理機能を処理するための一時オブジェクトです。たとえば、プログラムがファイルをオープンすると、プログラムをファイルに接続するために、オープン・データ・パス (ODP) と呼ばれるオブジェクトが作成されます。プログラムが、ファイルの処理方法を変更するために指定変更を作成すると、システムは指定変更オブジェクトを作成します。

データ管理機能の有効範囲指定の規則は、呼び出しスタックで実行中の複数のプログラムまたはプロシージャーによって、1 つのリソースがいつ共用可能であるかを決定します。たとえば、SHARE(*YES) パラメーター値を指定して作成されたオープン・ファイルまたはコミットメント定義オブジェクトは、同時に多くのプログラムによって使用可能です。データ管理機能リソースを共用できるかどうかは、データ管理機能リソースの有効範囲指定のレベルによって異なります。

データ管理機能の有効範囲指定の規則は、リソースの存在も決定します。システムはジョブ内の使用されていないリソースを、有効範囲指定の規則に基づいて自動的に削除します。この自動クリーンアップ操作の結果として、ジョブが使用するストレージが縮小され、ジョブのパフォーマンスが改善されます。

ILE は、OPM プログラムと ILE プログラムに関するデータ管理機能の有効範囲指定の規則を形式化して、以下の有効範囲指定レベルに分けています。

呼び出し
活動化グループ
ジョブ

使用中のデータ管理機能リソースによって異なりますが、1 つ以上の有効範囲指定レベルを明示的に指定することができます。有効範囲指定レベルを選択しないと、システムはいずれかのレベルをデフォルトとして選択します。

それぞれのデータ管理機能リソースがどのように有効範囲指定レベルをサポートしているかについては 151 ページの『第 11 章 データ管理機能の有効範囲指定』を参照してください。

呼び出しレベルの有効範囲指定

呼び出しレベルの有効範囲指定は、データ管理機能リソースがそのリソースを作成した呼び出しスタック項目に接続されると設定されます。 54 ページの図 27 に例を示します。呼び出しレベルの有効範囲指定は、通常、デフォルトの活動化グループ内で実行されるプログラムのデフォルトの有効範囲指定レベルです。この図で、OPM プログラム A、OPM プログラム B、または ILE プロシージャー P2 は、それぞれのファイル F1、F2、または F3 をクローズしないで戻る可能性があります。データ管理は、各ファイルの ODP を、ファイルをオープンした呼び出しレベル番号に関連付けます。RCLRSC コマンドにより、このコマンドに渡される特定の呼び出しレベル番号に基づいてファイルをクローズすることができます。

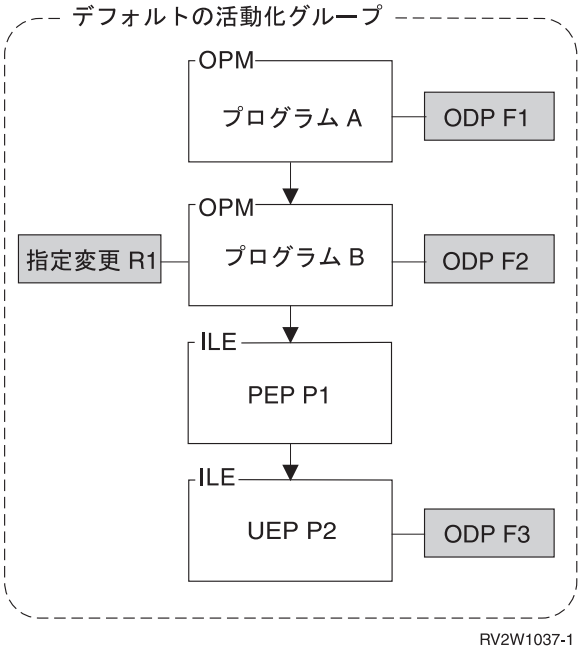

図 27. 呼び出しレベルの有効範囲指定： (ODP および指定変更の有効範囲は、呼び出しレベルにすることができます。)

有効範囲が特定の呼び出しレベルである指定変更は、対応する呼び出しスタック項目が戻ると削除されます。 指定変更は、その指定変更を作成した呼び出しレベルより下のいずれかの呼び出しスタック項目によって共用されている可能性があります。

活動化グループ・レベルの有効範囲指定

活動化グループ・レベルの有効範囲指定は、データ管理機能リソースが、そのリソースを作成した ILE プログラムまたは ILE サービス・プログラムの活動化グループに接続されると設定されます。活動化グループが削除されると、データ管理は、活動化グループ内で実行中のプログラムによってオープンされたままのその活動化グループに関連するすべてのリソースをクローズします。 55 ページの図 28 は、活動化グループ・レベルの有効範囲指定の例を示しています。活動化グループの有効範囲指定は、デフォルトの活動化グループ以外で実行中の ILE プロシージャーによって使用される大部分のタイプのデータ管理機能リソースのデフォルトの有効範囲指定レベルです。たとえば、この図はファイル F1、F2、および F3 の ODP と有効範囲が活動化グループである指定変更 R1 を示しています。

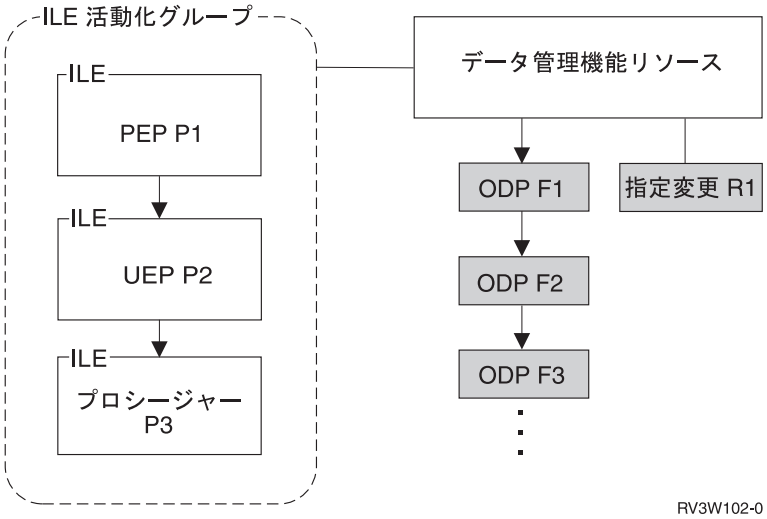

図 28. 活動化グループ・レベルの有効範囲指定： (ODP および指定変更の有効範囲は活動化グループにすることができます。)

有効範囲が活動化グループであるデータ管理機能リソースを共用できるのは、その活動化グループ内で実行中のプログラムだけです。これによって、アプリケーションの分離と保護が可能になります。たとえば、図のファイル F1 が SHARE(*YES) パラメーター値を指定してオープンされているとします。ファイル F1 は、同じ活動化グループ内で実行中のどの ILE プロシージャーによっても使用可能です。異なる活動化グループ内でファイル F1 に対してもう 1 つのオープン操作を行うと、このファイルに対する 2 番目の ODP が作成されます。

ジョブ・レベルの有効範囲指定

ジョブ・レベルの有効範囲指定は、データ管理機能リソースがジョブに接続されると設定されます。ジョブ・レベルの有効範囲指定は、OPM プログラムと ILE プログラムの両方で使用可能です。ジョブ・レベルの有効範囲指定によって、異なる活動化グループ内で実行中のプログラム間でデータ管理機能リソースを共用することができます。前のトピックで記述したように、リソースの有効範囲指定を活動化グループに設定すると、そのリソースの共用が当該活動化グループで実行中のプログラムに限定されます。ジョブ・レベルの有効範囲指定によって、ジョブ内で実行中のすべての ILE プログラムと OPM プログラムの間でのデータ管理機能リソースの共用が可能になります。

56 ページの図 29 は、ジョブ・レベルの有効範囲指定の例を示しています。プログラム A が、ジョブ・レベルの有効範囲を指定してファイル F1 をオープンしています。 このファイルの ODP は当該ジョブに接続されています。ジョブが終了しない場合、ファイルはシステムによってクローズされません。 ODP が SHARE(YES) パラメーター値を指定して作成されている場合、すべての OPM プログラムまたは ILE プロシージャーがファイルを共用できる可能性があります。

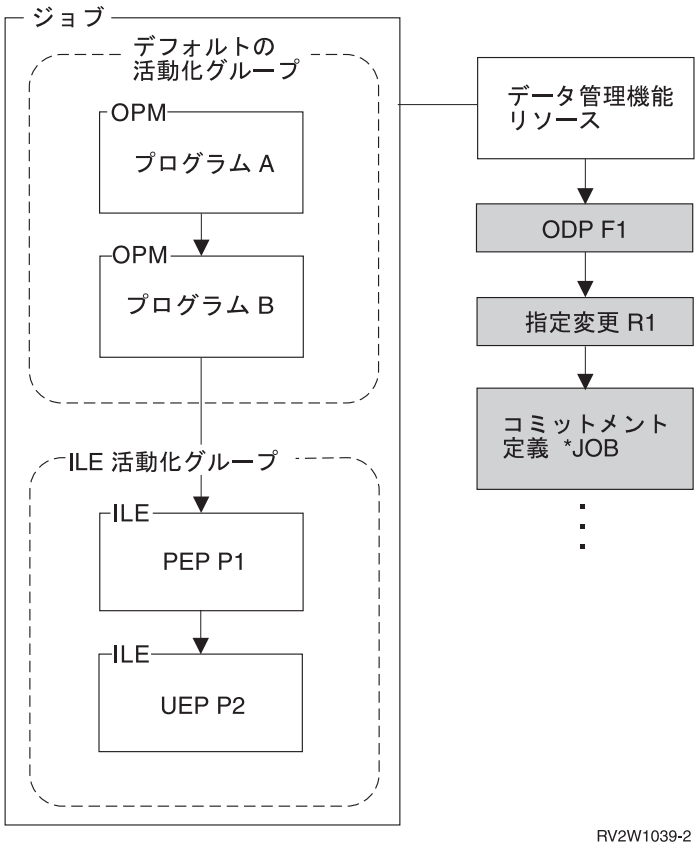

図 29. ジョブ・レベルの有効範囲指定： (ODP、指定変更、およびコミットメント定義の有効範囲はジョブ・レベルにすることができます。)

有効範囲がジョブ・レベルである指定変更は、ジョブ内のすべてのオープン・ファイル操作に影響を与えます。この例で、指定変更 R1 はプロシージャー P2 によって作成された可能性があります。ジョブ・レベルの指定変更は、明示的に削除されるまで、またはジョブが終了するまで活動状態のままです。ジョブ・レベルの指定変更は、組み合わせが行われる場合の優先順位が最高の指定変更です。なぜなら、複数の指定変更が呼び出しスタック上に存在する場合、呼び出しレベルの指定変更が一緒にマージされるからです。

データ管理機能の有効範囲指定レベルは、指定変更コマンドまたはコミットメント制御コマンドで有効範囲指定パラメーターを使用して、また各種の API を介して明示的に指定することができます。有効範囲指定の規則を使用するデータ管理機能リソースの詳細なリストが 151 ページの『第 11 章 データ管理機能の有効範囲指定』にあります。

第 4 章 テラスペースおよび単一レベル・ストア

ILE プログラムを作成する場合、iSeries サーバー上で 2 つのタイプのストレージであるテラスペースまたは単一レベル・ストアを選択することができます。本章では、より新しいテラスペース・オプションについて説明します。ILE プログラムは、デフォルトでは単一レベル・ストアを使用します。

テラスペースの特性

テラスペースは、ジョブにとってローカルにある大きな一時スペースです。テラスペースは 1 つの連続するアドレス・スペースを提供しますが、実際には、多くの個別に割り振られたエリアからなり、その間に割り振られていないエリアがあってもかまいません。テラスペースは、ジョブが開始してから終了するまでの間のみ存在します。

テラスペースは、スペース・オブジェクトではありません。このことは、テラスペースがシステム・オブジェクトではなく、システム・ポインターを使用して参照することはできないことを意味しています。ただし、テラスペースは、スペース・ポインターによってアドレッシングが可能です。

以下の表は、テラスペースと単一レベル・ストアの比較を示しています。

表 2. テラスペースと単一レベル・ストアの比較

属性	テラスペース	単一レベル・ストア
局所性	局所的な処理: 通常は所有しているジョブでのみアクセス可能。	グローバル: これに対するポインターを持っているすべてのジョブでアクセス可能。
サイズ	合計 1 TB	16 MB 単位のものが多数。
メモリー・マッピングのサポート	可	不可
8 バイト・ポインターによるアドレッシング	可	不可
ジョブ間の共用のサポート	共用メモリー API (たとえば、shmat または mmap) を使用して実行する必要がある。	他のジョブへポインターを渡すことによって、または共用メモリー API を使用することによって実行することができる。

プログラムでのテラスペースの使用可能化

ILE プログラムはデフォルトでは単一レベル・ストアを使用します。テラスペース・アドレスを処理するには、プログラムをテラスペース*使用可能* にする必要があります。テラスペース使用可能プログラムは、以下のようなさまざまなコンテキストで、テラスペース・アドレスを処理することができます。

- テラスペース・ヒープ・ストレージを割り振る要求からアドレスが戻された場合。
- テラスペース共用メモリーを割り振る要求からアドレスが戻された場合。
- 別のプログラムからアドレスが渡された場合。

以下のコンパイラーは、テラスペース使用可能コードを生成します。

- ILE C (モジュールおよびプログラムを作成する時に TERASPACE(*YES) を選択する)
- ILE C++ (モジュールおよびプログラムを作成する時に TERASPACE(*YES) を選択する)
- ILE RPG (V4R4 以降はテラスペース使用可能がデフォルト)
- ILE COBOL (V4R4 以降はテラスペース使用可能がデフォルト)
- ILE CL (V5R1 以降はテラスペース使用可能がデフォルト)

ILE C および C++ コンパイラーには、ソース・コードを変更せずに、ストレージ・インターフェースのテラスペース・バージョンを使用できるようにする、作成コマンドの TERASPACE (*YES *TSIFC) オプションが用意されています。たとえば、`malloc()` は `_C_TS_malloc()` にマップされます。

これらのコンパイラー・オプションの詳細については、「WebSphere Development Studio ILE C/C++ Programmer's Guide」 を参照してください。

OPM プログラムをテラスペース使用可能にするには、CHGPGM コマンドを使用します。

プログラム・ストレージ・モデルの選択

テラスペース・ストレージ・モデル を使用できるようにモジュールおよびプログラムを作成することによって、テラスペース使用可能の機能をさらに活用できます。テラスペース・ストレージ・モデルのプログラムは、自動、静的、および固定のストレージとしてテラスペースを使用します。テラスペース・ストレージ・モデルを選択した場合には、このようなタイプのストレージとして、より大きなエリアを使用することができます。テラスペース・ストレージ・モデルのプログラムは、このようなストレージをアドレッシングするために 8 バイトのポインターを使用することもできます。テラスペース・ストレージ・モデルについての詳細は 65 ページの『テラスペース・ストレージ・モデルの使用』を参照してください。

モジュールおよびプログラムのためのストレージ・モデルとして、単一レベル・ストア (*SNGLVL) またはテラスペース (*TERASPACE) の 2 つのうちの 1 つを指定するオプションがあります。サービス・プログラムが、その中で実行している活動化グループのストレージ・モデルを継承できるようにすることを選択することもできます。以下のトピックでは、テラスペース・ストレージ・モデルについて説明します。

テラスペース・ストレージ・モデルの指定

テラスペース・ストレージ・モデルを選択するには、コードをコンパイルするときに、以下のオプションを指定してください。

1. モジュールをテラスペース使用可能にする。モジュールの作成時に、TERASPACE パラメーターに *YES を指定してください。

   DSPMOD、DSPPGM、および DSPSRVPGM コマンドを使用すれば、それぞれモジュール、プログラム、およびサービス・プログラムのテラスペース属性を表示することができます。各モジュールの詳細を表示するためのオプション 5 を使用すれば、 DETAIL(*MODULE) によりテラスペース属性がわかります。
2. 使用している ILE プログラム言語のモジュールの作成コマンドのストレージ・モデル (STGMDL) パラメーターに *TERASPACE または *INHERIT を指定する。
3. CRTPGM または CRTSRVPGM コマンドの STGMDL パラメーターに *TERASPACE を指定する。この選択は、プログラムにバインドするモジュールのストレージ・モデルと互換性があるものでなければなりません。詳細については 61 ページの『モジュールのバインディングに関する規則』を参照してください。

   1 つだけのモジュールを含むバインド済みプログラムを 1 つのステップで作成する CRTBNDC および CRTBNDCPP コマンドの STGMDL パラメーターに、*TERASPACE を指定することもできます。

   CRTSRVPGM コマンドの場合、STGMDL パラメーターに *INHERIT を指定することもできます。これにより、サービス・プログラムが活動化される活動化グループ内で使用中であるストレージのタイプに従って、単一レベル・ストアまたはテラスペースを使用できるようにする方法で、サービス・プログラムが作成されます。

   *INHERIT 属性の使用によって、柔軟性がかなり向上しますが、ACTGRP パラメーターに *CALLER を指定する必要も生じます。この場合、サービス・プログラムは単一レベル・ストアまたはテラスペースのいずれかを使用して活動化できるので、プログラムのコードは両方の状態を効果的に処理できなければならないことに注意してください。たとえば、すべての静的変数の合計サイズは、単一レベル・ストアの小さい方の限界値以下でなければなりません。

表 3. 特定のタイプのプログラムに許可されるストレージ・モデル

プログラムのストレージ・モデル	プログラム・タイプ		
	OPM *PGM	**ILE *PGM**	**ILE *SRVPGM**
*TERASPACE	不可	可	可
*INHERIT	不可	不可	可、ただし ACTGRP(*CALLER) を指定した場合のみ
*SNGLVL	可	可	可

互換性のある活動化グループの選択

活動化グループは、活動化グループが作成される原因となったルート・プログラムのストレージ・モデルを反映します。ストレージ・モデルは、プログラムに提供される自動、静的、および固定のストレージのタイプを決定します。

単一レベル・ストア・ストレージ・モデルのプログラムは、単一レベル・ストアの自動、静的、および固定のストレージを受け取ります。デフォルトで、これらのプログラムはヒープ・ストレージ用の単一レベル・ストアも使用します。

テラスペース・ストレージ・モデルのプログラムは、テラスペースの自動、静的、および固定のストレージを受け取ります。デフォルトで、これらのプログラムはヒープ・ストレージ用のテラスペースも使用します。

テラスペース・ストレージ・モデルを使用するプログラムは、そのルート・プログラムが単一レベル・ストア・ストレージ・モデルを使用する活動化グループでは活動化することはできません。単一レベル・ストア・ストレージ・モデルを使用するプログラムは、そのルート・プログラムがテラスペース・ストレージ・モデルを使用する活動化グループでは活動化することはできません。

以下の表は、ストレージ・モデルと活動化グループのタイプの関係を要約しています。

表 4. ストレージ・モデルと活動化グループの関係

プログラムのストレージ・モデル	活動化グループの属性			
	***CALLER**	***DFTACTGRP**	***NEW**	**名前付き**
*TERASPACE	可。クライアント・プログラムもテラスペース・ストレージ・モデルを使用して作成されていることを確認してください。	許可されない。デフォルトの活動化グループは *SNGLVL のみです。	可	可
*INHERIT	可	許可されない。	許可されない。	許可されない。
*SNGLVL	可	可	可	可

プログラムまたはサービス・プログラムを実行する活動化グループを選択する場合には、以下のガイドラインに従ってください。

- サービス・プログラムで STGMDL(*INHERIT) を指定している場合には、ACTGRP(*CALLER) を指定してください。
- プログラムで STGMDL(*TERASPACE) を指定する場合には、以下のようにしてください。
  - ACTGRP(*NEW) または名前付き活動化グループを指定してください。
  - プログラムを呼び出すすべてのプログラムがテラスペース・ストレージ・モデルを使用していることが明らかな場合のみ ACTGRP(*CALLER) を指定してください。

ストレージ・モデル間の相互作用

ストレージ・モデルを使用するモジュールとプログラム間の整合性が必要です。プログラム間の相互作用が正しく行われるための規則を以下に示します。

- 61 ページの『モジュールのバインディングに関する規則』
- 61 ページの『サービス・プログラムへのバインディングに関する規則』
- 61 ページの『プログラムおよびサービス・プログラムの活動化の規則』
- 62 ページの『プログラムおよびプロシージャー呼び出しの規則』

モジュールのバインディングに関する規則

以下の表は、モジュールのバインディングに関する規則を示しています。

バインディングの規則: 指定されたストレージ・モデルを使用するプログラムへのモジュール M のバインディング		作成されるプログラムのストレージ・モデル		
		テラスペース	継承 (サービス・プログラムのみ)	単一レベル・ストア
M	テラスペース	テラスペース	エラー	エラー
	継承	テラスペース	継承	単一レベル・ストア
	単一レベル・ストア	エラー	エラー	単一レベル・ストア

サービス・プログラムへのバインディングに関する規則

以下の表は、ターゲット・サービス・プログラムへのプログラムのバインドに関する規則を示しています。

サービス・プログラムのバインド規則: 呼び出し側プログラムまたはサービス・プログラムがターゲット・サービス・プログラムにバインドできるかどうか。		ターゲットのサービス・プログラムのストレージ・モデル		
		テラスペース	継承	単一レベル・ストア
呼び出し側プログラムまたはサービス・プログラムのストレージ・モデル	テラスペース	可	可	可[2]
	継承[1]	可[3]	可	可[3]
	単一レベル・ストア	可[2]	可	可

注:

1. ストレージ・モデルの継承を指定できるのは、サービス・プログラムのみです。
2. ターゲット・サービス・プログラムは別個の活動化グループで実行する必要があります。たとえば、ターゲット・サービス・プログラムは ACTGRP(*CALLER) 属性を持つことができません。単一の活動化グループ内でストレージ・モデルを混合することはできません。
3. 継承ストレージ・モデルを使用するサービス・プログラムを、単一レベル・ストアまたはテラスペースのサービス・プログラムにバインドすることは許可されますが、操作が最終的に成功するかどうかは、活動化時まで分からない場合があります。ターゲット・サービス・プログラムが ACTGRP(*CALLER) 属性を持っている場合には、呼び出し側のサービス・プログラム (ストレージ・モデルの継承が指定されている) は、ターゲット・サービス・プログラムと互換性がある活動化グループで活動化する必要があります。ターゲット・サービス・プログラムは ACTGRP(*CALLER) を指定できません。すなわち、呼び出し側プログラムまたはサービス・プログラムとして、同じ名前の活動化グループを指定できません。

プログラムおよびサービス・プログラムの活動化の規則

継承ストレージ・モデルを指定するテラスペース使用可能サービス・プログラムは、単一レベル・ストア・ストレージ・モデルまたはテラスペース・ストレージ・

モデルを使用するプログラムが実行される活動化グループで活動化することができます。それ以外の場合、サービス・プログラムのストレージ・モデルは、活動化グループ内で実行される他のプログラムのストレージ・モデルと一致している必要があります。

プログラムおよびプロシージャー呼び出しの規則

異なるストレージ・モデルを使用するプログラムとサービス・プログラムが、相互に動作することができます。本章で説明した規則および制約事項に従っていれば、このようなプログラムとサービス・プログラムをバインドして、データを共用することができます。

テラスペース使用可能でないコードは、テラスペース・アドレスを処理できません。そのようなことを試みると、通常は MCH0607 (サポートされないスペースが使用された) という例外が発生します。

ストレージ・モデルを継承するためのサービス・プログラムの変換

ストレージ・モデルを継承するようにサービス・プログラムを変換することによって (STGMDL パラメーターに *INHERIT を指定して)、テラスペース環境または単一レベル・ストア環境のどちらでも、サービス・プログラムを使用可能にすることができます。既存のサービス・プログラムがテラスペース・ストレージ・モデルを使用できるようにするには、以下のステップを実行してください。

1. 継承ストレージ・モデルを使用して、すべてのモジュールを作成する。いずれかのモジュールが単一レベル・ストアまたはテラスペースのストレージ・モデルを使用して作成されている場合には、継承ストレージ・モデルを使用してサービス・プログラムを作成することはできません。
2. プログラムのコードが、テラスペースと単一レベル・ストア・ストレージの間のポインターを処理し、効果的に管理することを確認する。詳しくは 66 ページの『テラスペースの使用: 最適な方法』を参照してください。
3. 継承ストレージ・モデルを使用してサービス・プログラムを作成する。ACTGRP パラメーターに *CALLER も指定してください。

プログラムの変更および更新: テラスペースに関する考慮事項

特定の状況で、プログラムをテラスペース使用可能に変更または更新することができます。プログラムを更新するために使用するモジュールのストレージ・モデルには制限があります。

プログラムの変更:

CHGPGM および CHGSRVPGM コマンドを使用すれば、テラスペース使用不能プログラムをテラスペース使用可能プログラムに変換することができます。ILE プログラムおよびそのすべてのバインド済みモジュールのターゲット・リリースが V4R4M0 以降である場合に、この変換を実行することができます。ターゲット・リリースが V4R4M0 以降である OPM プログラムにも、この変換を実行することができます。

プログラムの更新:

同じストレージ・モデルを使用している限り、プログラム内でモジュールの追加および置換を行えます。ただし、更新コマンドを使用して、バインド済みモジュールまたはプログラムのストレージ・モデルを変更することはできません。

C および C++ コード内の 8 バイト・ポインターの利用

8 バイト・ポインターはテラスペースのみを指すことができます。8 バイト・プロシージャー・ポインターは、テラスペースを介して、活動状態のプロシージャーを参照します。8 バイト・タイプのポインターのみがスペースおよびプロシージャーのポインターです。

これとは対照的に、16 バイトのポインターには多くのタイプがあります。以下の表は、8 バイトと 16 バイトのポインターの比較を示しています。

表 5. ポインターの比較

特性	8 バイト・ポインター	16 バイト・ポインター
長さ (必要なメモリーの量)	8 バイト	16 バイト
タグ	なし	あり
位置合わせ	バイト位置合わせ (すなわち、パック構造) が許可される。パフォーマンスのためには「本来の」位置合わせ (8 バイト) が望ましい。	常に 16 バイト。
アトミシティ	8 バイト位置合わせの場合は、ロードと保管のアトミック操作。集合コピー操作に適用してはならない。	ロードと保管のアトミック操作。集合の一部である場合には、アトミック・コピー。
アドレス可能な範囲	テラスペース・ストレージ	テラスペース・ストレージ + 単一レベル・ストレージ
ポインターの内容	テラスペースへのオフセットを示す 64 ビット値。これには、有効アドレスは入らない。	16 バイトのポインター・タイプ・ビットおよび 64 ビットの有効アドレス。
参照の局所性	ローカル・ストレージ参照を処理する。(8 バイトのポインターは、ストレージ参照が発生するジョブのテラスペースのみを参照できる。)	ローカルまたは単一レベル・ストア・ストレージの参照を処理する。 (16 バイト・ポインターは、別のジョブが論理的に所有しているストレージを参照できる。)
許可される操作	スペース・ポインターおよびプロシージャー・ポインターに許可されるポインター固有の操作、および非ポインター・ビューを使用する、2 進データに適切なすべての算術演算および論理演算を、ポインターを無効にせずに使用することができる。	ポインター固有の操作のみ。
最速のストレージ参照	不可	可

表 5. ポインターの比較 (続き)

特性	8 バイト・ポインター	16 バイト・ポインター
最速のロード、保管、およびスペース・ポインター演算	可 (EAO オーバーヘッドの回避を含む)。	不可
ポインターにキャストされる場合に保存される 2 進値のサイズ	8 バイト	4 バイト
例外ハンドラーまたは取り消しハンドラーであるプロシージャーによって、パラメーターとして受け入れられる。	不可	可

C および C++ コンパイラーにおけるポインター・サポート

IBM C または C++ コンパイラーを使用してコードをコンパイルする場合に、8 バイト・ポインターを完全に利用するには、STGMDL(*TERASPACE) および DTAMDL(*LLP64) を指定してください。

C および C++ コンパイラーは、以下のポインター・サポートも提供します。

- 8 バイトまたは 16 バイトのポインターを明示的に宣言するための、以下の構文。
  - 8 バイト・ポインターを `char * __ptr64` として宣言する。
  - 16 バイト・ポインターを `char * __ptr128` として宣言する。
- C および C++ プログラミング環境に固有であるデータ・モデル を指定するためのコンパイラー・オプションおよび pragma。データ・モデルは、明示的修飾子のいずれかがない場合に、ポインターのデフォルトのサイズに影響を与えます。データ・モデルには、以下の 2 つの選択肢があります。
  - `P128` (4-4-16 とも呼ばれる) [1]
  - `LLP64` (4-4-8 とも呼ばれる) [2]

ポインターの変換

IBM C および C++ コンパイラーは、関数および変数の宣言に基づいて、必要に応じて、`__ptr128` から `__ptr64` への変換およびその逆の変換を行います。以下のことに特に注意してください。

- 単一レベル・ストア・ストレージを指す `_ptr128` は、任意の `__ptr64` 値に変換される。
- テラスペース使用可能でないコードはテラスペースにアクセスできない。
- ポインター間パラメーターを使用するインターフェースには特殊な処理が必要である。

コンパイラーは、ポインターの長さと一致させるためのポインター変換を自動的に挿入します。たとえば、関数に対するポインター引き数が、関数のプロトタイプ内のポインター・パラメーターの長さと一致しない場合に、変換が挿入されます。あ

1. ここで、4-4-16 = sizeof(int) - sizeof(long) - sizeof(pointer)

2. ここで、4-4-8 = sizeof(int) - sizeof(long) - sizeof(pointer)

るいは、異なる長さのポインターを比較する場合、コンパイラーは暗黙的に 8 バイト・ポインターを 16 バイト・ポインターに変換してから比較します。コンパイラーは、キャストとして、明示的変換の指定も許可します。ポインター・キャストを追加する場合には、下記の点を考慮してください。

- 16 バイト・ポインターから 8 バイト・ポインターへの変換は、16 バイト・ポインターにテラスペース・アドレスまたはヌル・ポインター値が入っている場合にのみ有効です。それ以外の場合、MCH0609 例外がシグナルで示されるか、または、任意のテラスペース・オフセット値が戻されます。
- 16 バイトのポインターはタイプの変換ができませんが、16 バイトの OPEN ポインターには任意のポインター・タイプを入れることができます。対照的に、8 バイトの OPEN ポインターは存在しませんが、8 バイトのポインターは、スペース・ポインターとプロシージャー・ポインターの間で論理的に変換することができます。ただし、8 バイトのポインター変換は単にポインター・タイプのビューの変換なので、スペース・ポインターがプロシージャーを指すように設定されていない場合には、スペース・ポインターをプロシージャー・ポインターとして実際に使用することはできません。

ポインターと 2 進値の間の明示的なキャストを追加する場合には、8 バイト・ポインターと 16 バイト・ポインターの動作が異なることに注意してください。 8 バイト・ポインターは完全な 8 バイトの 2 進値を保持できますが、16 バイト・ポインターは、4 バイトの 2 進値しか保持できません。 2 進値を保持しているポインターに定義されている唯一の操作は、2 進数フィールドに戻す変換のみです。他のすべての操作は、ポインターとしての使用、別のポインター長への変換、およびポインターの比較も含めて、未定義です。したがって、たとえば、8 バイト・ポインターと 16 バイト・ポインターに同じ整数値が割り当てられ、8 バイト・ポインターが 16 バイト・ポインターに変換されてから、16 バイト・ポインターの比較が実行された場合、比較の結果は未定義であり、等しい結果にならない可能性があります。

長さが異なるポインターの比較は、16 バイト・ポインターがテラスペース・アドレスを保持し、8 バイト・ポインターもテラスペース・アドレスを保持している (すなわち、8 バイト・ポインターに 2 進値が含まれていない) 場合のみ定義されています。この場合、8 バイト・ポインターを 16 バイト・ポインターに変換し、2 つの 16 バイト・ポインターを比較することは有効です。その他のすべての場合、比較の結果は未定義になります。したがって、たとえば、16 バイトのポインターが 8 バイトのポインターに変換された後に、8 バイト・ポインターと比較された場合、結果は未定義です。

テラスペース・ストレージ・モデルの使用

理想的なテラスペース環境では、すべてのモジュール、プログラム、およびサービス・プログラムがテラスペース・ストレージ・モデルを使用します。しかし、実際のレベルでは、両方のストレージ・モデルを使用するモジュール、プログラム、およびサービス・プログラムが結合された環境を管理する必要があります。

このセクションでは、理想的なテラスペース環境に近づくためにインプリメントできる方法について説明します。またこのセクションでは、単一レベル・ストアとテラスペースを使用するプログラムが混合している環境で、問題が発生する可能性を最小にする方法についても説明します。

テラスペースの使用: 最適な方法

- *テラスペース・ストレージ・モデルのモジュールのみを使用する。*

  テラスペース・ストレージ・モデルまたは継承ストレージ・モデルを使用するモジュールを作成します。単一レベル・ストア・モジュールは、プログラムにバインドできないので、テラスペース環境には不適切です。このようなモジュールをどうしても使用する必要がある場合には (たとえば、このようなモジュールのソース・コードへのアクセス権を持っていない場合) 69 ページの『テラスペース使用のヒント』のシナリオ 9 を参照してください。

- *テラスペース・ストレージ・モデルまたは継承ストレージ・モデルを使用するサービス・プログラムのみにバインドする。*

  テラスペース・ストレージ・モデルのプログラムは、ほとんどすべての種類のサービス・プログラムにバインドすることができます。しかし、通常は、継承ストレージ・モデルまたはテラスペース・ストレージ・モデルのサービス・プログラムにのみバインドします。サービス・プログラムを制御する場合、すべてのサービス・プログラムを、それらをバインドするプログラムのストレージ・モデルを継承できるように作成する必要があります。一般的に、IBM サービス・プログラムは、このように作成されています。特に、作成したサービス・プログラムをサード・パーティーのプログラマーに提供する計画であれば、これと同じことを行う必要があります。単一レベル・ストア・サービス・プログラムにどうしてもバインドする必要があれば 69 ページの『テラスペース使用のヒント』のシナリオ 10 を参照してください。

- *テラスペース使用可能プログラムのみを呼び出す。*

  プログラムが外部プログラムを呼び出す可能性があります。テラスペース使用可能でないプログラムを呼び出し、パラメーターがテラスペースにあると、呼び出し先プログラムは失敗します。この考慮事項は、ユーザー出口プログラムにも適用されます。

  さらに、呼び出したテラスペース使用可能プログラムが、テラスペース使用不能なプログラムまたはサービス・プログラムにテラスペース・アドレスを渡さないことを確認しておく必要があります。それ以外の場合も、このトピックで概説した「最適な方法」に従う必要がある場合があります。テラスペース使用不能なプログラムを呼び出す必要がある場合、または呼び出し中のプログラムがテラスペース使用可能であるかどうか判別できない場合でも 69 ページの『テラスペース使用のヒント』のシナリオ 9 の各ステップを実行すれば、プログラムを呼び出すことができます。

- *テラスペース使用可能プロシージャーに対してのみポインター呼び出しを行う。*

  プログラムのコードは、プロシージャー・ポインターを入手でき、これを使用してプロシージャーを呼び出すことができます。呼び出し先のプロシージャーがテラスペース使用可能プログラムまたはサービス・プログラムにあることを確認してください。さらに、このプロシージャーが、テラスペース使用不能であるかもしれないプログラムに、テラスペース・アドレスを渡さないことを確認してください。テラスペース・アドレスを渡している場合、または渡しているかどうかを判別できない場合には 69 ページの『テラスペース使用のヒント』のシナリオ 9 で説明したガイドラインに従ってください。

  プロシージャーを含んでいるプログラムまたはサービス・プログラムは、すべてのモジュールをテラスペース使用可能にする必要があります。さもないと、プロシージャー・ポインター呼び出しは実行時に MCH4443 で失敗します。プログラ

ム内のすべてのモジュールがテラスペース使用可能である場合、呼び出し先プロシージャーはテラスペース使用可能になります。

このトピックに示したガイドラインに従った場合には、プログラム内でテラスペースを使用することができます。ただし、デフォルトでは単一レベル・ストアが使用されるので、テラスペースを使用する場合は、十分注意してコーディングする必要があります。以下のトピックは、テラスペースを使用する場合に行うことができないこと、および行ってはならないことを示しています。システムが特定のアクションの実行を防止する場合もありますが、テラスペースと単一レベル・ストアの相互作用の可能性を自分で管理しなければならない場合もあります。

- 『作成時のテラスペース・プログラムに対するシステムによる制御』
- 『活動化時のテラスペース・プログラムに対するシステムによる制御』
- 『実行時のテラスペース・プログラムに対するシステムによる制御』

注: 継承ストレージ・モデルを使用するサービス・プログラムも、テラスペースを使用するために活動化される可能性があるので、上記のガイドラインに従う必要があります。

作成時のテラスペース・プログラムに対するシステムによる制御

多くの場合、以下のアクションが行われないよう、システムが防止します。

- 単一レベル・ストアとテラスペースのストレージ・モデルのモジュールを、同じプログラムまたはサービス・プログラムに結合すること。
- デフォルトの活動化グループ (ACTGRP(*DFTACTGRP)) も指定している、テラスペース・ストレージ・モデルのプログラムまたはサービス・プログラムを作成すること。
- 単一レベル・ストア・プログラムを、*CALLER の活動化グループも指定しているテラスペース・ストレージ・モデルのサービス・プログラムにバインドすること。

活動化時のテラスペース・プログラムに対するシステムによる制御

活動化時に、単一レベル・ストア・ストレージ・モデルとテラスペース・ストレージ・モデルの両方のプログラムまたはサービス・プログラムが、同じ活動化グループでの活動化を試みるようにプログラムおよびサービス・プログラムが作成されているものと、システムが判断する場合があります。このような場合、システムは活動化のアクセス違反例外を送信して、活動化は失敗します。

実行時のテラスペース・プログラムに対するシステムによる制御

システムは、実行時まで以下の問題を検出できません。

- テラスペース・ストレージ・モデルのコードから、テラスペース使用不能である単一レベル・ストア・ストレージ・モデルのコードを呼び出すこと。
- テラスペース使用不能であるプログラムで、テラスペースへのポインターの使用を試みること。プログラムは、呼び出し先のプロシージャーを含むモジュールだけでなく、完全にテラスペース使用可能でなければなりません。

OS/400 のインターフェースおよびテラスペース

一般的に、OS/400 はテラスペース使用可能として作成されます。

ポインター・パラメーターを持つ OS/400 インターフェースは、一般的にタグ付きの 16 バイト (__ptr128) ポインターを期待します。

- コンパイラーは必要に応じてポインターを変換するので、8 バイト (__ptr64) ポインターを直接使用して、単一レベルのポインターのみによって (たとえば、void f(char*p);)、インターフェースを呼び出すことができます。システム・ヘッダー・ファイルを必ず使用してください。
- 複数レベルのポインターを持つインターフェース (たとえば、void g(char**p);) は通常、2 次レベルとして 16 バイト・ポインターを明示的に宣言することを要求します。しかし、8 バイト・ポインターを受け入れるバージョンが、このタイプの大部分のシステム・インターフェースに提供されているので、8 バイト・ポインターのみを使用するコードからの直接呼び出しが可能です。これらのインターフェースは、データ・モデル (LLP64) オプションを選択した場合に、標準ヘッダー・ファイルを介して使用可能になります。

テラスペース使用のためのバインド可能 API:

IBM では、テラスペースの割り振りや廃棄を行うためのバインド可能な API を用意しています。[3]

- _C_TS_malloc() はテラスペース内のストレージを割り振ります。
- _C_TS_free() はテラスペースの直前の割り振りを 1 つ解除します。
- _C_TS_realloc() はテラスペースの直前の割り振りのサイズを変更します。
- _C_TS_calloc() はテラスペース内のストレージを割り振り、それを 0 に設定します。

malloc()、free()、calloc()、および realloc() は、TERASPACE(*YES *TSIFC) コンパイラー・オプションを指定してコンパイルされていない限り、呼び出し側プログラムのストレージ・モデルに従って、単一レベル・ストレージまたはテラスペース・ストレージの割り振りまたは割り振り解除を行います。

POSIX 共用メモリーとメモリー・マップ・ファイルのインターフェースは、テラスペースを使用する可能性があります。プロセス間通信 API および shmget() インターフェースについて詳しくは、iSeries Information Center の**プログラミング**・カテゴリーおよび **API** サブカテゴリーの中の *UNIX-Type API* のトピックを参照してください。

テラスペースの使用時に発生する可能性がある問題

プログラムでテラスペースを使用する場合、発生する可能性がある問題を知る必要があります。

- *テラスペース・アドレスは、テラスペースが使用不能なプログラムまたはプロシージャーに渡すことができません。*テラスペースが使用不能なコードを呼び出した場合、プログラム呼び出しのパラメーターをテラスペースに置くことはできません。また、プログラムまたはプロシージャーの呼び出し用のパラメーターとし

3. テラスペース・コンパイラー・オプション TERASPACE(*YES *TSIFC) は、ILE C および C++ コンパイラーで提供されており、STGMDL(*SNGLVL) が指定された場合に自動的に malloc()、free()、calloc() および realloc() を対応するテラスペース版にマップします。

て渡されたポインターには、テラスペース・アドレスを入れることができません。状況によっては、このような試みは MCH0607、MCH3601 または MCH3602 例外を引き起こします。

- *いくつかの MI 命令は、テラスペース・アドレスを処理できません。*以下の命令でテラスペース・アドレスを使用する試みによって、MCH0607 例外が発生します。
  - CIPHER (一部のオプションのみが制限される)
  - MATBPGM
  - MATPG
  - SCANX (一部のオプションのみが制限される)
  - SETDP
  - SETDPADR
- *ジョブ間のアクセスは予測不能です。*テラスペースが 1 つのジョブでローカルとして定義されていても、別のジョブに渡されたテラスペースへのポインターが使用可能になる場合があります。ジョブ間のアクセスが機能しない場合にアプリケーション障害が発生するのを防止するために、テラスペースへのポインターを別のジョブに渡すことを避けてください。
- *有効アドレス・オーバーフロー (EAO) はパフォーマンスを低下させる可能性があります。*この状態は、16 バイト・ポインターに関するアドレス計算の結果、開始アドレスと異なる 16 MB 領域のアドレスが生成された場合に発生します。システム・ソフトウェアによって処理されるハードウェア割り込みが生成されます。このような多くの割り込みは、パフォーマンスに悪影響を与える可能性があります。16 バイト・ポインター演算を使用する場合は、テラスペース内の 16 MB 境界にまたがるようなアドレス計算を頻繁に行わないようにしてください。

テラスペース使用のヒント

テラスペース・ストレージ・モデルを使用して作業する場合に、以下のシナリオに出会う可能性があります。お勧めするソリューションを以下に示します。

- *シナリオ 1: 単一割り振りで 16 MB より大きな動的ストレージが必要な場合*

  `_C_TS_malloc` を使用するか、または `malloc` を使用する前にコンパイラーの作成コマンドに `TERASPACE(*YES *TSIFC)` を指定してください。これにより、すべてのテラスペース使用可能プログラムにヒープ・ストレージが提供されます。
- *シナリオ 2: 16 MB より大きな共用メモリーが必要な場合*

  テラスペース・オプションを指定して共用メモリー (`shmget`) を使用してください。
- *シナリオ 3: 大きなバイト・ストリーム・ファイルへの効率的なアクセスが必要な場合*

  メモリー・マップ・ファイル (`mmap`) を使用してください。

  どのテラスペース使用可能プログラムからでもメモリー・マップ・ファイルにアクセスできますが、最高のパフォーマンスを得るためには、テラスペース・ストレージ・モデルおよび 8 バイト・ポインターのデータ・モデルを使用してください。

- *シナリオ 4: 16 MB より大きな隣接する自動ストレージまたは静的ストレージが必要な場合*

  テラスペース・ストレージ・モデルを使用してください。8 バイト・ポインターまたは 16 バイト・ポインターによってテラスペースを使用できますが、最高のパフォーマンスを得るためには、8 バイト・ポインターのデータ・モデルを選択してください。
- *シナリオ 5: アプリケーションによるスペース・ポインターの頻繁な使用*

  テラスペース・ストレージ・モデルおよび 8 バイト・ポインターのデータ・モデルを使用して、メモリー・フットプリントを削減し、ポインター操作の速度を上げてください。
- *シナリオ 6: 別のシステムからコードを移植するので、16 バイト・ポインター使用に関する固有な問題を回避する必要がある場合*

  テラスペース・ストレージ・モデルおよび 8 バイト・ポインターのデータ・モデルを使用してください。
- *シナリオ 7: テラスペース・プログラム内で単一レベル・ストア・ストレージを使用する必要がある場合*

  テラスペース・ストレージ・モデル・プログラム内で、単一レベル・ストア・ストレージを使用する以外に選択肢がない場合があります。たとえば、テラスペース使用不能である呼び出し側プログラムまたはサービス・プログラムのパラメーターを保管するために、そのストレージが必要な場合があります。あるいは、プロセス間通信用のユーザー・データを保管しなければならない場合があります。単一レベル・ストア・ストレージは、以下のいずれかのソースから獲得できます。
  - CRTS MI 命令から獲得されるユーザー・スペース内のストレージ
  - 単一レベル・ストア・バージョンの `malloc`
  - プログラムに渡された単一レベル・ストア参照
  - ALCHS MI 命令から獲得された単一レベル・ストア・ストレージのヒープ・スペース
- *シナリオ 8: プログラム・コード内での 8 バイト・ポインターの利用*

  STGMDL(*TERASPACE) を使用してモジュールおよびプログラムを作成してください。テラスペースを参照するための 8 バイト・ポインターを入手するには、DTAMDL(*LLP64) または明示宣言 (`__ptr64`) を使用してください (テラスペースを指す 16 バイト・ポインターとは異なります)。そうすれば 63 ページの『C および C++ コード内の 8 バイト・ポインターの利用』にリストした利点を活用できます。
- *シナリオ 9: 単一レベル・ストア・ストレージ・モデル・モジュールの組み込み*

  単一レベル・ストアのモジュールとテラスペース・ストレージ・モデルのモジュールをバインドすることはできません。これを実行する必要がある場合には、最初に、テラスペース・ストレージ・モデルを使用 (または継承) するバージョンのモジュールの獲得を試みてから 66 ページの『テラスペースの使用: 最適な方法』の説明に従って、そのモジュールを単純に使用してください。その他には、以下の 2 つのオプションがあります。

– モジュールを別個のサービス・プログラムにパッケージする。サービス・プログラムは単一レベル・ストア・ストレージ・モデルを使用するので、下記のシナリオ 10 に示した方法を使用してこのサービス・プログラムを呼び出してください。
– モジュールを別個のプログラムにパッケージする。このプログラムは単一レベル・ストア・ストレージ・モデルを使用します。以下のシナリオ 11 に示した方法を使用して、このプログラムを呼び出してください。

- *シナリオ 10: 単一レベル・ストア・ストレージ・モデルのサービス・プログラムへのバインディング*

  2 つのサービス・プログラムを別個の活動化グループで活動化する場合、テラスペース・プログラムを、単一レベル・ストアを使用するプログラムにバインドすることができます。単一レベル・ストア・サービス・プログラムが ACTGRP(*CALLER) オプションを指定している場合には、このようなバインドを実行できません。

  単一レベル・ストア・サービス・プログラムがテラスペース使用不能でもある場合には、テラスペース使用可能バージョンの獲得を試みてください。獲得できない場合には、以下のシナリオ 11 を参照してください。

- *シナリオ 11: テラスペース使用不能であるプログラムまたはサービス・プログラムの呼び出し*

  まず、可能であれば、テラスペース使用不能であるプログラムを呼び出す必要がないようなプログラムのコーディングを試みてください。しかし、呼び出す必要があれば、単一レベル・ストア・ストレージに保管されているパラメーターのみを注意深く渡してください。

  そのためには、データをテラスペースから単一レベル・ストア・ストレージにコピーし、それをプログラムに渡し、プログラムから戻った後、すべての結果または変更されたストレージをテラスペースにコピーします。

  テラスペース・ストレージ・モデルの呼び出し元から、テラスペース使用不能であるプログラム内のプロシージャーへのプロシージャー・ポインター呼び出しは実行できません。

- *シナリオ 12: ポインター間パラメーターをもつ関数の呼び出し*

  ポインター間パラメーターをもつ、ある種の関数の呼び出しでは、DTMDL(*LLP64 オプション) を指定してコンパイルされたモジュールの特別な処理が必要です。ポインター・パラメーターに対して、8 バイト・ポインターと 16 バイト・ポインターとの間の暗黙的な変換が行なわれます。ポインター・パラメーターが指すデータ・オブジェクトについては、たとえそのポインター・ターゲットがポインターである場合でも、変換は行なわれません。たとえば、一般的に使用される P128 データ・モデルを行使する、ヘッダー・ファイルで宣言された **char**** インターフェースの使用では、データ・モデル LLP64 で作成されたモジュール内に、ある種のコーディングが必要です。この場合、必ず 16 バイト・ポインターのアドレスを渡すようにしてください。以下に例を挙げます。

  – この例では、CRTCMOD などの作成コマンドで STGMDL (*TERASPACE)DTAMDL(*LLP64) オプションを指定して 8 バイト・ポインターを使用するテラスペース・ストレージ・モデル・プログラムを作成しました。次に、ポインターを、配列内の文字を指すポインターに、テラスペース・ストレ

ージ・モデル・プログラムから P128 **char**** インターフェースに渡すとします。これを行なうには、明示的に 16 バイト・ポインターを宣言する必要があります。

```
#pragma datamodel(P128)
void func(char **);
#pragma datamodel(pop)

char myArray[32];
char *_ptr128 myPtr;

myPtr = myArray; /* assign address of array to 16-byte pointer */
func(&myPtr);    /* pass 16-byte pointer address to the function */
```

– 一般的に使用される、ポインター間パラメーターを使用するアプリケーション・プログラミング・インターフェース (API) の一つに iconv があります。この API は 16 バイト・ポインターのみを想定しています。次に **iconv** のヘッダー・ファイルの一部を示します。

```
...
#pragma datamodel(P128)
...
size_t inconv(iconv_t cd,
              char     **inbuf,
              size_t   *inbytesleft,
              char     **outbuf,
              size_t   *outbytesleft);
...
#pragma datamodel(pop)
...
```

次のコードは DTAMDL(*LLP64) オプションを指定してコンパイルされたプログラムから **iconv** を呼び出しています。

```
...
iconv_t myCd;
size_t myResult;
char *_ptr128 myInBuf, myOutBuf;
size_t myInLeft, myOutLeft;
...
myResult = inconv(myCd, &myInBuf, &myInLeft, &myOutBuf, &myOutLeft);
...
```

ユーザー・スペース (QUSPTRUS) インターフェースに対する検索ポインターのヘッダー・ファイルが、実際にはポインター間パラメーターが期待されている個所に `void*` パラメーターを指定しているということにも注意を払う必要があります。第 2 オペランドには、必ず 16 バイト・ポインターのアドレスを渡すようにしてください。

- *シナリオ 13: コマンド処理、妥当性検査、およびプロンプト・オーバーライド・プログラムのパラメーターにアクセス*

  テラスペース・ストレージ・モデルを使用して作成されたコマンド処理、妥当性検査、およびプロンプト・オーバーライド・プログラムは、パラメーターを単一レベル・ストレージに受け取ります。そのようなプログラムは、コマンド行から CALL を使用して呼び出された場合は、テラスペース・ストレージにパラメーターを受け取ります。

  これらのプログラムは、まずパラメーターをテラスペースにコピーしてからでないと、8 バイト・ポインターを使用してパラメーターにアクセスすることはできません。その他のアプリケーションでテラスペース機能の利点を利用する 1 つの

方法は、オプション TERASPACE(*YES *TSIFC) および DTAMDL(*P128) を使用してコマンド処理、妥当性検査、ならびにプロンプト・オーバーライド・プログラムを作成することです。これらのオプションを使用することにより、プログラムがテラスペース利用可能なプログラムとなり、malloc の実行時にテラスペース・ストレージを獲得し、16 バイト・ポインターを使用するようになります。8 バイト・ポインターでアクセスされるすべてのパラメーターは、malloc で割り振られたテラスペース・ストレージにまずコピーすることができます。

コマンド処理プログラムに次のようなコードを組み込んで、テラスペース・ストレージ・モデルと 8 バイト・ポインターを使用するその他のアプリケーションに、コマンドのパラメーターを渡すことができます。

```
#include <stdlib.h>
#include <string.h>

#define ParmLen 32

int main(int argc, char *argv[])
{
   char * myTsPtr;
   void AppFunc(char *__ptr64); /* entry to rest of the application */
   ...
   /* module created with TERASPACE(*YES *TSIFC) */
   myTsPtr = (char *)malloc(ParmLen); /* allocate teraspace storage */
   ...
   /* copy parameter to teraspace */
   memcpy(myTsPtr, argv[1], (size_t)ParmLen);
   /* pass copied parameter along to rest of the application */
   AppFunc(myTsPtr); /* 16-byte pointer implicitly converted to 8-byte */
   ...
}
```

- *シナリオ 14: 関数の再宣言*

  IBM 提供のヘッダー・ファイルですでに宣言されている関数は、再度、宣言しないようにしてください。ローカル宣言では、ポインターの長さが正しく指定されないことがあります。そのような、よく使用されるインターフェースに、**errno** があります。これは OS/400 での関数呼び出しとしてインプリメントされています。

- *シナリオ 15: ポインターを戻すプログラムおよび関数でのデータ・モデル *LLP64 の使用*

  データ・モデル *LLP64 を使用する場合は、ポインターを戻すプログラムあるいは関数を注意深く観察してください。ポインターが単一レベル・ストレージを指している場合は、その値を 8 バイト・ポインターに正しく割り当てることができないため、それらのインターフェースのクライアントは戻り値を 16 バイト・ポインターに保存する必要があります。 QUSPTRUS は、そのような API の 1 つです。ユーザー・スペースは、単一レベル・ストレージにあります。

  ポインターを戻す関数の例としては、Java Native Interface (JNI) 関数の GetStringChars および GetByteArrayElements があります。最初の関数は、単一レベル・ストレージ内にあるユニコード文字ストリングを指すポインターを戻します。また、2 番目の関数はプリミティブな配列を指すポインター、あるいはその配列のコピーを戻します。

- *シナリオ 16: JNI 関数を使用する際の問題の回避*

  テラスペース・ストレージを使用していて、JNI 関数を呼び出す場合は、PTF MF26929 をインストールしてください。

- *シナリオ 17: ライセンス内部コード・オプション DetectConvertTo8BytePointerError による初期デバッグの実行*

  STGMDL(*TERASPACE) を指定して作成されたテラスペース・ストレージ・モデル・プログラムの初期デバッグについては、ライセンス内部コード・オプション DetectConvertTo8BytePointerError の使用を考慮してください。モジュールおよびプログラムの作成時にこのオプションを使用すると、実行時に単一レベルのストレージ・アドレスを 8 バイト・ポインターに変換しようとすると MCH0609 で通知するコードが生成されます。

- *シナリオ 18: ライセンス内部コード・オプション MinimizeTeraspaceFalseEAOs の使用*

  16 バイト・ポインターを使用して 16 MB より大きいテラスペース・ストレージにアドレッシングするプログラムに対しては、 161 ページの『第 13 章 拡張最適化技法』で説明しているライセンス内部コード・オプション MinimizeTeraspaceFalseEAOs の使用を考慮してください。 16 バイト・ポインターの代わりに 8 バイト・ポインターを使用することにより、有効アドレス・オーバーフロー (EAO) のオーバーヘッドを減らすこともできます。必要ない場合にこのオプションを使用すると、約 15% パフォーマンスが低下することが報告されていますので、注意してください。しかし、このオプションを正しく使用した場合は、最高 60% までパフォーマンスが向上します。

第 5 章 プログラム作成の概念

ILE プログラムまたはサービス・プログラムの作成プロセスによって、アプリケーションの設計や保守をより柔軟にコントロールできるようになりました。プロセスには以下の 2 つのステップがあります。

1. ソース・コードをコンパイルしてモジュールを作成する。
2. モジュールをバインディングして、ILE プログラムまたはサービス・プログラムを作成する。バインディングは、プログラムの作成 (CRTPGM)、またはサービス・プログラムの作成 (CRTSRVPGM) コマンドの実行によって行われます。

本章では、バインド・プログラム (バインダー) についての概念と、ILE プログラムまたはサービス・プログラムの作成プロセスに関連した概念について記述します。本章を読む前に 13 ページの『第 2 章 ILE の基本概念』で記述したバインディングの概念を理解しておくことが必要です。

プログラムの作成およびサービス・プログラムの作成コマンド

プログラム作成 (CRTPGM) コマンドとサービス・プログラムの作成 (CRTSRVPGM) コマンドは類似しており、同じパラメーターを数多く共有しています。この 2 つのコマンドで使用されるパラメーターを比較しておくと、それぞれのコマンドの使用方法を理解するのに役立ちます。

表 6 は、2 つのコマンドのパラメーターとそのデフォルト値を示しています。

表 6. CRTPGM および CRTSRVPGM コマンドのパラメーター

パラメーター・グループ	CRTPGM コマンド	CRTSRVPGM コマンド
識別	PGM(*CURLIB/WORK)	SRVPGM(*CURLIB/UTILITY)
	MODULE(*PGM)	MODULE(*SRVPGM)
プログラム・アクセス	ENTMOD (*FIRST)	EXPORT(*SRCFILE)
		SRCFILE(*LIBL/QSRVSRC)
		SRCMBR(*SRVPGM)
バインディング	BNDSRVPGM(*NONE)	BNDSRVPGM(*NONE)
	BNDDIR(*NONE)	BNDDIR(*NONE)
実行時	ACTGRP(*ENTMOD)	ACTGRP(*CALLER)
最適化	IPA(*NO)	IPA(*NO)
	IPACTLFILE(*NONE)	IPACTLFILE(*NONE)

表 6. CRTPGM および CRTSRVPGM コマンドのパラメーター (続き)

パラメーター・グループ	CRTPGM コマンド	CRTSRVPGM コマンド
その他	OPTION(*GEN *NODUPPROC *NODUPVAR *WARN *RSLVREF)	OPTION(*GEN *NODUPPROC *NODUPVAR *WARN *RSLVREF)
	DETAIL(*NONE)	DETAIL(*NONE)
	ALWUPD(*YES)	ALWUPD(*YES)
	ALWRINZ(*NO)	ALWRINZ(*NO)
	REPLACE(*YES)	REPLACE(*YES)
	AUT(*LIBCRTAUT)	AUT(*LIBCRTAUT)
	TEXT(*ENTMODTXT)	TEXT(*ENTMODTXT)
	TGTRLS(*CURRENT)	TGTRLS(*CURRENT)
	USRPRF(*USER)	USRPRF(*USER)
	STGMDL(*SNGLVL)	STGMDL(*SNGLVL)

両方のコマンドの識別パラメーターは、作成するオブジェクトおよびコピーするモジュールの名前を指定します。2 つのパラメーターの唯一の相違点は、オブジェクト作成時に使用されるデフォルトのモジュール名です。CRTPGM の場合、モジュールの名前として、プログラム (*PGM) パラメーターに指定した名前が使用されます。CRTSRVPGM の場合、モジュールの名前として、サービス・プログラム (*SRVPGM) パラメーターに指定した名前が使用されます。それ以外は、これらのパラメーターは外見も機能も同じです。

2 つのコマンドの最も重要な類似点は、バインド・プログラムがインポートとエクスポートとの間で記号を解決する方法です。いずれの場合も、バインド・プログラムは、モジュール (MODULE)、サービス・プログラムのバインディング (BNDSRVPGM)、およびディレクトリーのバインディング (BNDDIR) の各パラメーターからの入力を処理します。

2 つのコマンドの最も重要な相違点は、プログラム・アクセス・パラメーター (85 ページの『プログラム・アクセス』を参照) です。CRTPGM コマンドの場合、バインド・プログラムが識別すべき重要な点は、プログラム入り口プロシージャーが存在しているモジュールの名前です。プログラムを作成し、このプログラムに対して動的プログラム呼び出しを行うと、プログラム入り口プロシージャーが存在しているモジュールから処理が開始されます。CRTSRVPGM コマンドには、より多くのプログラム・アクセス情報が必要です。なぜなら、CRTSRVPGM コマンドは、他のプログラムまたはサービス・プログラム用の複数のアクセス・ポイントからなるインターフェースを提供することができるからです。

借用権限の使用 (QUSEADPAUT)

システム値 QUSEADPAUT は、プログラムを、借用権限の使用 (USEADPAUT(*YES)) 属性で作成できるユーザーを定義します。 システム値 QUSEADPAUT によって許可されたすべてのユーザーは、そのユーザーが必要な権限を持っている場合、借用権限を使用するプログラムおよびサービス・プログラムの作成や変更を行うことができます。必要な権限については、「iSeries 機密保護解説書」を参照してください。

このシステム値には、権限リストの名前を含めることができます。ユーザーの権限は、このリストに照らし合わせてチェックされます。ユーザーが、指定した権限リストに対して少なくとも *USE 権限を持っている場合は、そのユーザーは、プログラムまたはサービス・プログラムを USEADPAUT(*YES) 属性で作成、変更、または更新することができます。権限リストに対する権限を借用権限から持ってくることはできません。

このシステム値に権限リストが指定され、しかもその権限リストが欠落している場合、実行しようとしていた機能は完了しません。このことを示すメッセージが出されます。ただし、プログラムが QPRCRTPG API を使用して作成され、値 *NOADPAUT がオプション・テンプレートに指定されている場合には、権限リストが存在しなくてもそのプログラムは正常に作成されます。コマンドまたは API で複数の機能が要求され、権限リストが欠落している場合、その機能は実行されません。

表 7. QUSEADPAUT の可能な値

値	記述
権限リスト名	次のすべてに該当する場合、プログラムが USEADPAUT(*NO) で作成されたことを示す診断メッセージが出されます。 • 権限リストがシステム値 QUSEADPAUT に指定されている。 • ユーザーが、その権限リストに対する権限を持っていない。 • プログラムまたサービス・プログラムの作成時に、他のエラーが存在しない。 ユーザーが該当の権限リストに対する権限を持っている場合、プログラムまたはサービス・プログラムは、USEADPAUT(*YES) で作成されます。
***NONE**	システム値 QUSEADPAUT によって許可されているすべてのユーザーは、そのユーザーが必要な権限を持っている場合、借用権限を使用するプログラムおよびサービス・プログラムを作成または変更することができます。必要な権限については、「iSeries 機密保護解説書」を参照してください。

システム値 QUSEADPAUT についての詳細は、「機密保護解説書」を参照してください。

最適化パラメーターの使用

バインドされた ILE プログラムまたはサービス・プログラムをさらに最適化するために、最適化パラメーターを指定します。バインド時の最適化の詳細については 170 ページの『プロシージャー間分析 (IPA)』を参照してください。

記号の解決

記号の解決は、以下の 2 つの項目の照合を行うバインド・プログラムによるプロセスです。

- コピーによってバインドされるモジュールのセットからのインポート要求
- 指定のモジュールおよびサービス・プログラムによって提供されるエクスポートのセット

記号の解決時に使用されるエクスポートのセットは、順序付けられた (順序番号が付けられた) リストと考えることができます。エクスポートの順序は、以下によって決まります。

- CRTPGM または CRTSRVPGM コマンドの MODULE、BNDSRVPGM、および BNDDIR パラメーターに指定されたオブジェクトの順序
- 指定のモジュールの言語実行時ルーチンからのエクスポート

解決および未解決のインポート

インポートおよびエクスポートはそれぞれプロシージャーまたはデータ・タイプと名前から構成されます。**未解決インポート**は、タイプと名前がエクスポートのタイプと名前に一致しないインポートです。**解決されたインポート**は、タイプと名前がエクスポートのタイプと名前に正確に一致するインポートです。

コピーによってバインドされたモジュールからのインポートだけが、未解決インポート・リストに入ります。記号の解決時に、次の未解決インポートが使用されて、エクスポートの順序付けられたリストで一致するものが検索されます。順序付けられたエクスポートのセットを検査した後に未解決インポートがまだ存在する場合は、プログラム・オブジェクトまたはサービス・プログラムは、通常、作成されません。ただし、オプション・パラメーターに *UNRSLVREF が指定されている場合には、未解決インポートのあるプログラム・オブジェクトまたはサービス・プログラムが作成されます。このようなプログラム・オブジェクトまたはサービス・プログラムが実行時に未解決インポートを使用しようとすると、以下のことが生じます。

- プログラム・オブジェクトまたはサービス・プログラムが、バージョン 2 リリース 3 のシステムで作成、または更新された場合には、エラー・メッセージ MCH3203 が出されます。 このメッセージは、「機械語命令での機能エラー」が生じたことを示しています。
- プログラム・オブジェクトまたはサービス・プログラムが、バージョン 3 リリース 1 以降のシステムで作成または更新された場合には、エラー・メッセージ MCH4439 が出されます。このメッセージは、「解決されていないインポートを使用しようとした」ことを示しています。

コピーによるバインディング

MODULE パラメーターで指定されたモジュールは、常にコピーによってバインドされます。BNDDIR パラメーターで指定されたバインディング・ディレクトリーで名前が付けられたモジュールは、必要に応じて、コピーによってバインドされます。バインディング・ディレクトリーで名前が付けられたモジュールは、次のいずれかの場合に必要です。

- そのモジュールが、未解決インポートに対するエクスポートを提供する場合。
- そのモジュールが、サービス・プログラムの作成に使用されているバインド・プログラム言語ソース・ファイルの現行エクスポート・ブロックで名前付けされたエクスポートを提供する場合。

バインド・プログラム言語で検出されたエクスポートがモジュール・オブジェクトからのエクスポートの場合には、そのモジュールは、コマンド行で明示的に指定されていたか、バインディング・ディレクトリーからのモジュールであるかに関係なく常にコピーによりバインドされます。たとえば、

```
モジュール M1:  インポート P2
モジュール M2:  エクスポート P2
モジュール M3:  エクスポート P3
バインド・プログラム言語 S1: STRPGMEXP PGMLVL(*CURRENT)
                              EXPORT P3
                              ENDPGMEXP
バインディング・ディレクトリー BNDDIR1:  M2
                                        M3
CRTSRVPGM SRVPGM(MYLIB/SRV1) MODULE(MYLIB/M1) SRCFILE(MYLIB/S1)
          SRCMBR(S1) BNDDIR(MYLIB/BNDDIR1)
```

サービス・プログラム SRV1 は、M1、M2、および M3 の 3 つのモジュールを持ちます。M3 は、現行エクスポート・ブロックにあるので、コピーされます。

参照によるバインディング

BNDSRVPGM パラメーターに指定されたサービス・プログラムは、参照によってバインドされます。バインディング・ディレクトリーに指定されたサービス・プログラムが、未解決インポートに対応するエクスポートを提供する場合、このサービス・プログラムは参照によってバインドされます。このようにしてサービス・プログラムがバインドされると、新しいインポートは追加されません。

注: プログラムにバインドされるものをよりうまく制御するためには、一般的なサービス・プログラム名または特定のライブラリーを指定します。値 *LIBL をユーザー制御の環境において指定できるのは、プログラムにバインドされるものが正確にわかっている場合のみです。OPTION(*DUPPROC *DUPVAR) とともに BNDSRVPGM(*LIBL/*ALL) を指定しないでください。*ALL とともに *LIBL を指定すると、プログラム実行時に予期しない結果が発生する可能性があります。

多数のモジュールのバインディング

CRTPGM および CRTSRVPGM コマンドのモジュール (MODULE) パラメーターの場合、指定できるモジュール数には限度があります。バインドしたいモジュールの数がこの限度を超える場合、以下のいずれかの方式を使用することができます。

1. バインディング・ディレクトリーを使用して、他のモジュールが必要とするエクスポートを提供する多数のモジュールをバインドする。
2. CRTPGM および CRTSRVPGM コマンドの MODULE パラメーターで総称モジュール名が指定できる命名規則を使用する。たとえば、`CRTPGM PGM(mylib/payroll) MODULE(mylib/pay*)` のような方式です。`pay` で始まる名前をもつモジュールは、すべて無条件にプログラム `mylib/payroll` に組み込まれます。したがって、CRTPGM または CRTSRVPGM コマンドに指定された総称名が不必要なモジュールをバインドしないように、慎重に命名規則を選んでください。
3. モジュールを別個のライブラリーにグループ化して、値 *ALL が MODULE パラメーター上の特定のライブラリー名と一緒に使用できるようにする。たとえば、`CRTPGM PGM(mylib/payroll) MODULE(payroll/*ALL)` のような方式です。ライブラリー `payroll` 内のモジュールは、すべて無条件にプログラム `mylib/payroll` に組み込まれます。
4. 方式 2 および 3 で記述した総称名および特定のライブラリーを組み合わせて使用する。

5. サービス・プログラムの場合、バインディング・ソース言語を使用する。バインディング・ソース言語において指定されたエクスポートは、エクスポートを満たす場合、モジュールをバインドします。RTVBNDSRC コマンドは、バインディング・ソース言語の作成には役立ちます。RTVBNDSRC コマンドの MODULE パラメーターは MODULE パラメーターに明示的に指定できるモジュール数を制限しますが、総称モジュール名および特定のライブラリー名をもつ値 *ALL を使用できます。同一のソース・ファイルを指定する出力に対して RTVBNDSRC コマンドを複数回使用できます。しかし、この場合、バインディング・ソース言語を編集しなければならない場合があります。

エクスポートの順序の重要性

コマンドを多少変更するだけで、別の有効なプログラムを作成することができます。MODULE、BNDSRVPGM、および BNDDIR パラメーターでオブジェクトを指定する順序は、通常、以下のいずれにも該当する場合にのみ重要です。

- 複数のモジュールまたはサービス・プログラムが重複する記号名をエクスポートしている。
- 別のモジュールがその記号名のインポートを必要としている。

重複する記号は大部分のアプリケーションにはなく、プログラマーがオブジェクトを指定する順序について考慮する必要は、ほとんどありません。エクスポートされ、インポートもされる重複する記号があるアプリケーションの場合には、CRTPGM コマンドまたは CRTSRVPGM コマンドにオブジェクトをリストする順序を考慮してください。

以下の例は、記号の解決が行われる方法を示しています。81 ページの図 30 のモジュール、サービス・プログラム、およびバインディング・ディレクトリーは、 82 ページの図 31 および 84 ページの図 32 の CRTPGM 要求に使用されます。例に示したエクスポートとインポートはすべてプロシージャーであると想定してください。

例は、プログラム作成プロセスのバインディング・ディレクトリーの役割も示しています。ライブラリー MYLIB は、CRTPGM コマンドと CRTSRVPGM コマンドのライブラリー・リストにあると想定します。以下のコマンドは、ライブラリー MYLIB 内にバインディング・ディレクトリー L を作成します。

```
CRTBNDDIR BNDDIR(MYLIB/L)
```

以下のコマンドは、モジュール M1 と M2 の名前、およびサービス・プログラム S と T の名前をバインディング・ディレクトリー L に追加します。

```
ADDBNDDIRE BNDDIR(MYLIB/L) OBJ((M1 *MODULE) (M2 *MODULE) (S) (T))
```

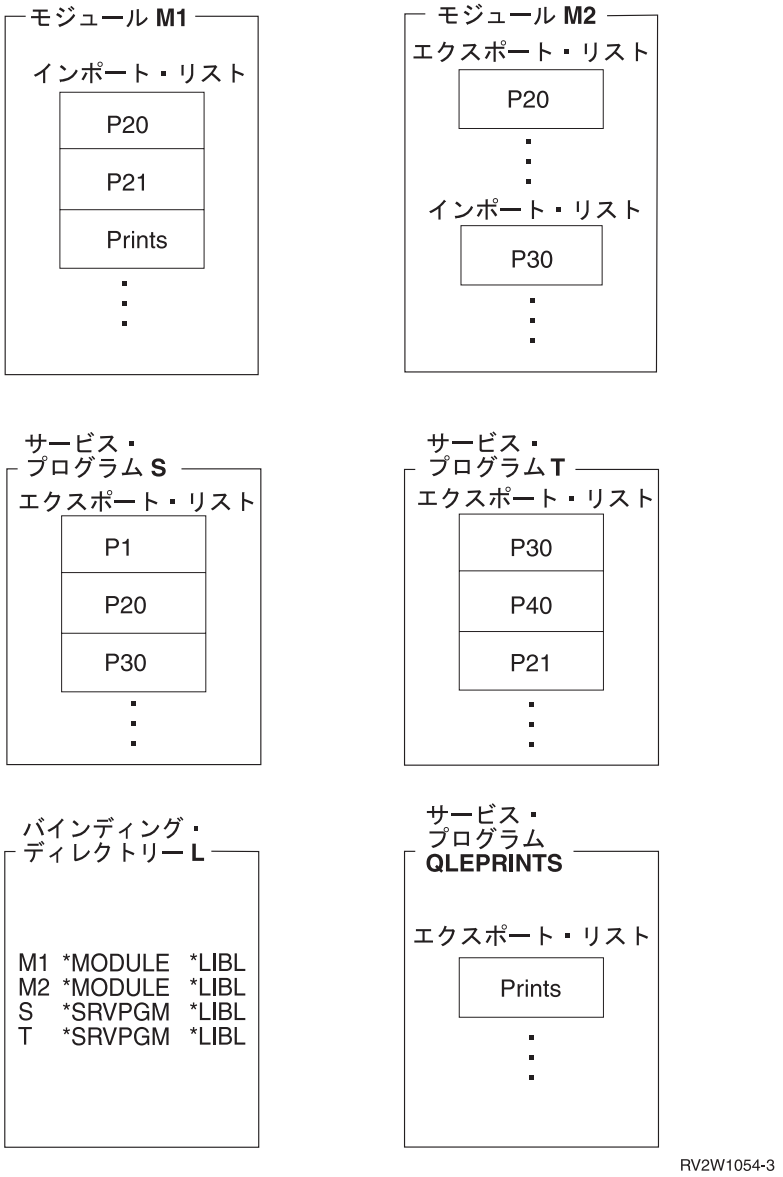

図 30. モジュール、サービス・プログラム、およびバインディング・ディレクトリー

プログラムの作成例 1

82 ページの図 31 のプログラム A の作成に、以下のコマンドを使用したと想定します。

```
CRTPGM  PGM(TEST/A)
        MODULE(*LIBL/M1)
        BNDSRVPGM(*LIBL/S)
        BNDDIR(*LIBL/L)
        OPTION(*DUPPROC)
```

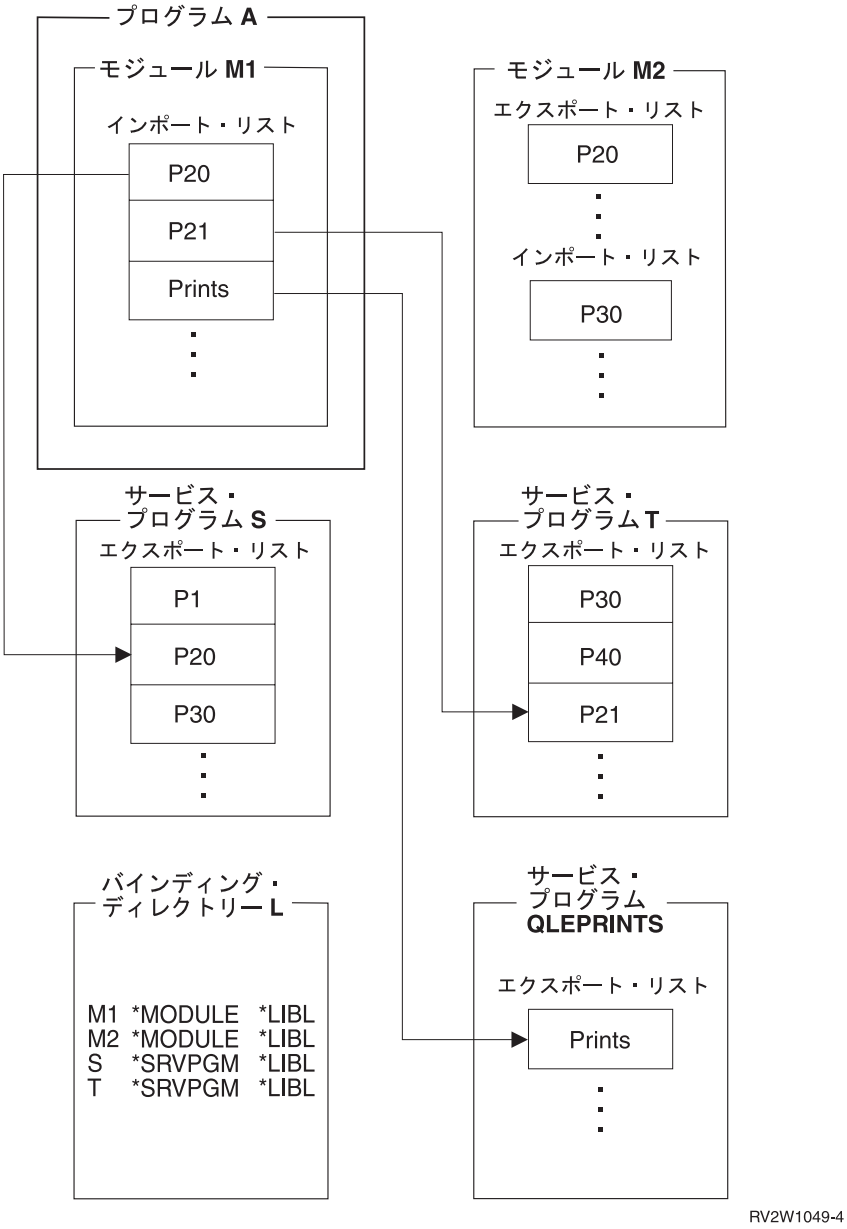

図 31. 記号の解決とプログラムの作成: 例 1

プログラム A を作成するために、バインド・プログラムは、CRTPGM コマンドのパラメーターに指定されたオブジェクトを指定された順序で処理します。

1. 最初のパラメーター (PGM) に指定された値は A で、これは作成するプログラムの名前です。
2. 2 番目のパラメーター (MODULE) に指定された値は M1 です。バインド・プログラムはここから処理を開始します。モジュール M1 には解決する必要がある 3 つのインポート、つまり P20、P21、および Prints が含まれています。
3. 3 番目のパラメーター (BNDSRVPGM) に指定された値は S です。バインド・プログラムは、未解決インポート要求を解決するためのプロシージャーを探して、サービス・プログラム S のエクスポート・リストを走査します。エクスポート・リストにはプロシージャー P20 があるので、このインポート要求が解決されます。

4. 4 番目のパラメーター (BNDDIR) に指定された値は L です。次にバインド・プログラムは、バインディング・ディレクトリー L を走査します。
   a. バインディング・ディレクトリーに指定されている最初のオブジェクトはモジュール M1 です。モジュール M1 は、モジュール・パラメーターに指定されているので現在知られていますが、エクスポートは提供していません。
   b. バインディング・ディレクトリーに指定されている 2 番目のオブジェクトは、モジュール M2 です。モジュール M2 はエクスポートを提供していますが、いずれのエクスポートも現在未解決のインポート要求 (P21 と Prints) に一致しません。
   c. バインディング・ディレクトリーに指定された 3 番目のオブジェクトは、サービス・プログラム S です。サービス・プログラム S は 3 (82 ページ) のステップですでに処理されており、追加のエクスポートは提供しません。
   d. バインディング・ディレクトリーで指定された 4 番目のオブジェクトは、サービス・プログラム T です。バインド・プログラムはサービス・プログラム T のエクスポート・リストを走査します。プロシージャー P21 が検出され、そのインポート要求が解決されます。
5. 解決する必要がある最後のインポート (Prints) は、どのパラメーターにも指定されていません。しかし、バインド・プログラムは Prints プロシージャーを、サービス・プログラム QLEPRINTS のエクスポート・リストの中から検出します。このサービス・プログラムは、この例のコンパイラーによって提供される共通実行時ルーチンです。モジュールのコンパイル時に、コンパイラーは、自分自身の実行時サービス・プログラムおよび ILE 実行時サービス・プログラムが入っているバインディング・ディレクトリーをデフォルトとして指定します。 これによって、バインド・プログラムは、コンパイラーによって提供された実行時サービス・プログラム内に残っている未解決参照を探す必要があることを知ります。バインド・プログラムが実行時サービス・プログラムを探し、解決できない参照がある場合には、バインドは通常失敗します。ただし、作成コマンドに OPTION (*UNRSLVREF) を指定すると、プログラムが作成されます。

プログラムの作成例 2

84 ページの図 32 は同様の CRTPGM 要求の結果を示しています。ただし、BNDSRVPGM パラメーターのサービス・プログラムが除去されている点が異なります。

```
CRTPGM  PGM(TEST/A)
        MODULE(*LIBL/M1)
        BNDDIR(*LIBL/L)
        OPTION(*DUPPROC)
```

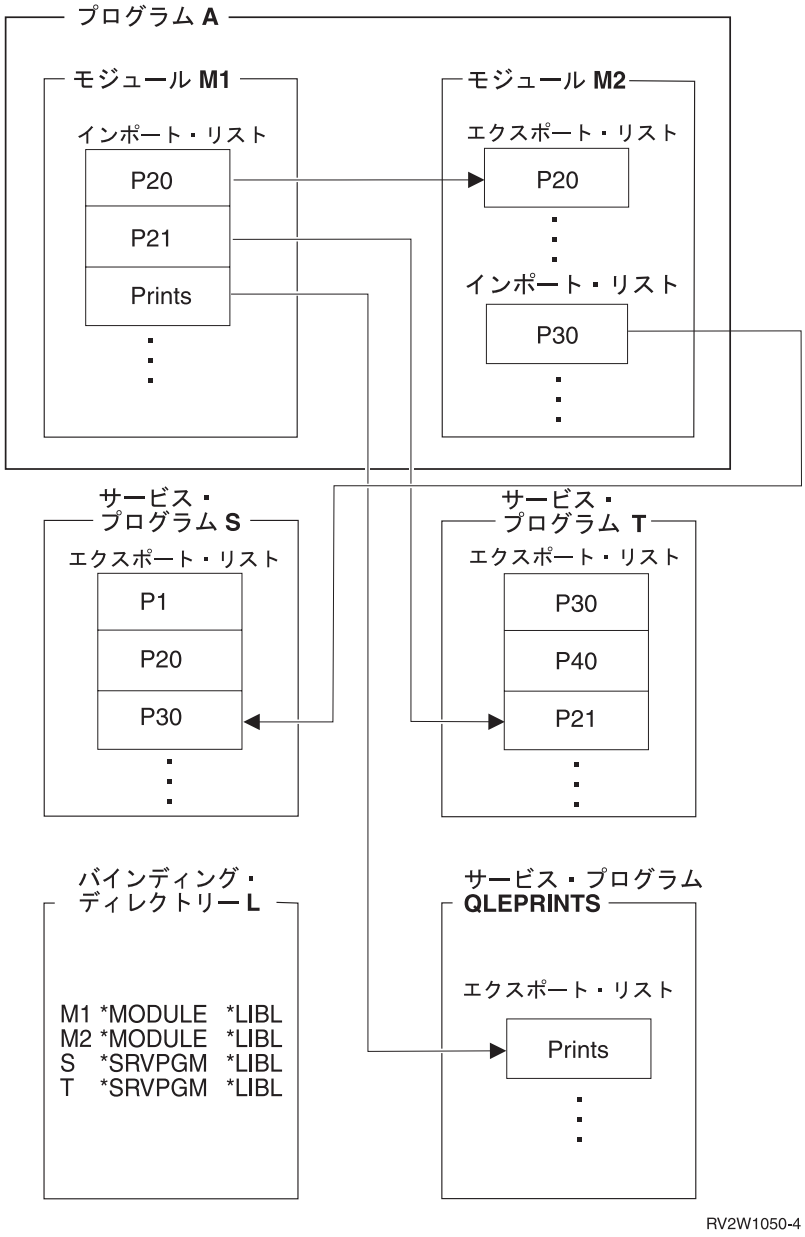

図 32. 記号の解決とプログラムの作成: 例 2

処理すべきオブジェクトの順序を変更すると、エクスポートの順序も変更されます。これによって、例 1 で作成されたプログラムとは異なるプログラムが作成されます。CRTPGM コマンドの BNDSRVPGM パラメーターにサービス・プログラム S が指定されていないので、バインディング・ディレクトリーが処理されます。モジュール M2 はプロシージャー P20 をエクスポートし、バインディング・ディレクトリーの中でサービス・プログラム S の前に指定されます。したがって、モジュール M2 は、この例の結果としてプログラム・オブジェクトにコピーされます。82 ページの図 31 と 図 32 を比較すると、以下の相違点があります。

- 例 1 のプログラム A はモジュール M1 だけを含んでおり、サービス・プログラム S、T、および QLEPRINTS からのプロシージャーを使用しています。
- 例 2 のプログラム A では、M1 と M2 の 2 つのモジュールが、サービス・プログラム T および QLEPRINTS を使用しています。

例 2 では、プログラムは以下のように作成されます。

1. 最初のパラメーター (PGM) は、作成するプログラムの名前を指定しています。
2. 2 番目のパラメーター (MODULE) に指定された値は M1 なので、バインド・プログラムは、またここから開始します。モジュール M1 には解決する必要がある同じ 3 つのインポート、つまり P20、P21、および Prints が含まれます。
3. この例では、指定された 3 番目のパラメーターは BNDSRVPGM ではなく、BNDDIR です。したがって、バインド・プログラムは指定されたバインディング・ディレクトリー (L) を最初に走査します。
   a. バインディング・ディレクトリーに指定されている最初の項目はモジュール M1 です。このライブラリーからのモジュール M1 は、モジュール・パラメーターによってすでに処理されています。
   b. バインディング・ディレクトリーに指定されている 2 番目の項目はモジュール M2 です。そこで、バインド・プログラムはモジュール M2 のエクスポート・リストを走査します。エクスポート・リストには P20 があるので、このインポート要求が解決されます。モジュール M2 はコピーによってバインドされ、モジュール M2 のインポートは、処理のため未解決インポート要求のリストに追加する必要があります。これによって、未解決インポート要求は P21、Prints、および P30 になります。
   c. 処理は続行され、バインディング・ディレクトリーで指定された次のオブジェクト、すなわち、サービス・プログラム S に移ります。この場合、サービス・プログラム S は、P21 および Prints の現在未解決のインポート要求に対し P30 エクスポートを提供します。処理は、バインディング・ディレクトリーにリストされている次のオブジェクト、サービス・プログラム T に進みます。
   d. サービス・プログラム T は、未解決のインポートに対しエクスポート P21 を提供します。
4. 例 1 と同様に、インポート要求 Prints は指定されていません。ただし、このプロシージャーは、モジュール M1 を作成した言語によって提供される実行時ルーチンにあります。

記号の解決は、エクスポートの強さによっても影響されます。ストロング・エクスポートとウイーク・エクスポートについては 88 ページの『インポートおよびエクスポートの概念』のエクスポートの項を参照してください。

プログラム・アクセス

ILE プログラム・オブジェクトまたはサービス・プログラム・オブジェクトを作成する場合、他のプログラムがそのプログラムにアクセスする方法を指定する必要があります。CRTPGM コマンドでは、入り口モジュール (ENTMOD) パラメーターによって指定します。 CRTSRVPGM コマンドでは、エクスポート (EXPORT) パラメーターによって指定します (75 ページの表 6 を参照)。

CRTPGM コマンドのプログラム入り口プロシージャー・モジュール・パラメーター

プログラム入り口プロシージャー・モジュール (ENTMOD) パラメーターは、以下のプロシージャーが存在するモジュールの名前をバインド・プログラムに伝えます。

プログラム入り口プロシージャー (PEP)
ユーザー入り口プロシージャー (UEP)

この情報は、作成されたプログラムの動的呼び出しが行われた場合に、制御が渡される PEP を含むモジュールを示します。

ENTMOD パラメーターのデフォルト値は *FIRST です。この値は、PEP を含むモジュール・パラメーターで指定されたモジュールのリストでバインド・プログラムが検出した最初のモジュールを入り口モジュールとして使用することを指定します。

以下の条件に該当する場合、
ENTMOD パラメーターに *FIRST が指定されている。
2 番目のモジュールが PEP を含んでいる。

バインド・プログラムは、この 2 番目のモジュールをプログラム・オブジェクトにコピーし、バインディング・プロセスを続行します。バインド・プログラムは、この他の PEP を無視します。

ENTMOD パラメーターに *ONLY の指定がある場合、プログラム中の 1 つのモジュールだけが PEP を含むことができます。*ONLY が指定されているときに、PEP を含む 2 番目のモジュールが見つかった場合、プログラムは作成されません。

明示的に制御するには、PEP を含んでいるモジュールの名前を指定します。他のすべての PEP は無視されます。明示的に指定したモジュールに PEP が含まれていない場合には、CRTPGM 要求は失敗します。

モジュールにプログラム入り口プロシージャーがあるかどうかを調べるには、モジュールの表示 (DSPMOD) コマンドを使用します。この情報は、「モジュール情報表示」画面の*プログラム入り口プロシージャー名* フィールドに表示されます。 このフィールドに *NONE が表示されている場合、このモジュールに PEP はありません。このフィールドに名前が表示されている場合、このモジュールに PEP があります。

CRTSRVPGM コマンドのエクスポート・パラメーター

エクスポート (EXPORT)、ソース・ファイルのエクスポート (SRCFILE)、およびソース・メンバーのエクスポート (SRCMBR) の各パラメーターは、作成されるサービス・プログラムへの共通インターフェースを示します。これらのパラメーターは、他の ILE プログラムまたはサービス・プログラムによりサービス・プログラムが使用できるエクスポート (プロシージャーおよびデータ) を指定します。

エクスポート・パラメーターのデフォルト値は *SRCFILE です。この値は、サービス・プログラムのエクスポートに関する情報の参照に SRCFILE パラメーターを使用するようにバインド・プログラムに指示します。この追加情報は、バインド・プログラム言語ソースが入っているソース・ファイルです (90 ページの『バインド・

プログラム言語』を参照)。バインド・プログラムはバインド・プログラム言語ソースを見つけ、そこに指定されているエクスポートすべき名前から 1 つ以上のシグニチャー (記号) を生成します。バインド・プログラム言語により、バインド・プログラムが生成するシグニチャーの代わりにユーザーの選択でシグニチャーを指定することができます。

バインダー・ソース検索 (RTVBNDSRC) コマンドを使用すると、バインド・プログラム言語ソースを含むソース・ファイルを作成することができます。このソースは、既存のサービス・プログラム、またはモジュールのセットのいずれかを基にすることができます。サービス・プログラムを基にすると、そのサービス・プログラムの再作成または更新に適したソースが作成されます。モジュールのセットを基にすると、そのモジュールからエクスポートするのに適したすべての記号を含むソースが作成されます。いずれの場合にも、エクスポートしたい記号だけを含むようにこのファイルを編集したうえで、CRTSRVPGM または UPDSRVPGM コマンドの SRCFILE パラメーターを使用してこのファイルを指定することができます。

エクスポート・パラメーターとして指定できる他の値は *ALL です。EXPORT(*ALL) を指定すると、コピーされたモジュールからエクスポートされるすべての記号が、サービス・プログラムからエクスポートされます。生成されるシグニチャーは、以下によって決まります。

- エクスポートされる記号の数
- エクスポートされる記号のアルファベット順

EXPORT(*ALL) を指定した場合、サービス・プログラムからのエクスポートを定義するためにバインド・プログラム言語は必要ありません。この値を使用するのが最も簡単な方法です。なぜなら、バインド・プログラム言語ソースを生成する必要がないからです。しかし、EXPORT(*ALL) を指定して作成したサービス・プログラムは、エクスポートが他のプログラムによって使用される場合、更新または訂正が困難になる可能性があります。サービス・プログラムを変更すると、エクスポートの順序または個数も変更される可能性があります。したがって、サービス・プログラムのシグニチャーも変更される可能性があります。シグニチャーが変更されると、変更されたサービス・プログラムを使用するすべてのプログラムまたはサービス・プログラムを再作成しなければなりません。

EXPORT(*ALL) は、サービス・プログラムで使用されたモジュールからエクスポートされたすべての記号がサービス・プログラムからエクスポートされることを示します。 ILE C は、エクスポートをグローバルまたは静的として定義することができます。ILE C でグローバルとして宣言された外部変数だけが、EXPORT(*ALL) で使用可能です。 ILE RPG では、以下を EXPORT(*ALL) で使用することができます。

- RPG メイン・プロシージャー名
- エクスポートされるサブプロシージャーの名前
- キーワード EXPORT で定義された変数

ILE COBOL では、以下の言語要素がモジュール・エクスポートです。

- 字句単位の最外部の COBOL プログラム (*PGM オブジェクトと混同しないでください) の中の PROGRAM-ID 段落にある名前。これはストロング・プロシージャー・エクスポートにマップします。

- プログラムが INITIAL 属性を持たない場合の上記の PROGRAM-ID 段落にある名前から引き出された COBOL コンパイラー生成名。これはストロング・プロシージャー・エクスポートにマップします。ストロング・エクスポートとウイーク・エクスポートについては 『インポートおよびエクスポートの概念』のエクスポートの項を参照してください。
- EXTERNAL として宣言されたデータ項目またはファイル。これはウイーク・エクスポートにマップします。

ソース・ファイルおよびソース・メンバーのパラメーターとともに使用されるエクスポート・パラメーター

エクスポート・パラメーターのデフォルト値は *SRCFILE です。エクスポート・パラメーターに *SRCFILE を指定した場合、バインド・プログラムは、バインド・プログラム言語ソースを見つけるために、SRCFILE および SRCMBR パラメーターも使用しなければなりません。

以下のコマンドの例は、バインド・プログラム言語ソースを見つけるためにデフォルト値を使用して、UTILITY という名前のサービス・プログラムをバインドしています。

```
CRTSRVPGM SRVPGM(*CURLIB/UTILITY)
          MODULE(*SRVPGM)
          EXPORT(*SRCFILE)
          SRCFILE(*LIBL/QSRVSRC)
          SRCMBR(*SRVPGM)
```

このコマンドによってサービス・プログラムを作成するには、UTILITY という名前のメンバーがソース・ファイル QSRVSRC になければなりません。このメンバーには、バインド・プログラムが 1 つのシグニチャーおよび 1 組のエクスポート ID に変換するバインド・プログラム言語ソースが入っていなければなりません。デフォルトでは、サービス・プログラムの名前 UTILITY と同じ名前のメンバーからバインド・プログラム言語ソースを入手します。これらのパラメーターに指定した値をもつファイル、メンバー、またはバインド・プログラム言語ソースが見つからない場合、サービス・プログラムは作成されません。

SRCFILE パラメーターのファイルの最大レコード幅

V3R7 以降のリリースでは、CRTSRVPGM または UPDSRVPGM コマンドのソース・ファイル (SRCFILE) パラメーターに指定するファイルの最大レコード幅は、240 文字です。ファイルがこの最大レコード幅を超える場合、メッセージ CPF5D07 が出されます。 V3R2 では、最大レコード幅は 80 文字です。
V3R6、V3R1、V2R3 では、最大レコード幅に限界はありません。

インポートおよびエクスポートの概念

ILE 言語は、以下のタイプのエクスポートとインポートをサポートします。

- ウイーク・データ・エクスポート
- ウイーク・データ・インポート
- ストロング・データ・エクスポート
- ストロング・データ・インポート
- ストロング・プロシージャー・エクスポート

- ウイーク・プロシージャー・エクスポート
- プロシージャー・インポート

ILE モジュール・オブジェクトは、プロシージャーまたはデータ項目を他のモジュールにエクスポートすることができます。また、ILE モジュール・オブジェクトは、他のモジュールからのプロシージャーまたはデータ項目をインポート (参照) することができます。サービス・プログラムを作成する CRTSRVPGM コマンドにモジュール・オブジェクトを指定すると、そのエクスポートは必要に応じてそのサービス・プログラムからエクスポートします。 (86 ページの『CRTSRVPGM コマンドのエクスポート・パラメーター』を参照してください。) エクスポートの強さ (ストロングまたはウィーク) は、プログラム言語に依存します。強さは、エクスポートのサイズなどの特性を設定するためにそのデータ項目について十分に知ることができる時点を判別します。ストロング・エクスポートの特性はバインド時に設定されます。エクスポートの強さは、記号の解決に影響します。

- 1 つ以上のウイーク・エクスポートがストロング・エクスポートと同じ名前を持っている場合には、バインド・プログラムはストロング・エクスポートの特性を使用します。
- ウイーク・エクスポートがストロング・エクスポートと同じ名前を持っていない場合には、その特性は活動化の時点まで設定されません。活動化の時点で、複数の同じ名前のウイーク・エクスポートが存在する場合には、最大のウイーク・エクスポートが使用されます。既に活動化された同じ名前のウイーク・エクスポートにその特性が設定されていない限り、最大のウイーク・エクスポートが使用されます。
- バインド時に、バインディング・ディレクトリーが使用され、ウィーク・インポートに一致するウィーク・エクスポートが見つかると、それらはバインドされます。ただし、バインディング・ディレクトリーは、解決すべき未解決のインポートがある場合にのみ検索されます。すべてのインポートが解決されると、バインディング・ディレクトリー項目の探索は停止します。重複するウィーク・エクスポートは、重複する変数またはプロシージャーとしてフラグが付けられることはありません。バインディング・ディレクトリーにおける項目の順序は、きわめて重要です。

ウイーク・エクスポートは、プログラム・オブジェクトまたはサービス・プログラムの外部にエクスポートすることが可能で、活動化の時点で解決されます。これは、バインド時にのみ、サービス・プログラムの外部にのみエクスポートできるストロング・エクスポートとは対照的です。

ただし、ストロング・エクスポートは、プログラム・オブジェクトの外部にエクスポートすることはできません。ストロング・プロシージャー・エクスポートは、バインド実行時に次のいずれかを満たす場合、サービス・プログラムの外部にエクスポートすることができます。

- そのサービス・プログラムを参照によってバインドするプログラム中のインポート
- そのプログラムへの参照によってバインドされる他のサービス・プログラム中のインポート

サービス・プログラムは、ソース言語のバインディングにより共通インターフェースを定義します。

ウィーク・プロシージャー・エクスポートを、ソース言語のバインディングにより、サービス・プログラムの共通インターフェースの一部にすることができます。ただし、ソース言語のバインディングによりサービス・プログラムからウィーク・プロシージャー・エクスポートをエクスポートすると、そのエクスポートはもはやウィークとしてマークされません。ストロング・プロシージャー・エクスポートとして扱われます。

ウィーク・データは、活動化グループにのみエクスポートできます。バインド・プログラム・ソース言語の使用により、サービス・プログラムからエクスポートされる共通インターフェースの一部にすることはできません。バインド・プログラム・ソース言語にウィーク・データを指定すると、そのバインドは失敗します。

表 8 は、いくつかの ILE 言語がサポートするインポートおよびエクスポートのタイプを要約しています。

表 8. ILE 言語がサポートするインポートおよびエクスポート

ILE 言語	ウイーク・データ・エクスポート	ウイーク・データ・インポート	ストロング・データ・エクスポート	ストロング・データ・インポート	ストロング・プロシージャー・エクスポート	ウイーク・プロシージャー・エクスポート	プロシージャー・インポート
RPG IV	不可	不可	可	可	可	不可	可
COBOL[2]	可[3]	可[3]	不可	不可	可[1]	不可	可
CL	不可	不可	不可	不可	可[1]	不可	可
C	不可	不可	可	可	可	不可	可
C++	不可	不可	可	可	可	可	可

注:

1. COBOL および CL がモジュールからのエクスポートを許可するのは、1 つのプロシージャーだけです。
2. COBOL は、ウイーク・データ・モデルを使用します。外部として宣言されるデータ項目は、そのモジュールに対して、ウイーク・エクスポートおよびウイーク・インポートの両方になります。
3. COBOL では、NOMONOPRC オプションが必要です。このオプションがないと、小文字は自動的に大文字に変換されます。

特定の言語に関して、どの宣言がインポートおよびエクスポートになるかについては、以下のいずれかの資料を参照してください。

- 「WebSphere Development Studio: ILE RPG プログラマーの手引き」
- 「WebSphere Development Studio: ILE COBOL プログラマーの手引き」
- 「WebSphere Development Studio ILE C/C++ Programmer's Guide」

バインド・プログラム言語

バインド・プログラム言語は、サービス・プログラムに対するエクスポートを定義する実行不能なコマンドの小さなセットです。バインド・プログラム言語によって、原始ステートメント入力ユーティリティー (SEU) の構文検査機能は、BND ソース・タイプ指定時に入力のプロンプトと妥当性検査を行うことができます。

注: ワイルドカードを含むバインド・プログラム・ソース・ファイルに対し、タイプ BND の SEU 構文検査を使用することはできません。また、254 文字を超える名前を含むバインド・プログラム・ソース・ファイルに対して、SEU の構文検査を使用することはできません。

バインド・プログラム言語は、以下のコマンドのリストから構成されます。

1. プログラム・エクスポート・リストの開始 (STRPGMEXP) コマンド。サービス・プログラムからのエクスポートのリストの始まりを識別します。
2. プログラム記号のエクスポート (EXPORT) コマンド。各コマンドは、サービス・プログラムからエクスポートできる記号名を示します。
3. プログラム・エクスポート・リストの終了 (ENDPGMEXP) コマンド。サービス・プログラムからのエクスポートのリストの終わりを識別します。

図 33 は、ソース・ファイルのバインド・プログラム言語の例です。

```
STRPGMEXP PGMLVL(*CURRENT) LVLCHK(*YES)
   .
   .
EXPORT SYMBOL(p1)
EXPORT SYMBOL('p2')
EXPORT SYMBOL('P3')
   .
   .
ENDPGMEXP
   .
   .
   .

STRPGMEXP  PGMLVL(*PRV)
   .
   .
EXPORT SYMBOL(p1)
EXPORT SYMBOL('p2')
   .
   .
ENDPGMEXP
```

図 33. ソース・ファイルのバインド・プログラム言語の例

バインダー・ソース検索 (RTVBNDSRC) コマンドの使用は、1 つ以上のモジュールからのエクスポートに基づくバインド・プログラム言語ソースを生成するのに役立ちます。

シグニチャー (インターフェース識別値)

STRPGMEXP PGMLVL (*CURRENT) と ENDPGMEXP のペアの間で指定された各記号は、サービス・プログラムの共通インターフェースを定義します。共通インターフェースは、**シグニチャー (インターフェース識別値)** で示されます。シグニチャーは、サービス・プログラムによりサポートされるインターフェースを識別する値です。

注: このトピックで記述するシグニチャーとディジタル・オブジェクト・シグニチャー *(署名)* を混同しないでください。OS/400 オブジェクトのディジタル・シグニチャーによって、ソフトウェアとデータの整合性が保たれます。このシグニチャーは、データの悪用、ウィルスによる感染、またはオブジェクトの無許

可の変更を阻止する手段としても機能します。シグニチャーは、さらにデータの発信元の明確な識別も提供します。ディジタル・オブジェクト・シグニチャーの詳細については、iSeries Information Center の **セキュリティー**・カテゴリーを参照してください。

明示的シグニチャーを指定しない場合、バインド・プログラムは、エクスポートされるプロシージャーとデータ項目の名前のリストおよびそれらの指定順序からシグニチャーを生成します。したがって、シグニチャーを使用すれば、サービス・プログラムの共通インターフェースの妥当性を簡便に検査することができます。シグニチャーは、サービス・プログラム内の特定のプロシージャーに対するインターフェースの妥当性は検査しません。

注: サービス・プログラムに対して互換性のない変更を行うのを避けるために、バインド・プログラム言語のソースの既存のプロシージャーやデータ項目の名前の除去や再配置を行ってはなりません。追加のエクスポート・ブロックには、既存のエクスポート・ブロックと同じ順序で同じ記号が含まれていなければなりません。追加の記号は、リストの終わりにのみ追加しなければなりません。

既存のプログラムやサービス・プログラムと互換性のある方法でサービス・プログラムのエクスポートを除去する方法はありません。そのエクスポートがそのサービス・プログラムにバインドされたプログラムやサービス・プログラムによって必要になることがあるからです。

サービス・プログラムに対し互換性のない変更が行われると、そのサービス・プログラムにバインドされたままになっている既存プログラムは、正しく機能しなくなります。サービス・プログラムに対する互換性のない変更は、そのような変更が行われた後で、そのサービス・プログラムにバインドされているすべてのプログラムやサービス・プログラムが、**CRTPGM** または **CRTSRVPGM** コマンドを用いて確実に再作成される場合にのみ行うことができます。

プログラム・エクスポート・リストの開始コマンドとプログラム・エクスポート・リストの終了コマンド

プログラム・エクスポート・リストの開始 (STRPGMEXP) コマンドは、サービス・プログラムからのエクスポートのリストの始まりを識別します。プログラム・エクスポート・リストの終了 (ENDPGMEXP) コマンドは、サービス・プログラムからのエクスポートのリストの終わりを識別します。

1 つのソース・ファイルに STRPGMEXP と ENDPGMEXP の複数のペアを指定すると、複数のシグニチャーが作成されます。STRPGMEXP と ENDPGMEXP の各組の指定順序に意味はありません。

STRPGMEXP コマンドのプログラム・レベル・パラメーター

PGMLVL(*CURRENT) を指定できるのは 1 つの STRPGMEXP コマンドだけですが、最初の STRPGMEXP コマンドである必要はありません。ソース・ファイル内の他のすべての STRPGMEXP コマンドには PGMLVL(*PRV) を指定しなければなりません。現行シグニチャーは、PGMLVL(*CURRENT) の指定がある STRPGMEXP コマンドを示します。

STRPGMEXP コマンドのレベル・チェック・パラメーター

STRPGMEXP コマンドのレベル検査 (LVLCHK) パラメーターは、バインド・プログラムがサービス・プログラムへの共通インターフェースを自動的に検査する必要があるかどうかを指定します。LVLCHK(*YES) を指定するか、デフォルト値 LVLCHK(*YES) を使用すると、バインド・プログラムはシグニチャー (インターフェース識別値) パラメーターを調べます。シグニチャー (インターフェース識別値) パラメーターは、バインド・プログラムが明示的シグニチャーを使用するかまたはゼロ以外のシグニチャー値を生成するかを決めます。バインド・プログラムがシグニチャー値を生成する場合には、システムは、値がサービス・プログラムのクライアントに知られる値と一致しているかどうかを確認します。値が一致する場合、サービス・プログラムのクライアントは、再コンパイルせずに共通インターフェースを使用することができます。

LVLCHK(*NO) を指定すると、自動シグニチャー検査は行われません。以下の条件が存在する場合に、この機能を使用することができます。

- サービス・プログラムのインターフェースに対する特定の変更によっても互換性が保たれていることがわかっている場合。
- バインド・プログラム言語ソース・ファイルの更新またはクライアントの再コンパイルを回避したい場合。

LVLCHK(*NO) の値は、共通インターフェースが前のレベルとの互換性を保っているかどうかを手操作で検査するのはユーザーの責任であることを意味するので注意して使用してください。クライアントによって呼び出されるサービス・プログラムのプロシージャーおよびクライアントによって使用される変数を制御できる場合にのみ、 LVLCHK(*NO) を指定してください。共通インターフェースを制御できない場合には、実行時エラーまたは活動化エラーが発生する可能性があります。バインド・プログラム言語の使用によって発生する可能性がある共通エラーの記述については 203 ページの『バインド・プログラム言語のエラー』を参照してください。

STRPGMEXP コマンドのシグニチャー (インターフェース識別値) パラメーター

インターフェース識別値 (SIGNATURE) パラメーターにより、サービス・プログラムに対するシグニチャーを明示的に指定することができます。明示的シグニチャーは、16 進数ストリング、または文字ストリングのいずれでもかまいません。次のいずれかの理由で、シグニチャーを明示指定したい場合があります。

- バインド・プログラムが望んでいない互換性のあるシグニチャーを生成する可能性がある。シグニチャーは、指定されたエクスポートの名前とその順序に基づいています。したがって、2 つのエクスポート・ブロックが同じ順序で同じエクスポートを持っていると、それらは同じシグニチャーを持つことになります。サービス・プログラム提供者として、2 つのインターフェースに互換性がないことを知っている場合があります (たとえば、それらのパラメーター・リストが異なる)。この場合には、バインド・プログラムに互換性のあるシグニチャーを生成させる代わりに新しいシグニチャーを明示的に指定することができます。これを行うと、ユーザーのサービス・プログラムに非互換性が生じ、いくつかの、またはすべてのクライアントの再コンパイルが必要になります。
- バインド・プログラムが望んでいない非互換のシグニチャーを生成する可能性がある。2 つのエクスポート・ブロックが異なったエクスポートまたは異なった順序を持つ場合には、それらは異なったシグニチャーを持つことになります。サー

ビス・プログラム提供者として、2 つのインターフェースに互換性のあることを知っている場合には (たとえば、関数名が変更され、その関数が同じままである場合など)、非互換のシグニチャーをバインド・プログラムに生成させる代わりに、前にバインド・プログラムによって生成されたのと同じシグニチャーを明示して指定することができます。同じシグニチャーを指定した場合には、ユーザーはサービス・プログラムの互換性を維持し、ユーザーのクライアントはそのサービス・プログラムを再バインドすることなく使用することができます。

シグニチャー・パラメーターのデフォルト値 *GEN は、バインド・プログラムに、エクスポートされた記号からシグニチャーを生成させます。

サービス・プログラムに関するシグニチャー値は、サービス・プログラムの表示 (DSPSRVPGM) コマンドを使用し、DETAIL(*SIGNATURE) を指定することにより、判別することができます。

プログラム記号のエクスポート・コマンド

プログラム記号のエクスポート (EXPORT) コマンドは、サービス・プログラムからエクスポートできる記号名を識別します。

エクスポートされる記号に小文字が含まれている場合、記号名を 91 ページの図 33 に示すようにアポストロフィで囲まなければなりません。アポストロフィが使用されない場合には、記号名はすべて英大文字に変換されます。この例では、バインド・プログラムが p1 ではなく P1 の名前のエクスポートを探索します。

また、記号名は、ワイルド・カード文字 (<<< または >>>) の使用により、エクスポートすることができます。記号名が存在し、指定されたワイルドカードと一致する場合、その記号名がエクスポートされます。次のいずれかの条件に該当する場合は、エラーが生じ、サービス・プログラムは作成されません。

- 指定されたワイルドカードと一致する記号名がない。
- 指定されたワイルドカードと一致する記号名が複数存在する。
- 指定されたワイルドカードと記号名は一致するが、エクスポートに使用できない。

ワイルドカードで指定するサブストリングは、引用符で囲む必要があります。

シグニチャーは、ワイルドカードで指定された文字によって判別されます。ワイルドカードの指定を変更すると、変更されたワイルドカードの指定が同じエクスポートと一致する場合でも、シグニチャーが変更されます。ワイルドカードの指定 ″r″>>> と ″ra″>>> は、両方とも記号 ″rate″ をエクスポートしますが、2 つの異なるシグニチャーを作成します。したがって、できるだけエクスポートする記号に類似したワイルドカードを指定することを強くお勧めします。

注: ワイルドカードを含むバインド・プログラム・ソース・ファイルに対し、タイプ BND の SEU 構文検査を使用することはできません。

ワイルドカードを指定した記号のエクスポートの例

以下の例で、エクスポート可能な記号のリストは次のように構成されていることを想定しています。

```
interest_rate
```

international
prime_rate

以下の例では、選択されたエクスポートまたはエラーが発生した理由を示しています。

EXPORT SYMBOL (″interest″>>>)
記号 ″interest_rate″ は ″interest″ で始まる唯一の記号なので、″interest_rate″ をエクスポートします。

EXPORT SYMBOL (″i″>>>″rate″>>>)
記号 ″interest_rate″ は ″i″ で始まり、その後に ″rate″ が含まれる唯一の記号なので、″interest_rate″ をエクスポートします。

EXPORT SYMBOL (<<<″i″>>>″rate″)
「ワイルドカードの指定と一致する記号が複数存在する」のエラーになります。″prime_rate″ および ″interest_rate″ は、いずれも ″i″ を含み ″rate″ で終わっています。

EXPORT SYMBOL (″inter″>>>″prime″)
「ワイルドカードの指定と一致する記号名がない」のエラーになります。″inter″ で始まり、″prime″ で終わる記号はありません。

EXPORT SYMBOL (<<<)
「ワイルドカードの指定と一致する記号が複数存在する」のエラーになります。3 つの記号すべてが一致します。したがって、無効になります。エクスポート・ステートメントは、記号を 1 つだけエクスポートすることができます。

バインド・プログラム言語の例

バインド・プログラム言語の使用例として、以下のプロシージャーを使用する簡単な金融アプリケーションを開発していると想定します。

- Rate プロシージャー

  Loan_Amount、Term_of_Payment、Payment_Amount の値が与えられた場合、それらの値に基づいて Interest_Rate を計算します。

- Amount プロシージャー

  Interest_Rate、Term_of_Payment、および Payment_Amount の値が与えられた場合、それらの値に基づいて Loan_Amount を計算します。

- Payment プロシージャー

  Interest_Rate、Term_of_Payment、および Loan_Amount の値が与えられた場合、それらの値に基づいて Payment_Amount を計算します。

- Term プロシージャー

  Interest_Rate、Loan_Amount、および Payment_Amount の値が与えられた場合、それらの値に基づいて Term_of_Payment を計算します。

このアプリケーションの出力リストのいくつかを 193 ページの『付録 A. CRTPGM、CRTSRVPGM、UPDPGM、または UPDSRVPGM コマンドからの出力リスト』に示しています。

このバインド・プログラム言語の例では、各モジュールに複数のプロシージャーが入ります。この例は、プロシージャーが 1 つしか含まれないモジュールにも当てはまります。

バインド・プログラム言語の例 1

Rate、Amount、Payment、および Term の各プロシージャーのバインド・プログラム言語は以下のようになります。

ファイル: MYLIB/QSRVSRC メンバー: FINANCIAL

```
STRPGMEXP  PGMLVL(*CURRENT)
  EXPORT SYMBOL('Term')
  EXPORT SYMBOL('Rate')
  EXPORT SYMBOL('Amount')
  EXPORT SYMBOL('Payment')
ENDPGMEXP
```

設計の初期の段階でいくつかの決定が行われ、3 つのモジュール (MONEY、RATES、および CALCS) が必要なプロシージャーを提供します。

97 ページの図 34 に示したサービス・プログラムを作成するために、CRTSRVPGM コマンドにバインド・プログラム言語が指定されています。

```
CRTSRVPGM  SRVPGM(MYLIB/FINANCIAL)
           MODULE(MYLIB/MONEY MYLIB/RATES MYLIB/CALCS)
           EXPORT(*SRCFILE)
           SRCFILE(MYLIB/QSRVSRC)
           SRCMBR(*SRVPGM)
```

SRCFILE パラメーターに指定されているライブラリー MYLIB のソース・ファイル QSRVSRC がバインド・プログラム言語ソースを含むファイルであることに注意してください。

また、サービス・プログラムの作成に必要なすべてのモジュールが MODULE パラメーターに指定されているので、バインディング・ディレクトリーは不要であることにも注意してください。

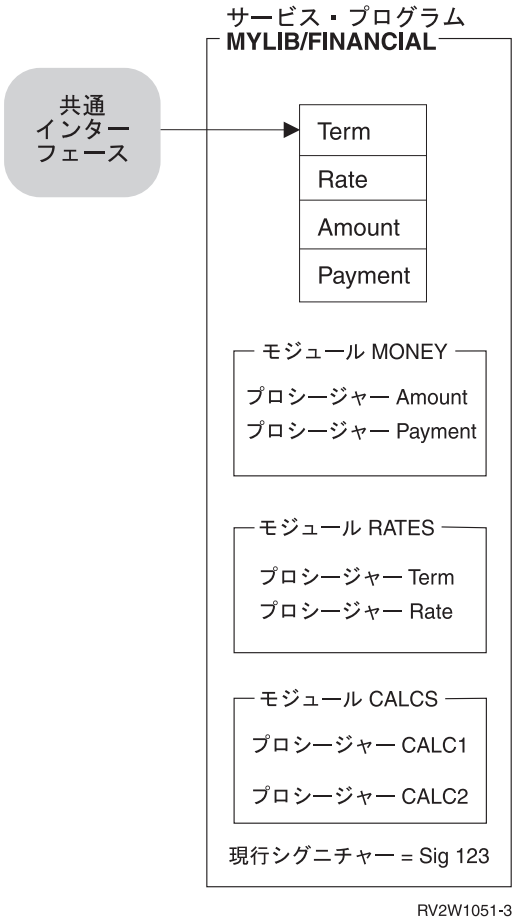

図 34. バインド・プログラム言語の使用によるサービス・プログラムの作成

バインド・プログラム言語の例 2

このアプリケーションの開発を進める過程で、BANKER というプログラムを作成することになりました。BANKER はサービス・プログラム FINANCIAL の Payment というプロシージャーを使用する必要があります。BANKER プログラムを追加した結果としてのアプリケーションを 98 ページの図 35 に示しています。

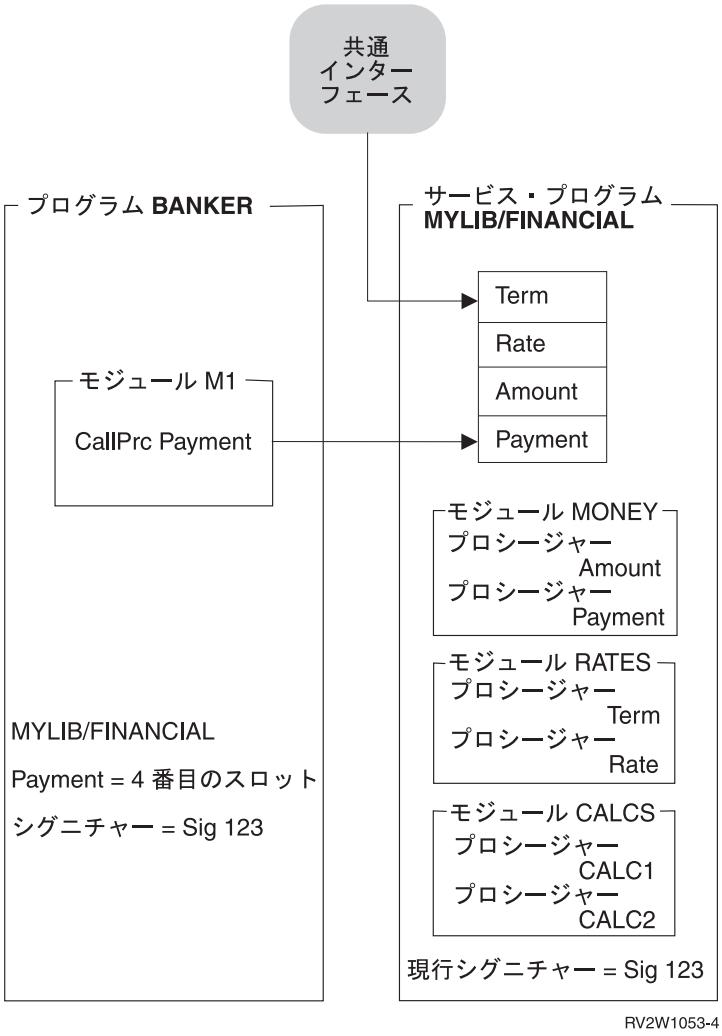

図 35. サービス・プログラム FINANCIAL の使用法

BANKER プログラムを作成した時点で、BNDSRVPGM パラメーターにサービス・プログラム MYLIB/FINANCIAL を指定しました。記号 Payment が、サービス・プログラム FINANCIAL の共通インターフェースの 4 番目のスロットからエクスポートされるものとして見つかります。MYLIB/FINANCIAL の現行シグニチャーおよび Payment インターフェースに関連するスロットが、BANKER プログラムとともに保管されます。

BANKER を実行可能にするプロセスの過程で、活動化によって以下が検査されます。

- ライブラリー MYLIB にサービス・プログラム FINANCIAL が存在すること。
- サービス・プログラムが、BANKER に保管されているシグニチャー (SIG 123) をまだサポートしていること。

このシグニチャー検査によって、BANKER の作成時に使用された共通インターフェースが、実行時にまだ有効であるかどうかが検査されます。

図 35 に示したように、BANKER が呼び出される時点で、BANKER が使用する共通インターフェースを、MYLIB/FINANCIAL がまだサポートしています。 活動化

によって、MYLIB/FINANCIAL に一致するシグニチャーが見つからないか、またはサービス・プログラム MYLIB/FINANCIAL が見つからない場合、以下のようになります。

BANKER の活動化が失敗する。

エラー・メッセージが出される。

バインド・プログラム言語の例 3

アプリケーションがさらに大規模になり、この金融パッケージに 2 つの新しいプロシージャーを追加することになりました。2 つの新しいプロシージャー OpenAccount と CloseAccount は、それぞれ口座を開設し閉鎖します。プログラム BANKER を再作成せずに MYLIB/FINANCIAL を更新するには、以下のステップを行う必要があります。

1. プロシージャー OpenAccount および CloseAccount を作成する。
2. 新しいプロシージャーを指定してバインド・プログラム言語を更新する。

更新されたバインド・プログラム言語は、新しいプロシージャーをサポートします。また、更新されたバインド・プログラム言語によって、FINANCIAL サービス・プログラムを使用する既存の ILE プログラムまたはサービス・プログラムを変更せずに済みます。このバインド・プログラム言語は以下のようになります。

ファイル: MYLIB/QSRVSRC メンバー: FINANCIAL

```
STRPGMEXP  PGMLVL(*CURRENT)
  EXPORT SYMBOL('Term')
  EXPORT SYMBOL('Rate')
  EXPORT SYMBOL('Amount')
  EXPORT SYMBOL('Payment')
  EXPORT SYMBOL('OpenAccount')
  EXPORT SYMBOL('CloseAccount')
ENDPGMEXP

STRPGMEXP  PGMLVL(*PRV)
  EXPORT SYMBOL('Term')
  EXPORT SYMBOL('Rate')
  EXPORT SYMBOL('Amount')
  EXPORT SYMBOL('Payment')
ENDPGMEXP
```

以下の両方を行うために、サービス・プログラムの更新操作が必要になった場合、

- 新しいプロシージャーまたはデータ項目をサポートする。
- 変更後のサービス・プログラムを使用する既存のプログラムおよびサービス・プログラムを変更しないで済ませる。

2 つの代案のうち、1 つを選択する必要があります。 最初の方法は、以下のステップを行います。

1. PGMLVL(*CURRENT) を含む STRPGMEXP、ENDPGMEXP ブロックを複写します。
2. 複写された PGMLVL(*CURRENT) の値を PGMLVL(*PRV) に変更します。
3. PGMLVL(*CURRENT) を含む STRPGMEXP コマンドのリストの最後に、エクスポートする新しいプロシージャーまたはデータ項目を追加します。
4. 変更結果をソース・ファイルに保管します。
5. 新しいモジュールを作成するか、または変更済みモジュールを再作成します。

6. 更新されたバインド・プログラム言語を使用して、新しいモジュールまたは変更されたモジュールからサービス・プログラムを作成します。

2 番目の方法は、STRPGMEXP コマンドのシグニチャー (インターフェース識別値) パラメーターを使用することと、エクスポート・ブロックの終わりに新しい記号を加えることです。

```
STRPGMEXP PGMVAL(*CURRENT) SIGNATURE('123')
  EXPORT SYMBOL('Term')
    .
    .
    .
  EXPORT SYMBOL('OpenAccount')
  EXPORT SYMBOL('CloseAccount')
ENDPGMEXP
```

101 ページの図 36 に示した拡張されたサービス・プログラムを作成するために、99 ページに示した更新されたバインド・プログラム言語が以下の CRTSRVPGM コマンドで使用されます。

```
 CRTSRVPGM  SRVPGM(MYLIB/FINANCIAL)
            MODULE(MYLIB/MONEY MYLIB/RATES MYLIB/CALCS MYLIB/ACCOUNTS))
            EXPORT(*SRCFILE)
            SRCFILE(MYLIB/QSRVSRC)
            SRCMBR(*SRVPGM)
```

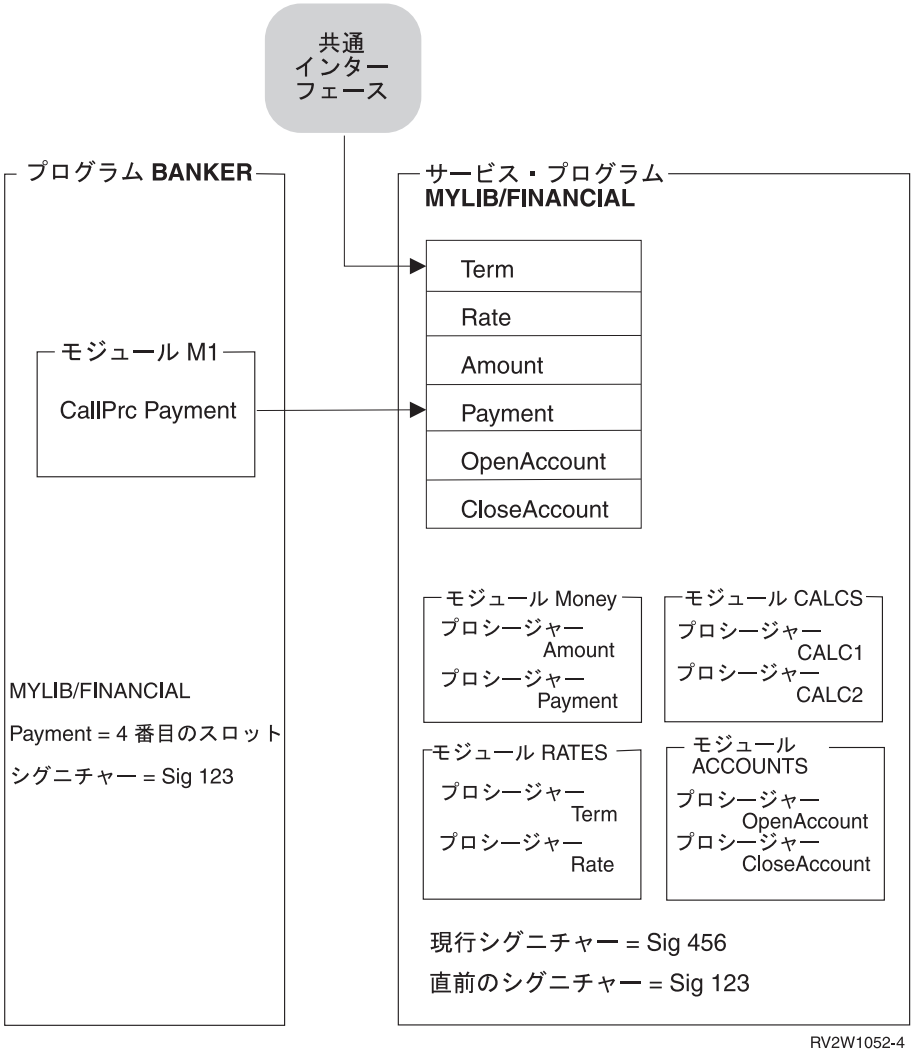

図 36. バインド・プログラム言語の使用によるサービス・プログラムの更新

BANKER プログラムは、前のシグニチャーがまだサポートされているので、変更する必要はありません。 (サービス・プログラム MYLIB/FINANCIAL の前のシグニチャーと BANKER に保管されているシグニチャーを参照してください。) BANKER を CRTPGM コマンドによって再作成すると、BANKER とともに保管されるシグニチャーは、サービス・プログラム FINANCIAL の現行シグニチャーになります。プログラム BANKER を再作成する唯一の理由は、サービス・プログラム FINANCIAL によって提供される新しいプロシージャーの 1 つをプログラム BANKER が使用する場合があるからです。バインド・プログラム言語によって、変更後のサービス・プログラムを使用するプログラムまたはサービス・プログラムを変更せずに、サービス・プログラムを拡張することができます。

バインド・プログラム言語の例 4

更新後のサービス・プログラム FINANCIAL を出荷した後で、以下の項目に基づく利率の作成を依頼されたとします。

Rate プロシージャーの現行パラメーター
申込者のクレジット歴

Credit_History という 5 番目のパラメーターを、Rate プロシージャーの呼び出しに追加しなければなりません。Credit_History は、Rate プロシージャーから戻される Interest_Rate パラメーターを更新します。他の要件として、サービス・プログラム FINANCIAL を使用する既存の ILE プログラムまたはサービス・プログラムを変更しないで済むようにしなければなりません。使用中の言語が、可変数のパラメーターの引き渡しをサポートしていない場合、以下の両方の実現は困難と思われます。

- サービス・プログラムの更新
- FINANCIAL サービス・プログラムを使用する他のすべてのオブジェクトの再作成の回避

しかし、幸いなことに、実現する方法があります。以下のバインド・プログラム言語は、更新後の Rate プロシージャーをサポートします。さらに、FINANCIAL サービス・プログラムを使用する既存の ILE プログラムまたはサービス・プログラムを変更せずに済みます。

ファイル: MYLIB/QSRVSRC メンバー: FINANCIAL

```
STRPGMEXP  PGMLVL(*CURRENT)
  EXPORT SYMBOL('Term')
  EXPORT SYMBOL('Old_Rate')  /* 4 つのパラメーターを持つオリジナル Rate
                                プロシージャー */
  EXPORT SYMBOL('Amount')
  EXPORT SYMBOL('Payment')
  EXPORT SYMBOL('OpenAccount')
  EXPORT SYMBOL('CloseAccount')
  EXPORT SYMBOL('Rate')           /* 5 番目のパラメーター Credit_History +
                                     をサポートする新しい Rate プロシージャー */
ENDPGMEXP

STRPGMEXP  PGMLVL(*PRV)
  EXPORT SYMBOL('Term')
  EXPORT SYMBOL('Rate')
  EXPORT SYMBOL('Amount')
  EXPORT SYMBOL('Payment')
  EXPORT SYMBOL('OpenAccount')
  EXPORT SYMBOL('CloseAccount')
ENDPGMEXP

STRPGMEXP  PGMLVL(*PRV)
  EXPORT SYMBOL('Term')
  EXPORT SYMBOL('Rate')
  EXPORT SYMBOL('Amount')
  EXPORT SYMBOL('Payment')
ENDPGMEXP
```

元の記号 Rate は Old_Rate と名前変更されていますが、エクスポートする記号の同じ相対位置に残っています。これは重要なので、覚えておいてください。

コメントは Old_Rate 記号に関する記述です。コメントは /* と */ の間のすべてです。 バインド・プログラムは、サービス・プログラムの作成時点で、バインド・プログラム言語ソースのコメントを無視します。

追加のパラメーター Credit_History をサポートする新しいプロシージャー Rate もエクスポートしなければなりません。この更新済みのプロシージャーはエクスポートのリストの最後に追加されます。

元の Rate プロシージャーを処理するには、以下の 2 つの方法があります。

- 4 つのパラメーターをサポートする元の Rate プロシージャーを Old_Rate に名前変更します。Old_Rate プロシージャーを複写します (Rate という名前にします)。 5 番目のパラメーター Credit_History をサポートするようにコードを更新します。
- 5 番目のパラメーター Credit_History をサポートするように、元の Rate プロシージャーを更新します。 Old_Rate という名前の新しいプロシージャーを作成します。Old_Rate は、Rate の元の 4 つのパラメーターをサポートします。また、Old_Rate は更新済みの新しい Rate プロシージャーを、5 番目のパラメーターとしてダミーを指定して呼び出します。

  このほうが、保守り簡単で、オブジェクトのサイズが小さいので、より望ましい方法といえます。

更新済みのバインド・プログラム言語、およびプロシージャー Rate、Term、および Old_Rate をサポートする新しい RATES モジュールを使用すれば、以下の FINANCIAL サービス・プログラムを作成することができます。

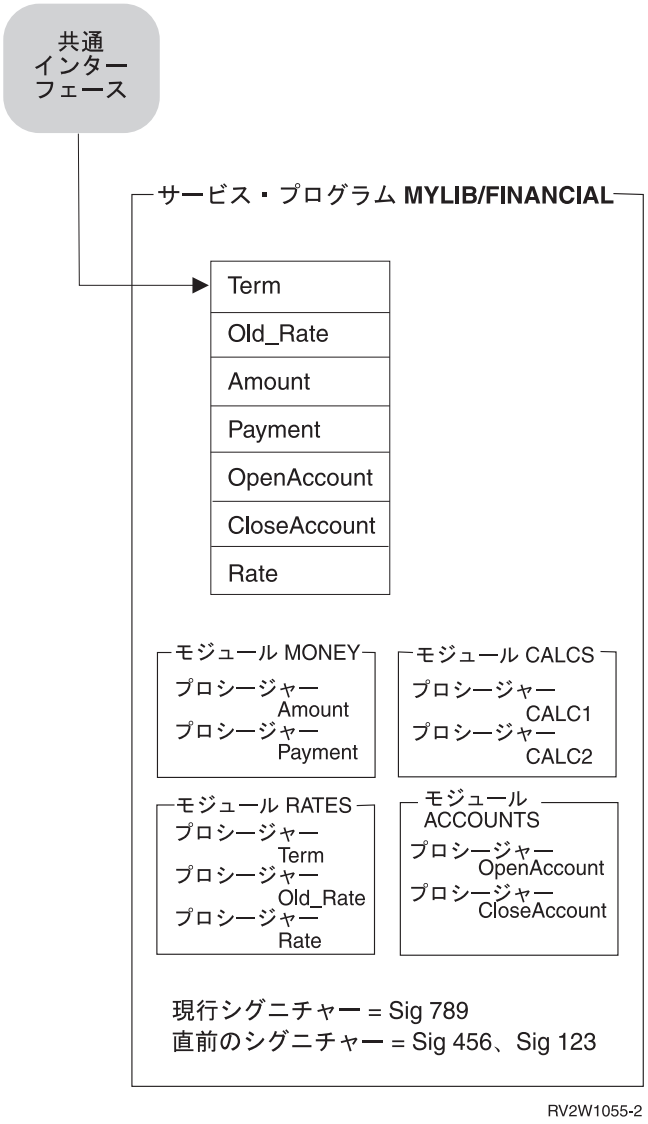

図 37. バインド・プログラム言語の使用によるサービス・プログラムの更新

FINANCIAL サービス・プログラムの元の Rate プロシージャーを使用する ILE プログラムおよびサービス・プログラムは、スロット 2 に行きます。これは、呼び出しを Old_Rate プロシージャーに向けるので有利です。なぜなら、Old_Rate プロシージャーは、元の 4 つのパラメーターを処理するからです。元の Rate プロシージャーを使用していた ILE プログラムまたはサービス・プログラムを再作成する必要がある場合、以下のいずれかを行います。

- 元の 4 つのパラメーターがある Rate プロシージャーを使用し続けたい場合には、Rate プロシージャーの代わりに Old_Rate プロシージャーを呼び出してください。
- 新しい Rate プロシージャーを使用したい場合には、Rate プロシージャーの各呼び出しに、5 番目のパラメーター Credit_History を追加してください。

サービス・プログラムの更新が、以下の要件を満たす必要がある場合には、

- 処理できるパラメーターの個数を変更したプロシージャーをサポートする。

- 変更後のサービス・プログラムを使用する既存のプログラムおよびサービス・プログラムを変更しなくてもよいようにする。

この 2 点が必要な場合には、以下のステップを行う必要があります。

1. PGMLVL(*CURRENT) を含む STRPGMEXP、ENDPGMEXP ブロックを複写します。
2. 複写された PGMLVL(*CURRENT) の値を PGMLVL(*PRV) に変更します。
3. PGMLVL(*CURRENT) を含む STRPGMEXP コマンドで、元のプロシージャー名を変更しますが、同じ相対位置に残します。

   この例では、Rate が Old_Rate に変更されましたが、エクスポートする記号のリスト内で同じ相対位置に残っています。
4. PGMLVL(*CURRENT) を指定した STRPGMEXP コマンドで、元のプロシージャー名を、異なる個数のパラメーターをサポートするリストの最後に置きます。

   この例では、Rate が、エクスポートされる記号のリストの最後に追加されています。この Rate プロシージャーは、追加のパラメーター Credit_History をサポートします。
5. 変更結果をバインド・プログラム言語ソース・ファイルに保管します。
6. ソース・コードを含んでいるファイルで、元のプロシージャーを拡張して、新しいパラメーターをサポートするようにします。

   この例では、これは、5 番目のパラメーター Credit_History をサポートするよう既存の Rate プロシージャーを変更することを意味します。
7. 元のパラメーターを入力として処理する新しいプロシージャーを作成します。このプロシージャーは、ダミー・パラメーターを追加して新しいプロシージャーを呼び出します。

   この例では、これは、元のパラメーターを処理する Old_Rate プロシージャーを追加し、5 番目のパラメーターとしてダミーを指定して新しい Rate プロシージャーを呼び出すことを意味します。
8. バインド・プログラム言語ソース・コードの変更結果を保管します。
9. 新しいプロシージャーおよび変更したプロシージャーによりモジュール・オブジェクトを作成します。
10. 更新したバインド・プログラム言語を使用して、新しいモジュールおよび変更したモジュールからサービス・プログラムを作成します。

プログラム変更

プログラム変更 (CHGPGM) コマンドは、プログラムの属性を変更します。プログラムの再コンパイルは必要としません。変更可能な属性のいくつかを、以下に示します。

- 最適化属性。
- ユーザー・プロファイル属性。
- 借用権限使用属性。
- パフォーマンス収集属性。
- プロファイル作成データ属性。
- プログラム・テキスト。

- ライセンス内部コードのオプション。
- テラスペース属性。

指定した属性が現行の属性と同じである場合でも、プログラムの再作成を強制することができます。これは、プログラム再作成の強制 (FRCCRT) パラメーターに値 *YES を指定することによって行うことができます。

プログラム再作成の強制 (FRCCRT) パラメーターには、値 *NO および *NOCRT を指定することもできます。これらの値によって、変更によってプログラムの再作成が必要となる場合に、要求されたプログラム属性が実際に変更されるかどうかが決まります。以下のプログラム属性を変更すると、プログラムが再作成される可能性があります。

- プログラムの最適化プロンプト (OPTIMIZE パラメーター)。
- 借用権限の使用プロンプト (USEADPAUT パラメーター)。
- パフォーマンス収集使用可能プロンプト (ENBPFRCOL パラメーター)。
- プロファイルリング。データ・プロンプト (PRFDTA パラメーター)。
- ユーザー・プロファイル・プロンプト (USRPRF パラメーター)。
- ライセンス内部コード・オプション・プロンプト (LICOPT パラメーター)。
- テラスペース・プロンプト (TERASPACE パラメーター)。

プログラム再作成の強制 (FRCCRT) パラメーターに値 *NO を指定した場合、再作成は強制されませんが、再作成を要求するプログラム属性のうちのいずれかが変更されたときには、プログラムが再作成されます。このオプションにより、システムは、変更が必要かどうかを判別します。

1 つ以上のジョブがプログラムを使用しているときに CHGPGM または CHGSRVPGM を指定してそのプログラムを再作成すると、「オブジェクトの破棄」例外が起こり、それらのジョブは失敗します。プログラム再作成の強制 (FRCCRT) パラメーターのコマンドのデフォルトを *NOCRT に変更すると、不注意によってこのようなことが起こるのを防ぐことができます。

プログラムの更新

ILE プログラム・オブジェクトまたはサービス・プログラムが作成された後は、それに対してエラーの訂正または拡張の追加をすることができます。しかし、オブジェクトの保守を行った後は、オブジェクトが大きくなりすぎて、オブジェクト全体を出荷することは困難であり、また費用もかかる場合があります。

プログラムの更新 (UPDPGM) コマンドまたはサービス・プログラムの更新 (UPDSRVPGM) コマンドを使って出荷サイズを減らすことができます。これらのコマンドは、指定されたモジュールのみを置き換えます。そして、変更、または追加されたモジュールだけが顧客に出荷されます。

PTF 処理を使用する場合には、UPDPGM または UPDSRVPGM コマンドに対する 1 つ以上の呼び出しを含む出口プログラムを使用して、更新機能を行うことができます。同一のモジュールを複数のプログラム・オブジェクトやサービス・プログラムにバインドするには、それぞれの *PGM および *SRVPGM オブジェクトごとに UPDPGM または UPDSRVPGM コマンドを実行する必要があります。

例えば、図 38 を参照してください。

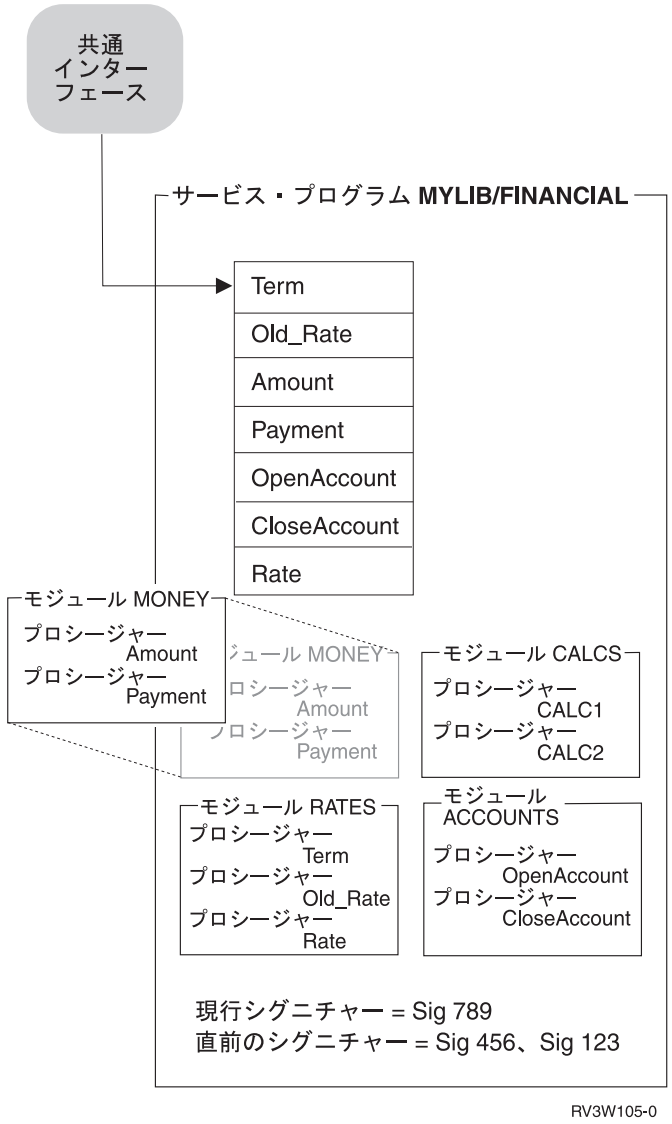

図 38. サービス・プログラムでのモジュールの置き換え

別のジョブで活動状態になっているプログラムまたはサービス・プログラムが更新された場合、そのジョブでは、旧バージョンのプログラムまたはサービス・プログラムが引き続き使用されます。新たに活動化を行うと、そのプログラムまたはサービス・プログラムの更新されたバージョンが使用されます。

CRTPGM または CRTSRVPGM コマンドの更新可能 (ALWUPD) パラメーターおよび *SRVPGM ライブラリー更新可能 (ALWLIBUPD) パラメーターは、プログラム・オブジェクトまたはサービス・プログラムのどちらが更新できるかを判別します。ALWUPD(*NO) が指定されると、プログラム・オブジェクトまたはサービス・プログラムにあるモジュールは、UPDPGM または UPDSRVPGM コマンドによって置き換えられません。ALWUPD(*YES) および ALWLIBUPD(*YES) の指定により、プログラムを更新して、以前には指定されていなかったライブラリーからのサービス・プログラムを使用することができます。ALWUPD(*YES) および ALWLIBUPD (*NO) の指定により、モジュールの更新はできますが、バインドされるサービス・

プログラム・ライブラリーの更新はできません。ALWUPD(*NO) および ALWLIBUPD(*YES) を同時に指定することはできません。

UPDPGM および UPDSRVPGM コマンドのパラメーター

モジュール・パラメーターで指定された各モジュールは、プログラム・オブジェクトまたはサービス・プログラムにバインドされたものと同じ名前を持つモジュールを置き換えます。プログラム・オブジェクトまたはサービス・プログラムにバインドされている複数のモジュールが同じ名前を持っている場合には、置き換えライブラリー (RPLLIB) パラメーターが使用されます。このパラメーターは、置き換えるべきモジュールを選択する方法を指定します。同じ名前を持つモジュールが既にプログラム・オブジェクトまたはサービス・プログラムにバインドされていない場合には、プログラム・オブジェクトまたはサービス・プログラムは更新されません。

サービス・プログラムのバインド (BNDSRVPGM) パラメーターは、既にバインドされているプログラム・オブジェクトまたはサービス・プログラムに加えて追加のサービス・プログラムを指定します。置き換えモジュールが置き換えるモジュールより多くのインポートまたはより少ないエクスポートを含んでいる場合には、これらのサービス・プログラムがこのインポートを解決するために必要となることがあります。

サービス・プログラム・ライブラリー (SRVPGMLIB) パラメーターを使用すると、バインドされたサービス・プログラムを保管するライブラリーを指定できます。UPDPGM または UPDSRVPGM コマンドを実行するたびに、指定されたライブラリーから得られた、バインドされたサービス・プログラムが使用されます。ALWLIBUPD(*YES) が使用されると、UPDPGM または UPDSRVPGM コマンドによってライブラリーを変更できます。

ディレクトリーのバインディング (BNDDIR) パラメーターは、追加のインポートを解決するために必要になるモジュールやサービス・プログラムを含むバインディング・ディレクトリーを指定します。

活動化グループ (ACTGRP) パラメーターでは、プログラムまたはサービス・プログラムを活動化するときに使用される活動化グループ名を指定します。このパラメーターを使用して、指定された活動化グループの活動化グループ名を変更することもできます。

より少ないインポートを持つモジュールにより置き換えられるモジュール

モジュールがより少ないインポートを持つ別のモジュールによって置き換えられた場合には、新しいプログラム・オブジェクトまたはサービス・プログラムが常に作成されます。ただし、以下の条件が存在する場合には、更新されたプログラム・オブジェクトまたはサービス・プログラムが分離されたモジュールを含みます。

- 今欠落しているインポートのために、プログラム・オブジェクトまたはサービス・プログラムにバインドされたモジュールの 1 つが、もはやインポートを解決しなくなります。
- そのモジュールは、元は、CRTPGM または CRTSRVPGM コマンドで使用されたバインディング・ディレクトリーからきたものです。

分離されたモジュールを持つプログラムは、時間とともに著しく成長します。もはやインポートを解決しない、元はバインディング・ディレクトリーからきているモジュールを除去するには、オブジェクトを更新する際に OPTION(*TRIM) を指定することができます。しかし、このオプションを使用すると、このモジュールが含むエクスポートは、将来のプログラム更新には使用不能になります。

より多いインポートを持つモジュールにより置き換えられるモジュール

モジュールがより多いインポートを持つモジュールにより置き換えられる場合に、この追加のインポートが解決されて、以下のようなことがあると、プログラム・オブジェクトまたはサービス・プログラムは更新することができます。

- モジュールの既存のセットがオブジェクト内にバインドされている。
- サービス・プログラムがオブジェクトにバインドされている。
- バインディング・ディレクトリーがコマンド上に指定されている。これらのバインディング・ディレクトリーのうちの 1 つにあるモジュールが必要とされるエクスポートを含む場合、モジュールはプログラムまたはサービス・プログラムに加えられます。これらのバインディング・ディレクトリーのうちの 1 つにあるサービス・プログラムが必要とされるエクスポートを含む場合、サービス・プログラムはプログラムまたはサービス・プログラムへの参照によってバインドされます。
- 暗黙的バインディング・ディレクトリー。**暗黙的バインディング・ディレクトリー**は、モジュールを含むプログラムの作成に必要なエクスポートを含むバインディング・ディレクトリーです。各 ILE コンパイラーは、作成するモジュールそれぞれに暗黙的バインディング・ディレクトリーのリストを組み入れます。

これら追加のインポートが解決されなければ、OPTION(*UNRSLVREF) が更新コマンドに指定されていない限り、更新操作は失敗します。

より少ないエクスポートを持つモジュールにより置き換えられるモジュール

モジュールがより少ないエクスポートを持つ別のモジュールにより置き換えられた場合、以下の条件があると更新が起こります。

- 欠落しているエクスポートが、バインディングに必要でない。
- 欠落しているエクスポートが、UPDSRVPGM の場合にサービス・プログラムからエクスポートされない。

EXPORT(*ALL) を指定してサービス・プログラムを更新すると、新規エクスポート・リストが作成されます。新規エクスポート・リストは、オリジナルのエクスポート・リストとは異なっています。

以下の条件が存在すると、更新は起こりません。

- インポートのいくつかが、欠落しているエクスポートのために解決できない。
- 欠落しているエクスポートが、コマンド上で指定されたエクストラのサービス・プログラムとバインディング・ディレクトリーから検出することができない。
- バインド・プログラム言語は記号のエクスポートを示しているが、エクスポートが欠落している。

より多いエクスポートを持つモジュールにより置き換えられるモジュール

モジュールがより多いエクスポートを持つ別のモジュールにより置き換えられる場合に、すべての追加のエクスポートが固有の名前を持っていると、更新操作が起こります。サービス・プログラム・エクスポートは、EXPORT(*ALL) が指定されていると異なります。

しかし、1 つ以上の追加のエクスポートに固有の名前が無いと、重複した名前が問題を起こします。

- OPTION(*NODUPPROC) または OPTION(*NODUPVAR) が更新コマンド上に指定されていると、プログラム・オブジェクトまたはサービス・プログラムは更新されません。
- OPTION(*DUPPROC) または OPTION(*DUPVAR) を指定した場合、更新は行われますが、オリジナルのエクスポートではなく、それと同じ名前の追加のエクスポートが使用される可能性があります。

モジュール、プログラム、およびサービス・プログラムの作成上のヒント

モジュール、ILE プログラム、およびサービス・プログラムを便利よく作成し保持するには、以下について考慮してください。

- プログラムまたはサービス・プログラムを作成するためにコピーされるモジュールは命名規則に従ってください。

  共通の接頭語を使用する命名の方法によって、モジュールをモジュール・パラメーターで総称的に指定しやすくなります。
- 保守を容易にするためには、プログラムまたはサービス・プログラム 1 つだけに各モジュールを組み込んでください。複数のプログラムに 1 つのモジュールを使用する必要がある場合には、モジュールをサービス・プログラムに入れてください。このようにすると、モジュールを再設計するときに、一個所でモジュールを再設計するだけですみます。
- シグニチャーの確保には、サービス・プログラムの作成に必ずバインド・プログラム言語を使用してください。

  バインド・プログラム言語によって、使用するプログラムおよびサービス・プログラムを再作成せずに、サービス・プログラムを容易に更新することができます。

  バインダー・ソース検索 (RTVBNDSRC) コマンドを使用すると、1 つ以上のモジュールまたはサービス・プログラムからのエクスポートに基づいてバインド・プログラム言語ソースを生成するのに役立ちます。

  以下のいずれかの条件が存在する場合、

  – サービス・プログラムが決して変更されない
  – シグニチャーが変更されても、プログラムの変更によってサービス・プログラムのユーザーが影響を受けない

  バインド・プログラム言語を使用する必要はありません。しかし、この状態は、ほとんどのアプリケーションには起こり得ないので、すべてのサービス・プログラムについてバインド・プログラム言語の使用を考慮してください。

- CRTPGM、CRTSRVPGM、あるいは UPDPGM のようなプログラム作成コマンドを使用していて CPF5D04 メッセージを受け取ったが、それでもプログラムまたはサービス・プログラムが作成された場合は、以下の 2 つのことが考えられます。

  1. プログラムが OPTION(*UNRSLVREF) を指定して作成されたが、未解決の参照がある。
  2. *PUBLIC *EXCLUDE 権限を付けて出荷された、*BNDDIR QSYS/QUSAPIBD にリストされている *SRVPGM をバインドしているが、権限をもっていない。オブジェクトに対する権限を誰が持っているかを見るには、DSPOBJAUT コマンドを使用してください。システムの *BNDDIR QUSAPIBD には、システム API を提供する *SRVPGMs の名前が入っています。これらの API のいくつかはセキュリティー依存であるため、それらが入っている *SRVPGM は *PUBLIC *EXCLUDE 権限を付けて出荷されます。これらの *SRVPGM は QUSAPIBD の終了時にグループにまとめられます。このリストにある *PUBLIC *EXCLUDE サービス・プログラムを使用する際は、通常、バインド・プログラムは他の *PUBLIC *EXCLUDE *SRVPGM をユーザーのものより先に調べる必要があり、CPF5D04 が出ます。

  CPF5D04 メッセージが出ないようにするには、以下のいずれかの方法を使用します。

  – プログラムあるいはサービス・プログラムをバインドさせる任意の *SRVPGM を明示的に指定する。プログラムあるいはサービス・プログラムをバインドさせる *SRVPGM のリストを見るには、 DSPPGM または DSPSRVPGM DETAIL(*SRVPGM) を使用します。これらの *SRVPGM は CRTPGM または CRTSRVPGM BNDSRVPGM パラメーターに指定できます。また、これらは、CRTBNDRPG、CRTRPGMOD、CRTBNDCBL、CRTPGM、または CRTSRVPGM コマンドの BNDDIR パラメーターで示されているか、または RPG H-spec から得られた、バインディング・ディレクトリーに入れることができます。このアクションにより、すべての参照が解決されてから *BNDDIR QUSAPIBD 内の *PUBLIC *EXCLUDE *SRVPGM を調べられるようになります。

  – CPF5D04 メッセージにリストされている *SRVPGM に対して、*PUBLIC または個別に権限を付与する。この場合、セキュリティー・センシティブであるインターフェースに対して、不必要にユーザーに権限を与えてしまう可能性があるという欠点があります。

  – OPTION(*UNRSLVREF) が使用され、プログラムに未解決の参照がある場合は、 必ずすべての参照を解決するようにしてください。

- 作成しているプログラム・オブジェクトまたはサービス・プログラムを他のユーザーが使用する場合、作成時に OPTION(*RSLVREF) を指定してください。アプリケーションを開発しているとき、未解決インポートがあるプログラム・オブジェクトまたはサービス・プログラムを作成する必要が生じる場合があります。 ただし、実稼動環境では、すべてのインポートを解決すべきです。

  OPTION(*WARN) が指定されると、未解決の参照が、CRTPGM または CRTSRVPGM 要求を含むジョブ・ログにリストされます。 DETAIL パラメーターでリストを指定すれば、未解決の参照はプログラム・リストに含まれます。このジョブ・ログまたはリストを保存しておいてください。

- 新しいアプリケーションを設計する場合、1 つ以上のサービス・プログラムに入れるべき共通プロシージャーを識別できるかどうか調べてください。

  共通プロシージャーの識別と設計は、新しいアプリケーションで最も容易といえます。ILE を使用できるように既存のアプリケーションを変換する場合、サービス・プログラム用の共通プロシージャーを決定するのは、より難しくなります。それでも、アプリケーションに必要な共通プロシージャーの識別および共通プロシージャーを含むサービス・プログラムの作成を試みてください。
- 既存の 1 つのアプリケーションを ILE に変換する場合、数本の大きいプログラムの作成を考えてください。

  数個所の、通常のわずかな変更のとき、ILE 機能を利用して既存のアプリケーションを容易に変換することができます。モジュールを作成した後、モジュールを数本の大きなプログラムに結合することは、ILE に変換する最も容易で最も経済的な方法です。

  多くの小さなプログラムの代わりに少数の大きなプログラムを使用すれば、必要なストレージが少なくて済む利点も得られます。
- アプリケーションが使用するサービス・プログラムの個数の制限を試みてください。

  このために、サービス・プログラムを複数のモジュールから作成する必要が生じることがあります。利点は、活動化時間およびバインディング・プロセスの短縮です。

  アプリケーションが使用すべきサービス・プログラムの明確な個数を挙げることはできません。プログラムが数百のサービス・プログラムを使用するなら、多すぎると言えます。一方、サービス・プログラムが 1 つでは実用的ではありません。

  1 つの例として、OS/400 によって提供される言語固有ルーチンおよび共通実行時ルーチンの場合、約 10 個のサービス・プログラムを使用しています。これらの 10 個のサービス・プログラムを作成するために、70 個を超えるモジュールを使用しています。これは、パフォーマンス、理解容易性および保守容易性について良好なバランスを保つ比率と考えることができます。

第 6 章 活動化グループの管理

本章では、活動化グループを使用してアプリケーションを構成する方法の例を示します。以下のトピックについて記述します。

- 複数のアプリケーションのサポート
- OPM および ILE プログラムにおけるリソース再利用 (RCLRSC) コマンドの使用法
- 活動化グループの再利用 (RCLACTGRP) コマンドによる活動化グループの削除
- サービス・プログラムと活動化グループ

同じジョブで実行される複数のアプリケーション

ユーザー指定活動化グループを使用すると、後で使用するために活動化グループをジョブに残しておくことができます。通常の戻り操作または制御境界を超えるスキップ操作 (ILE C 用の longjmp() など) によって活動化グループが削除されることはありません。

これによって、アプリケーションを最後に使用された状態のままにしておくことができます。静的変数およびオープン・ファイルは、アプリケーションへの各呼び出しの間で変更されないまま残ります。これは、処理時間を節約できるだけでなく、提供したい機能の実行に必要になることがあります。

ただし、同じジョブで実行する複数の独立したクライアントからの要求を受け入れる用意が必要です。システムは、ILE サービス・プログラムにバインドできる ILE プログラムの数を制限しません。結果として、複数のクライアントのサポートが必要になります。

114 ページの図 39 は、ユーザー指定の活動化グループのパフォーマンスの利点を活用し、一方で、共通サービス機能を使う手法を示しています。

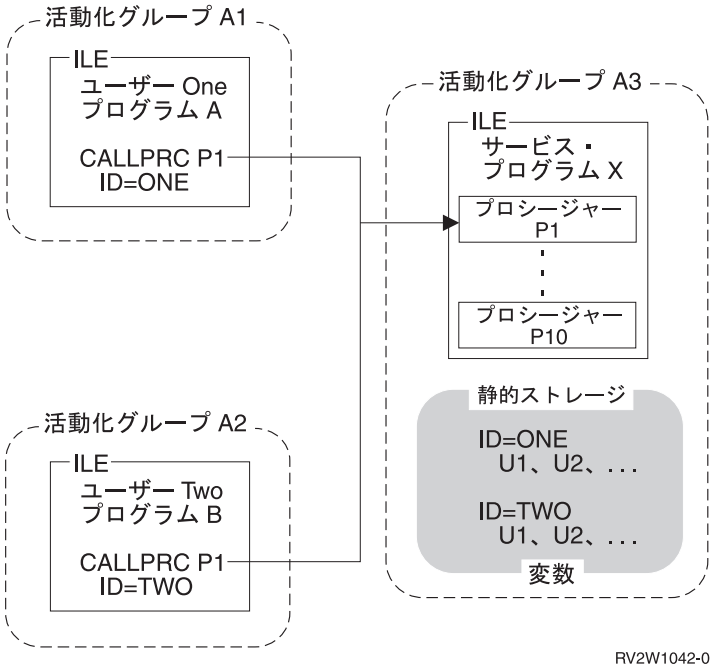

図 39. 同じジョブで実行される複数のアプリケーション

サービス・プログラム X のプロシージャーの各呼び出しには、ユーザー・ハンドルが必要です。フィールド ID はこの例のユーザー・ハンドルを示しています。各ユーザーは、このハンドルを提供する責任があります。各ユーザーに固有のハンドルを戻すための初期設定ルーチンは、自分で実行します。

使用中のサービス・プログラムに対する呼び出しが行われると、該当するユーザーに関連するストレージ変数を見つけるために、ユーザー・ハンドルが使用されます。これにより、活動化グループの作成時間を節約すると同時に、複数のクライアントをサポートすることができます。

リソース再利用コマンド

リソース再利用 (RCLRSC) コマンドは、**レベル番号**と呼ばれるシステムの概念に基づいています。 レベル番号は、ジョブで使用される特定のリソースにシステムが割り当てる固有の値です。3 つのレベル番号が以下のように定義されています。

呼び出しレベル番号
各呼び出しスタック項目には、固有のレベル番号が与えられます。

プログラム活動化レベル番号
各 OPM および ILE プログラムの活動化には、固有のレベル番号が与えられます。

活動化グループ・レベル番号
各活動化グループには固有のレベル番号が与えられます。

ジョブの実行の際に、上記のリソースの新しいオカレンスごとに、システムは固有のレベル番号の割り当てを続行します。レベル番号の値は昇順に割り当てられます。より高いレベル番号のリソースは、より低いレベル番号のリソースの後に作成されます。

図 40 は、OPM および ILE のプログラムにおける RCLRSC コマンドの使用例を示しています。この例のオープン・ファイルには、呼び出しレベルの有効範囲指定が使用されています。呼び出しレベルの有効範囲指定を使用した場合、各データ管理機能リソースには、そのリソースを作成した呼び出しスタック項目と同じレベル番号が与えられます。

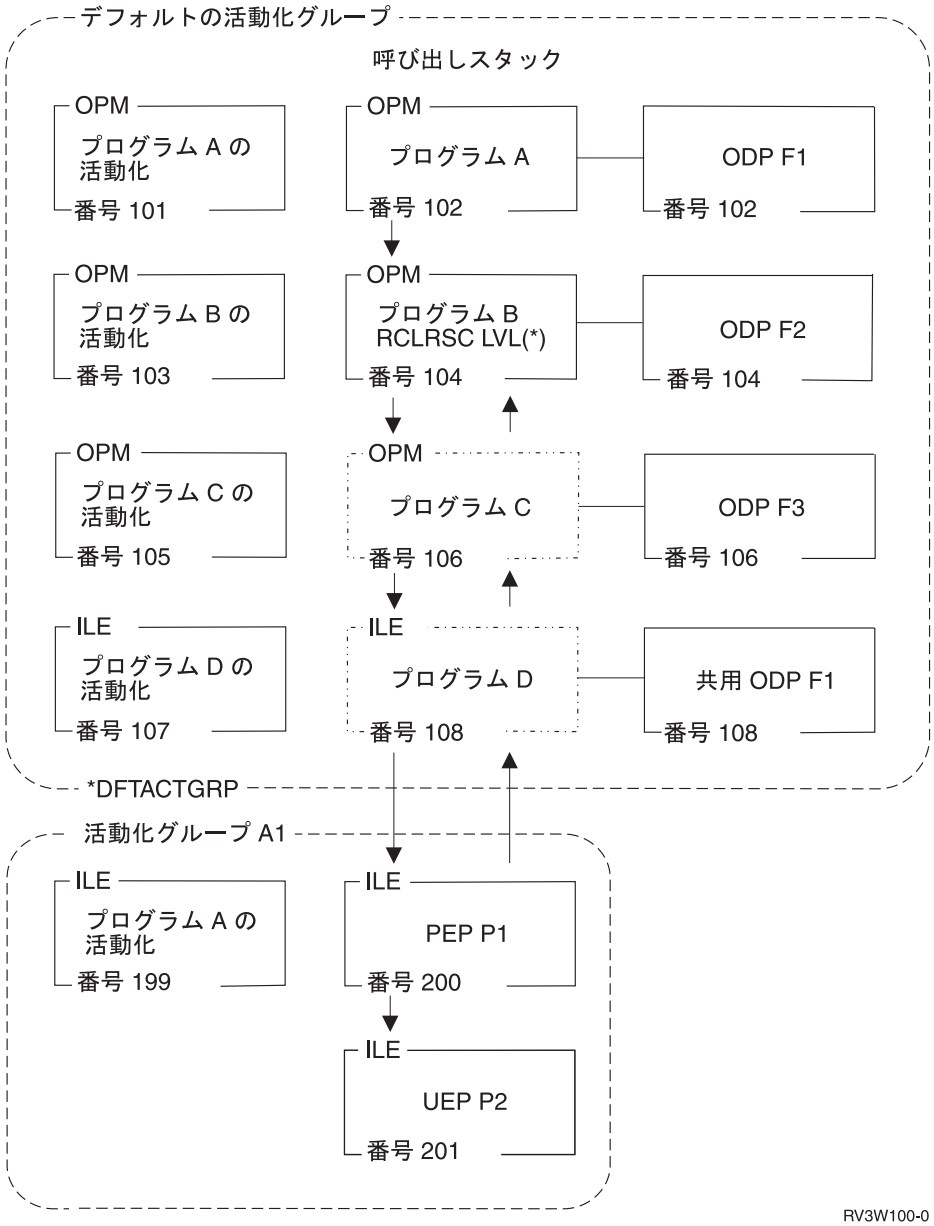

図 40. リソースの再利用

この例では、呼び出し順序はプログラム A、B、C、および D になります。プログラム D および C はプログラム B に戻ります。プログラム B は、オプションの LVL(*) を指定して RCLRSC コマンドを使おうとしています。RCLRSC コマンドはレベル (LVL) パラメーターを使用してリソースをクリーンアップします。現行呼び出しスタック項目の呼び出しレベル番号より大きな呼び出しレベル番号をもつすべてのリソースがクリーンアップされます。この例では、開始点として呼び出しレベル番号 104 が使用されます。呼び出しレベル番号が 104 より大きいすべてのリソ

ースが削除されます。呼び出しレベル番号 200 および 201 のリソースは、ILE 活動化グループにあるので、RCLRSC の影響を受けません。RCLRSC は、デフォルトの活動化グループでのみ機能します。

さらに、プログラム C と D からのストレージおよびファイル F3 のオープン・データ・パス (ODP) がクローズされます。ファイル F1 は、プログラム A でオープンされた ODP と共用されています。共用 ODP はクローズされますが、ファイル F1 はオープンされたまま残ります。

OPM プログラムの場合のリソース再利用コマンド

リソース再利用 (RCLRSC) コマンドを使用すると、終了せずに戻った OPM プログラムのオープン・ファイルをクローズし、静的ストレージを解放することができます。OPM 言語によっては (たとえば、RPG)、プログラムを終了せずに戻ることができます。後で、そのプログラムのファイルをクローズし、そのストレージを解放したい場合には、 RCLRSC コマンドを使用することができます。

ILE プログラムの場合のリソース再利用コマンド

DFTACTGRP(*YES) を指定した CRTBNDxxx コマンドによって作成された ILE プログラムの場合、RCLRSC コマンドは、OPM プログラムに対するのと同じように静的ストレージを解放します。DFTACTGRP(*YES) を指定した CRTBNDxxx コマンドにより作成されて**いない** ILE プログラムの場合には、 RCLRSC コマンドは、デフォルトの活動化グループの中で作成された活動化を再初期設定しますが静的ストレージを解放しません。大量の静的ストレージを使用する ILE プログラムは、ILE 活動化グループの中で活動化される必要があります。活動化グループの削除によって、このストレージがシステムに戻されます。 RCLRSC コマンドは、デフォルトの活動化グループの中で実行中のサービス・プログラムまたは ILE プログラムによってオープンされたファイルをクローズします。 RCLRSC コマンドは、サービス・プログラムの静的ストレージの再初期設定は行わず、またデフォルト以外の活動化グループに影響を与えることもありません。

この RCLRSC コマンドを ILE から直接使用するために、QCAPCMD API または ILE CL プロシージャーのいずれかを使用することができます。QCAPCMD API により、CL プログラムを使用せずにシステム・コマンドを直接呼び出すことができます。 115 ページの図 40 のシステム・コマンドの直接呼び出しは、特定の ILE プロシージャーの呼び出しレベル番号を使用することができるため、重要です。ILE C などの言語でも、OS/400 コマンドを直接実行できるシステム機能を備えています。

活動化グループの再利用コマンド

活動化グループの再利用 (RCLACTGRP) コマンドは、使用されていないデフォルト以外の活動化グループを削除するのに用いられます。このコマンドによって、使用可能なすべての活動化グループを削除することも、あるいは活動化グループの名前を指定して削除することもできます。

サービス・プログラムと活動化グループ

ILE サービス・プログラムを作成する場合、ACTGRP パラメーターで *CALLER オプションを指定するか、または名前を指定するかを決める必要があります。このオプションによって、サービス・プログラムが呼び出し元の活動化グループで活動化されるか、または別個に指定した活動化グループに活動化されるかが決まります。どちらを選択しても、利点と欠点があります。 このトピックでは、各オプションについて記述します。

ACTGRP(*CALLER) オプションを指定すると、サービス・プログラムは以下のように機能します。

- 迅速な静的プロシージャー呼び出し

  同じ活動化グループ内で実行されると、サービス・プログラムへの静的プロシージャー呼び出しが最適化されます。

- 外部データの共用

  サービス・プログラムは、同じ活動化グループ内の他のプログラムまたはサービス・プログラムによって使用されるデータをエクスポートすることができます。

- データ管理機能リソースの共用

  オープン・ファイルおよび他のデータ管理機能リソースを、活動化グループ内のサービス・プログラムおよび他のプログラム間で共用することができます。サービス・プログラムは、活動化グループ内の他のプログラムに影響を与えるコミット操作またはロールバック操作を行うことができます。

- 制御境界がない

  サービス・プログラム内の未処理例外は、クライアント・プログラムにパーコレートされます。サービス・プログラム内で使用される HLL 終了 verb によって、クライアント・プログラムの活動化グループを削除することができます。

ACTGRP(名前) オプションを指定した場合、サービス・プログラムは以下のように機能します。

- 変数に関する別個のアドレス・スペース

  クライアント・プログラムは、作業ストレージをアドレッシングするポインターを操作できません。これは、サービス・プログラムが借用権限によって実行中の場合に重要になる可能性があります。

- 別個のデータ管理機能リソース

  ユーザーには自分自身のオープン・ファイルとコミットメント定義があるので、オープン・ファイルの意図しない共用を防ぐことができます。

- 状態情報の制御

  アプリケーションのストレージを削除する時点を制御できます。 HLL 終了 verb または言語の通常の戻りステートメントを使用することによって、アプリケーションを削除する時点を決定することができます。ただし、複数のクライアントに関する状態情報を管理しなければなりません。

第 7 章 プロシージャー呼び出しとプログラム呼び出し

ILE の呼び出しスタックと引き数の引き渡し方式によって、言語間のコミュニケーションが容易になり、混合言語のアプリケーションの作成が容易になります。本章では 25 ページの『プログラムおよびプロシージャーの呼び出し』で紹介した動的プログラム呼び出しおよび静的プロシージャー呼び出しに関する様々な例について説明します。また、呼び出しの 3 番目のタイプであるプロシージャー・ポインター呼び出しを紹介します。

さらに、本章では、OPM および ILE のアプリケーション・プログラミング・インターフェース (API) に対するオリジナル・プログラム・モデル (OPM) サポートについて説明します。

呼び出しスタック

呼び出しスタックは、**呼び出しスタック項目**の後入れ先出し (LIFO) のリストです。呼び出された各プロシージャーまたはプログラムごとに 1 つの呼び出しスタック項目が対応しています。各呼び出しスタック項目には、プロシージャーまたはプログラムの自動変数に関する情報、および有効範囲が呼び出しスタック項目に設定されている他のリソース (たとえば、条件ハンドラーおよび取り消しハンドラー) に関する情報が含まれます。

呼び出しスタックはジョブごとに 1 つあります。1 つの呼び出しによって、呼び出されたプロシージャーまたはプログラムに関する新しい項目が呼び出しスタックに追加され、呼び出されたオブジェクトに制御が渡されます。1 つの戻りによって、スタック項目が除去され、直前のスタック項目の呼び出し元プロシージャーまたはプログラムに制御が戻されます。

呼び出しスタックの例

120 ページの図 41 は、OPM プログラム (プログラム A) と ILE プログラム (プログラム B) の 2 つのプログラムに関する呼び出しスタックのセグメントを示しています。プログラム B には、プログラム入り口プロシージャー、ユーザー入り口プロシージャー、およびもう 1 つのプロシージャー (P1) の合計 3 つのプロシージャーがあります。プログラム入り口プロシージャー (PEP) とユーザー入り口プロシージャー (UEP) の概念は 14 ページの『モジュール・オブジェクト』で定義されています。呼び出しの流れには、以下のステップが含まれます。

1. プログラム A に対する動的プログラム呼び出し
2. プログラム A が制御をプログラム B の PEP に渡して、プログラム B を呼び出す。プログラム B に対するこの呼び出しは、動的プログラム呼び出しです。
3. PEP が UEP を呼び出す。これは静的プロシージャー呼び出しです。
4. UEP がプロシージャー P1 を呼び出す。これは静的プロシージャー呼び出しです。

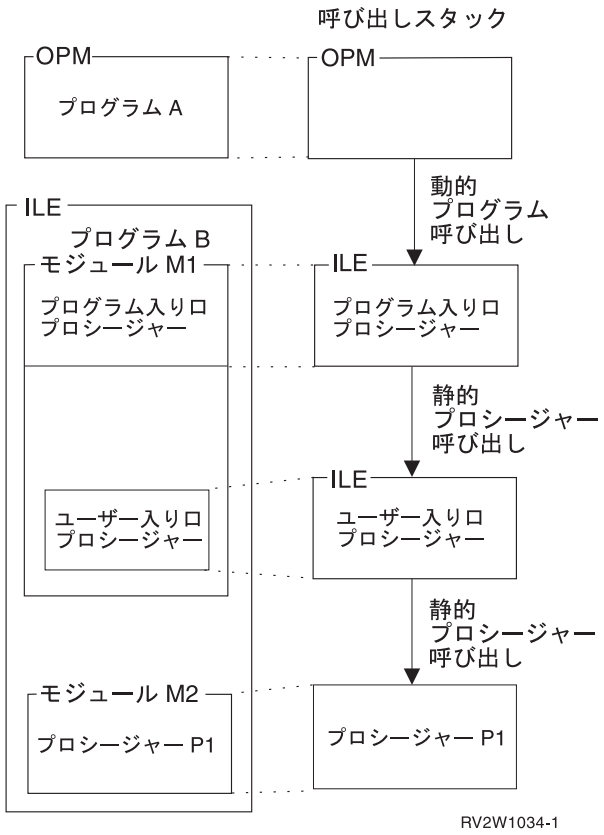

図 41. 呼び出しスタックにおける動的プログラム呼び出しと静的プロシージャー呼び出し

図 41 は、この例の呼び出しスタックを示しています。スタックで最後に呼び出された項目は、スタックの最下部に示されています。これが現在処理中の項目です。現行呼び出しスタック項目は以下のいずれかの処理を行うことができます。

- 他のプロシージャーまたはプログラムを呼び出す。これによって、スタックの最下部に別の項目が追加されます。
- 処理が終了した後、制御を呼び出し側に戻す。これによって、スタックから現行の呼び出しスタック項目が除去されます。

プロシージャー P1 が完了した後、プログラム B によるそれ以上のプロセスは必要としないと想定します。プロシージャー P1 は UEP に制御を戻し、P1 はスタックから除去されます。次に、UEP は制御を PEP に戻し、その UEP はスタックから除去されます。最後に、PEP は制御をプログラム A に戻し、その PEP はスタックから除去されます。プログラム A だけが、呼び出しスタックのこのセグメントに残されます。プログラム A は、プログラム B に対する動的プログラム呼び出しを行った点から、処理を続行します。

プログラム呼び出しとプロシージャー呼び出し

ILE の実行時に行うことのできる呼び出しには 3 つのタイプ、動的プログラム呼び出し、静的プロシージャー呼び出し、およびプロシージャー・ポインター呼び出しがあります。

ILE プログラムが活動化されると、そのプログラムの PEP を除くすべてのプロシージャーが、静的プロシージャー呼び出しおよびプロシージャー・ポインター呼び出しで使用できるようになります。プログラムが動的プログラム呼び出しによって呼び出され、活動化がすでに存在していない場合に、プログラムの活動化が起こります。プログラムが活動化されると、このプログラムにバインドされているすべてのサービス・プログラムも活動化されます。ILE サービス・プログラム内のプロシージャーは、静的プロシージャー呼び出しまたはプロシージャー・ポインター呼び出しによってのみアクセスすることができます (動的プログラム呼び出しによってはアクセスできません)。

静的プロシージャー呼び出し

ILE プロシージャーの呼び出しによって、新しい呼び出しスタック項目がスタックの最下部に追加され、制御が指定のプロシージャーに渡されます。 この呼び出しの例には以下が含まれます。

1. 同じモジュール内のプロシージャーの呼び出し
2. 同じ ILE プログラムまたはサービス・プログラム内の異なるモジュール内のプロシージャーの呼び出し
3. 同じ活動化グループ内の ILE サービス・プログラムからエクスポートされたプロシージャーの呼び出し
4. 異なる活動化グループ内の ILE サービス・プログラムからエクスポートされたプロシージャーの呼び出し

例 1、2、および 3 の場合、静的プロシージャー呼び出しは活動化グループ境界を超えません。パフォーマンスに影響を与える呼び出しのパスの長さは同じです。 この呼び出しのパスは、ILE プログラムまたは OPM プログラムの動的プログラム呼び出しのパスよりかなり短くなります。 4 の例では、呼び出しは活動化グループ境界を超えており、活動化グループのリソースを切り替えるための追加の処理が行われます。呼び出しパスの長さは、活動化グループ内の静的プロシージャー呼び出しのパスよりも長くなりますが、動的プログラム呼び出しのパスよりは短くなります。

静的プロシージャー呼び出しの場合、呼び出されるプロシージャーは、バインディング時に呼び出し側プロシージャーにバインドされていなければなりません。この呼び出しによって、常に同じプロシージャーがアクセスされます。これは、呼び出しのターゲットが各呼び出しごとに異なる可能性があるポインターを介してのプロシージャーの呼び出しとは異なります。

プロシージャー・ポインター呼び出し

プロシージャー・ポインター呼び出しは、プロシージャーを動的に呼び出す方法を提供します。たとえば、プロシージャー名またはアドレスからなる配列またはテーブルを操作することによって、1 つのプロシージャー呼び出しを複数のプロシージャーに動的に経路指定することができます。

プロシージャー・ポインター呼び出しは、静的プロシージャー呼び出しとまったく同様に、項目を呼び出しスタックに追加します。静的プロシージャー呼び出しを使用して呼び出すことができるプロシージャーはいずれも、プロシージャー・ポインターを介して呼び出すこともできます。呼び出されるプロシージャーが同じ活動化

グループにある場合、プロシージャー・ポインター呼び出しのコストは、静的プロシージャー呼び出しのコストとほとんど同じです。プロシージャー・ポインター呼び出しは、活動化されているすべての ILE プログラムのプロシージャーに対して追加のアクセスができます。

ILE プロシージャーへの引き数の引き渡し

ILE プロシージャー呼び出しでは、**引き数**は、呼び出し側プロシージャーが、呼び出しで指定されたプロシージャーに渡す値を示す式です。ILE 言語は引き数の引き渡しに 3 つの方式を使用します。

値によって、直接に
そのデータ・オブジェクトの値が引き数リストに直接入れられます。

値によって、間接に
データ・オブジェクトの値は、一時的な場所にコピーされます。 コピーのアドレス (ポインター) が引き数リストに入れられます。

参照によって
データ・オブジェクトへのポインターが引き数リストに入れられます。呼び出し先プロシージャーによって行われる引き数の変更は、呼び出し元プロシージャーに反映されます。

図 42 は、これらの引き数の引き渡しのスタイルを示しています。すべての ILE 言語が、値による直接の引き渡しをサポートしているわけではありません。使用可能な引き渡しスタイルについては、該当の ILE HLL の「プログラマーの手引き」を参照してください。

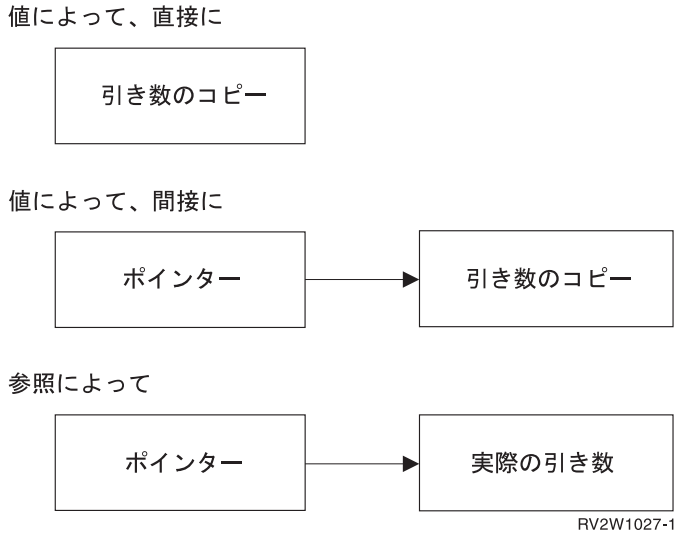

図 42. ILE プロシージャーへの引き数の引き渡し方式

データが値によって渡されるか、参照によって渡されるかは、通常、HLL のセマンティクスによって決まります。たとえば、ILE C は引き数を値によって直接に受け渡ししますが、ILE COBOL および ILE RPG は、通常、引き数を参照によって渡します。呼び出し先プロシージャーの期待する方法で、呼び出し元のプログラムま

たはプロシージャーが引き数を渡していることを確認してください。異なる言語へ引き数を渡す方法の詳細については、該当の ILE HLL の「プログラマーの手引き」を参照してください。

静的プロシージャー呼び出しの場合、最高 400 個の引き数を指定することができます。各 ILE 言語は、引き数の最大数をさらに制限している可能性があります。ILE 言語は、以下の引き数引き渡し形式をサポートします。

- ILE C および C++ は、引き数を値によって直接受け渡し、デフォルトで整数および浮動小数点値を拡大しています。 #pragma 引き数 (name, nowiden) を指定すると、引き数を拡大しないで受け渡すことができます。呼び出される関数に #pragma 引き数ディレクティブを指定することによって、引き数を値によって間接的に渡すこともできます。
- ILE COBOL は、値、参照、または間接的な値により、引き数を受け渡します。
- ILE RPG は、値または参照により、引き数を受け渡します。 RPG はデフォルトでは整数および浮動小数点を拡大しませんが、値によって受け渡されるパラメーターについては、EXTPROC(*CWIDEN) をコーディングすることによって拡大することができます。
- ILE CL は、参照および値により引き数を受け渡しします。

関数の結果

関数 (結果の引き数を戻すプロシージャー) の定義が可能な HLL をサポートするために、モデルは 図 43 に示すように特殊な関数の結果の引き数の存在を想定しています。 ILE HLL の「プログラマーの手引き」で説明されているように、関数の結果をサポートする ILE 言語は、関数の結果を戻すために共通の手段を使用しています。

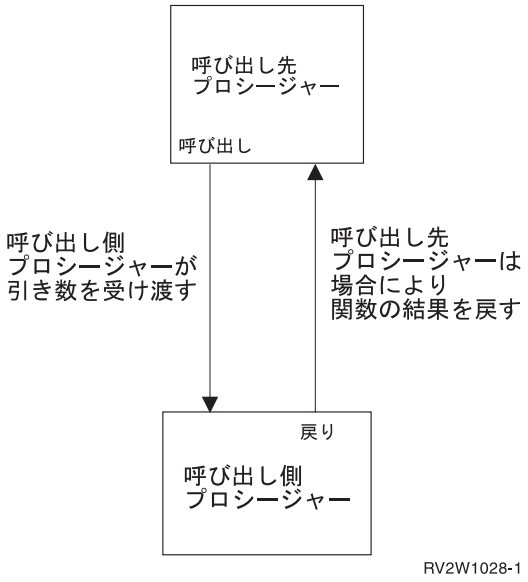

図 43. プログラム呼び出し引き数の用語

省略された引き数

すべての ILE 言語は、省略された引き数をシミュレートすることができます。これによって、ILE 条件ハンドラーおよび他の実行時プロシージャーのためのフィードバック・コードのメカニズムを使用することができます。たとえば、ILE C プロシージャーまたは ILE バインド可能 API が、参照によって渡される引き数を期待している場合、引き数のポインターの代わりにヌル・ポインターを渡すことによって引き数を省略することができます。特定の ILE 言語の中で省略された引き数を指定する方法については、該当する言語の「プログラマーの手引き」を参照してください。 iSeries Information Center の**プログラミング**・カテゴリーの中の *API* セクションには、各 API ごとに省略できる引き数が指定されています。

呼び出されたプロシージャーについて引き数が省略されているかどうかをテストする技法が組み込まれていない ILE 言語の場合、省略された引き数のテスト (CEETSTA) バインド可能 API を使用することができます。

動的プログラム呼び出し

動的プログラム呼び出しは、プログラム・オブジェクトに対して行われる呼び出しです。たとえば、CL コマンドの CALL を使用する場合、動的プログラム呼び出しを行うことを意味します。

OPM プログラムは、動的プログラム呼び出しを使用して呼び出されます。また、OPM プログラムは、動的プログラム呼び出しだけを行うことができます。

ILE プログラムは、動的プログラム呼び出しによって呼び出すこともできます。活動化された ILE プログラム内のプロシージャーには、静的プロシージャー呼び出しまたはプロシージャー・ポインター呼び出しを使ってアクセスすることができます。活動化されていない ILE プログラムは、動的プログラム呼び出しによって呼び出さなければなりません。

コンパイル時にバインドされる静的プロシージャー呼び出しと異なり、動的プログラム呼び出しにおける記号は、呼び出しの実行時にアドレスに変換されます。結果として、動的プログラム呼び出しは、静的プロシージャー呼び出しよりも多くのシステム・リソースを使用します。動的プログラム呼び出しの例を以下に示します。

- ILE プログラム、または OPM プログラムの呼び出し
- バインド不能 API の呼び出し

ILE プログラムの動的プログラム呼び出しは、指定されたプログラムの PEP に制御を渡します。次に、PEP はプログラムの UEP に制御を渡します。呼び出されたプログラムが処理を終了すると、呼び出し側プログラム命令の次の命令に制御が渡ります。

動的プログラム呼び出しでの引き数の引き渡し

ILE プログラムまたは OPM プログラムの呼び出しは (ILE プロシージャーの呼び出しと異なり)、通常、参照によって引き数を渡します。これは、呼び出されたプログラムが引き数のアドレスを受け取ることを意味します。

動的プログラム呼び出しを使用する場合、呼び出し先プログラムが期待する引き数の引き渡し方式、および必要な場合には、それをシミュレートする方法を知ってい

る必要があります。動的プログラム呼び出しでは最高 255 個の引き数を指定できます。各 ILE 言語は、引き数の最大数をさらに制限している可能性があります。 ILE 言語の中には、組み込み関数 CALLPGMV をサポートするものがあります。この関数を使用すると、最大 16383 個まで引き数を使用することができます。種々の引き渡し方式の使用法については、ILE HLL の「プログラマーの手引き」を参照してください。

言語間のデータの互換性

ILE 呼び出しの場合、異なる HLL で作成されたプロシージャー間で引き数を渡すことができます。HLL 間でのデータ共用を容易にするために、ILE 言語によってはデータ・タイプが追加されています。たとえば、ILE COBOL には、USAGE PROCEDURE-POINTER が新しいデータ・タイプとして追加されています。

HLL 間で引き数を渡すためには、各 HLL が期待する受け取りの形式を知る必要があります。呼び出しプロシージャーは、引き数が呼び出し先プロシージャーが期待するサイズとタイプであることを確認しなければなりません。たとえば、ILE C の関数は、短精度整数 (2 バイト) がパラメーター・リストに宣言されている場合でも、4 バイト整数を期待している可能性があります。引き数を渡すためにデータ・タイプの要件を満たす方法については、ILE HLL の「プログラマーの手引き」を参照してください。

混合言語アプリケーションでの引き数の引き渡しに関する構文

ILE 言語によっては、他の ILE 言語のプロシージャーに引き数を渡すための構文が用意されています。たとえば、ILE C には、値の引き数を値によって間接的に他の ILE プロシージャーに渡すための #pragma 引き数が用意されています。また RPG には、EXTPROC プロトタイプ・キーワードのための特別な値があります。

操作記述子

操作記述子は、別の HLL で作成されたプロシージャーから引き数を受け取るプロシージャーまたは API を作成する場合に有用です。**操作記述子**は、呼び出し先プロシージャーが引き数の形式 (たとえば、種々のタイプのストリング) を正確に予期できない場合に、呼び出し先プロシージャーに記述情報を提供します。この追加情報によって、プロシージャーは引き数を正確に解釈することができます。

引き数は値を提供します。操作記述子はその引き数のサイズおよびタイプに関する情報を提供します。たとえば、この情報には、文字ストリングの長さおよびストリングのタイプが含まれていることがあります。

操作記述子により、各 HLL ごとに種々のバインディングを行うバインド可能 API などのサービスが不要になり、HLL は互換不能なデータ・タイプを模倣する必要がなくなります。ILE バインド可能 API によっては、操作記述子を使用して、HLL 間の共通ストリング・データ・タイプの欠落を補っているものもあります。操作記述子の存在を API ユーザーは意識する必要はありません。

操作記述子は HLL セマンティクスをサポートしますが、操作記述子を使用しないか、それを期待しないプロシージャーにとっては目につかないものです。各 ILE 言語は、言語に適したデータ・タイプを使用することができます。各 ILE 言語コンパ

イラーは、操作記述子を生成する少なくとも 1 つの手法を提供しています。 操作記述子に関する HLL のセマンティクスの詳細については、 ILE HLL の「解説書」を参照してください。

操作記述子は、従来の他のデータ記述子とは異なります。たとえば、操作記述子は、分散データまたはファイルに関連する記述子とは関係ありません。

操作記述子の必要性

操作記述子を使用しなければならないのは、異なる ILE 言語で作成された呼び出されるプロシージャーが操作記述子を必要としている場合、および ILEバインド可能 API が操作記述子を必要としている場合です。一般的に、バインド可能 API は、ほとんどのストリング引き数に記述子を必要としています。 iSeries Information Center の**プログラミング**・カテゴリーの中の *API* セクションのバインド可能 API に関する情報に、与えられたバインド可能 API が操作記述子を必要とするかどうかが示されています。

必要な記述子の欠落

必要な記述子がないとエラーになります。プロシージャーが特定のパラメーターの記述子を必要とする場合、その要件は、そのプロシージャーのインターフェースの一部を形成します。必要な記述子が提供されていない場合、実行時にプロシージャーは失敗します。

不要な記述子の存在

必要でない記述子があっても、呼び出し先プロシージャーによる引き数のアクセスに影響を与えません。操作記述子が必要でないか、期待されていない場合、呼び出し先プロシージャーは操作記述子を無視するだけです。

注: 不要な記述子が生成された場合、言語間のコミュニケーションの障害になる可能性があります。記述子により呼び出しパスが長くなり、パフォーマンスが低下することがあります。

操作記述子アクセス用のバインド可能 API

記述子は通常、プロシージャーの作成に用いられた HLL のセマンティクスに従って、呼び出されたプロシージャーによって直接アクセスされます。プロシージャーが操作記述子を期待するようにプログラミングされていれば、プログラマーによるそれ以上の処理は通常は必要ありません。ただし、場合によっては、呼び出し先プロシージャーが記述子にアクセスする前に、必要な記述子の存在の有無を判別する必要があります。この目的のために、以下のバインド可能 API が提供されています。

- 操作記述子情報検索 (CEEDOD) バインド可能 API
- ストリング情報入手 (CEESGI) バインド可能 API

OPM および ILE API のサポート

ILE で新しい機能を開発する場合または既存のアプリケーションを ILE に変換する場合、OPM からの呼び出しレベルの API のサポートを続行することができます。このトピックでは、ILE でアプリケーションを保守する一方で、この二重サポートを行うために使用できる 1 つの手法を説明します。

ILE サービス・プログラムは、すべての ILE 言語からアクセス可能なバインド可能 API を開発して配布する方法を提供しています。同じ機能を OPM プログラムに提供する場合、ILE サービス・プログラムは OPM プログラムからは直接呼び出せない事実を考慮する必要があります。

使用する手法は、サポートしたい各バインド可能 API に ILE プログラム・スタブを開発することです。バインド可能 API につける名前は、ILE プログラム・スタブと同じでも同じでなくても構いません。各 ILE プログラム・スタブには、実際のバインド可能 API に対する静的プロシージャー呼び出しが入ります。

図 44 は、この手法の例を示しています。

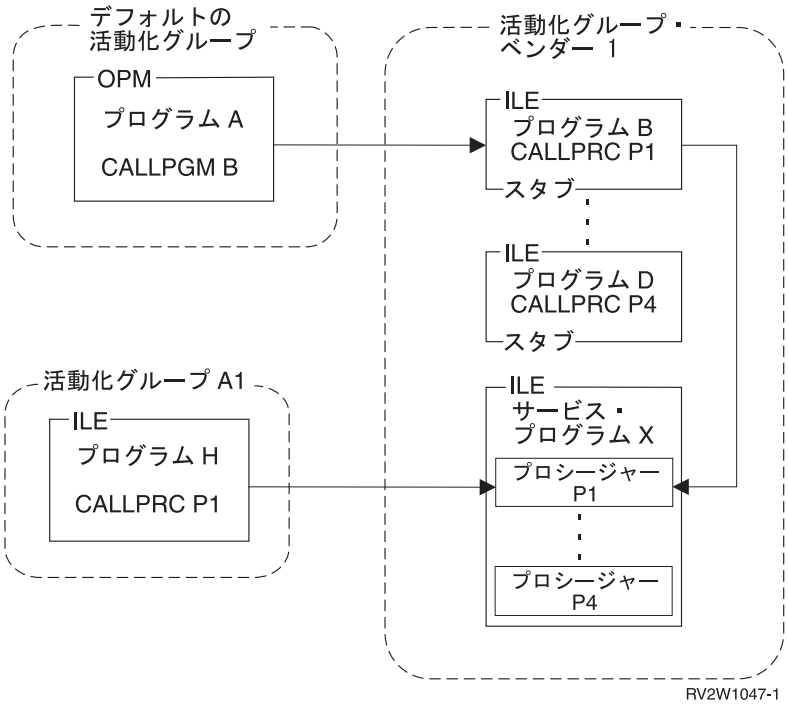

図 44. OPM および ILE API のサポート

プログラム B から D までは、ILE プログラム・スタブです。サービス・プログラム X には各バインド可能 API の実際の処理が含まれています。各プログラム・スタブおよびサービス・プログラムには、同じ活動化グループ名が与えられます。この例では、活動化グループ名 VENDOR1 が選択されています。

活動化グループ VENDOR1 は、必要な場合、システムによって作成されます。OPM プログラム A からの動的プログラム呼び出しは、OPM プログラムからの最初の呼び出し時に、活動化グループを作成します。ILE プログラム H からの静的プロシージャー呼び出しは、ILE プログラム H の活動時に、活動化グループを作成します。いったん活動化グループが存在するようになると、プログラム A またはプログラム H のいずれからも使用することができます。

API の使用は、ILE プロシージャー (この例ではプロシージャー P1) に指定しなければなりません。このプロシージャーは、プロシージャー呼び出しにより直接に呼び出すことも動的プログラム呼び出しにより間接に呼び出すこともできます。特定の呼び出しスタック構造に依存する例外メッセージの送信などの機能は使用できま

せん。プログラム・スタブまたはインプリメント中のプロシージャーから正常に戻った場合、活動化グループは、後からの使用のためにジョブ内に残されます。各呼び出しごとに制御境界がプログラム・スタブまたはインプリメント中のプロシージャーのいずれに設定されているかを判別して、API プロシージャーをインプリメントすることができます。呼び出しが OPM プログラムによるものかまたは ILE プログラムによるものかに関係なく、HLL 終了 verb によって活動化グループは削除されます。

第 8 章 ストレージ管理

オペレーティング・システムは、ILE 高水準言語に対するストレージ・サポートを提供します。このストレージ・サポートにより、各言語の実行時環境に固有のストレージ管理機能が不要になります。このサポートによって、各高水準言語の種々のストレージ管理機能やメカニズムとの間の互換性が確保されます。

オペレーティング・システムは、実行時にプログラムおよびプロシージャーによって使用される自動ストレージ、静的ストレージ、および動的ストレージを提供します。 自動ストレージおよび静的ストレージは、オペレーティング・システムによって管理されます。つまり、自動ストレージおよび静的ストレージの必要性は、コンパイル時にプログラム変数の宣言から分かります。動的ストレージは、プログラムまたはプロシージャーによって管理されます。動的ストレージの必要性は実行時にのみ分かります。

プログラムの活動化が行われると、プログラム変数の静的ストレージが割り振られ初期設定されます。

プログラムまたはプロシージャーが実行を開始すると、自動ストレージが割り振られます。プログラムまたはプロシージャーが呼び出しスタックに追加されると、そのプログラムまたはプロシージャーの変数に対して、自動ストレージ・スタックが拡張されます。

プログラムまたはプロシージャーの実行時に、動的ストレージがプログラムの制御のもとで割り振られます。このストレージは、追加のストレージが必要になると拡張されます。 ユーザーは動的ストレージを制御できます。本章の残りの部分では、動的ストレージおよびその制御方法について説明します。

単一レベル・ストア・ヒープ

ヒープは、動的ストレージの割り振りに使用されるストレージです。アプリケーションに必要な動的ストレージの量は、ヒープを使用するプログラムやプロシージャーにより処理されるデータによって異なります。オペレーティング・システムは、動的に作成され、廃棄される複数の単一レベル・ストア・ヒープを使用することができます。ALCHSS 命令は常に単一レベル・ストアを使用します。言語によっては、動的ストレージ用のテラスペースの使用もサポートします。

ヒープの特性

各ヒープは以下の特性を持ちます。

- システムは活動化グループ内の各ヒープに対して固有のヒープ ID を割り当てます。

  デフォルトのヒープのヒープ ID は常にゼロです。

プログラムまたはプロシージャーによって呼び出されるストレージ管理バインド可能 API は、処理対象であるヒープの識別にこのヒープ ID を使用します。バインド可能 API は、ヒープを所有している活動化グループ内で実行しなければなりません。

- ヒープを作成する活動化グループは、そのヒープを所有します。

  活動化グループがヒープを所有するので、ヒープの存続期間は、ヒープを所有している活動化グループの存続期間を超えることはありません。ヒープ ID は、それを所有している活動化グループ内でのみ意味があり、固有です。
- ヒープのサイズは、割り振り要求を満たすために動的に拡張されます。

  ヒープの最大サイズは 4 ギガバイトから 512K バイトを引いた値です。 これは、割り振りの (1 時点の) 合計数が 128 000 を超えない場合の最大のヒープ・サイズです。
- ヒープからの単一の割り振りの最大サイズは、16 メガバイトから 64K バイトを引いた値に制限されます。

デフォルトのヒープ

単一レベル・ストアを使用している活動化グループ内のデフォルトのヒープからの動的ストレージに対する最初の要求によって、ストレージの割り振りを行うためのデフォルトのヒープが作成されます。動的ストレージに対するその後の要求を満たすストレージがヒープにない場合、システムはそのヒープを拡張し、追加のストレージを割り振ります。

割り振られた動的ストレージは、明示的に解放されるか、またはシステムがそのヒープを廃棄するまで、割り振られたまま存続します。デフォルトのヒープは、ヒープを所有している活動化グループが終了した場合にのみ廃棄されます。

同じ活動化グループ内のプログラムは、動的ストレージがデフォルトのヒープから割り振られている場合には、そのストレージを自動的に共用します。ただし、活動化グループ内の特定のプログラムやプロシージャーによって使用される動的ストレージを分離することができます。これを行うには、1 つ以上のヒープを作成します。

ユーザー作成ヒープ

ILE バインド可能 API を使用して、1 つ以上のヒープの明示的な作成や廃棄を行うことができます。これによって、ヒープおよびそれらのヒープから割り振られる動的ストレージを管理することができます。

たとえば、システムは、活動化グループ内のプログラム用のユーザー作成ヒープに割り振られた動的ストレージの共用が可能であり、また共用しないこともできます。動的ストレージの共用は、プログラムにより参照されるヒープ ID によって決まります。動的ストレージの自動的な共用を回避するために、1 つ以上のヒープを使用することができます。このようにして、データの論理グループを分離することができます。1 つ以上のユーザー作成ヒープを使用する他の理由は、次のとおりです。

- 特定のストレージのオブジェクトをグループ化して、一度だけの要求を満たすことができます。要求を満たした後は、ヒープ廃棄 (CEEDSHP) バインド可能 API を一度呼び出すことにより、割り振られていた動的ストレージを解放することが

できます。この操作は、動的ストレージを解放し、ヒープを廃棄します。これにより、動的ストレージは他の要求を満たすために使用できるようになります。

- ヒープ・マーク付け (CEEMKHP) およびヒープ解放 (CEERLHP) のバインド可能 API を使用して、割り振られた複数の動的ストレージを一度に解放することができます。CEEMKHP バインド可能 API によって、ヒープをマークできます。ヒープのマーク以降に行われた割り振りのグループの解放が可能になった時点で、CEERLHP バインド可能 API を使用します。マーク機能と解放機能を使用することによって、ヒープをそのままにして、ヒープから割り振られていた動的ストレージを解放することができます。このように既存のヒープを再使用して、動的ストレージの要求を満たすことによって、ヒープの作成に伴うシステム・オーバーヘッドを回避することができます。
- ストレージの要件が、デフォルトのヒープを定義しているストレージ属性と一致しないことがあります。たとえば、デフォルトのヒープの初期サイズは 4K バイトです。しかし、合計すると 4K バイトを超える多くの動的ストレージの割り振りが必要だとします。4K バイトを超える初期サイズでヒープを作成することができます。これにより、ヒープの暗黙的な拡張や、そのヒープの拡張部分へのアクセスで生じるシステム・オーバーヘッドを減らすことができます。同様に、4K バイトより大きいヒープの拡張を指定することもできます。ヒープ・サイズの定義については『ヒープ割り振りのストラテジー』およびヒープ属性の説明を参照してください。

何らかの理由により、デフォルトのヒープではなく複数のヒープを使用したい場合があります。ストレージ管理バインド可能 API を使用して、ユーザー作成のヒープおよびそれらのヒープに割り振られた動的ストレージの両方を管理することができます。 IBM では、ストレージ管理バインド可能 API について説明しているオンライン情報を提供しています。iSeries Information Center の**プログラミング**・カテゴリーの中の *API* セクションを参照してください。

単一ヒープのサポート

組み込みの複数ヒープ・ストレージ・サポートがない言語では、デフォルトである単一レベル・ストア・ヒープを使用します。デフォルトのヒープに、ヒープ廃棄 (CEEDSHP)、ヒープ・マーク付け (CEEMKHP)、またはヒープ解放 (CEERLHP) の各バインド可能 API を使用することはできません。デフォルトのヒープによって割り振られる動的ストレージは、明示的な解放操作を使用するか、またはそれを所有する活動化グループを終了することによって解放することができます。

デフォルトのヒープの使用に関するこれらの制約は、割り振られた動的ストレージが、混合言語アプリケーションで不用意に解放されるのを防止するのに役立ちます。ヒープ解放およびヒープ廃棄の操作は、潜在的に異なるストレージ・サポートで既存のコードを再使用する大きなアプリケーションの場合、危険であると考えられます。デフォルトのヒープに対して有効なヒープ解放操作を使用すべきでないことを覚えておいてください。ヒープ解放操作がデフォルトのヒープに対し有効であった場合、マーク機能を個別に正しく使用したアプリケーションの複数部分が、一緒に使用すると失敗することがあります。

ヒープ割り振りのストラテジー

デフォルトのヒープに関連する属性は、デフォルト割り振りのストラテジーを介してシステムによって定義されます。この割り振りのストラテジーは、4K バイトのヒ

ープ作成サイズおよび 4K バイトの拡張部分サイズなどの属性を定義します。このデフォルト割り振りストラテジーを変更することはできません。

ただし、ヒープ作成 (CEECRHP) バインド可能 API を介して明示的に作成したヒープを制御することはできます。ヒープ割り振りストラテジー定義 (CEE4DAS) バインド可能 API を介して、明示的に作成したヒープの割り振りストラテジーを定義することもできます。この場合、ヒープを明示的に作成すると、定義した割り振りストラテジーによってヒープ属性が提供されます。このようにして、1 つ以上の明示的に作成したヒープごとに個別の割り振りストラテジーを定義することができます。

割り振りストラテジーを定義せずに、CEECRHP バインド可能 API を使用することができます。この場合、ヒープは _CEE4ALC 割り振りストラテジー・タイプの属性によって定義されます。_CEE4ALC 割り振りストラテジー・タイプは、4K バイトのヒープ作成サイズ、および 4K バイトの拡張部分サイズを指定します。_CEE4ALC 割り振りストラテジー・タイプには以下の属性が含まれています。

```
Max_Sngl_Alloc = 16MB - 64K  /* 単一の割り振りの最大サイズ */
Min_Bdy        = 16          /* 任意の割り振りの最小境界合わせ */
Crt_Size       = 4K          /* ヒープの初期作成サイズ */
Ext_Size       = 4K          /* ヒープのエクステンション・サイズ */
Alloc_Strat    = 0           /* 割り振りストラテジーの 1 つの選択 */
No_Mark        = 1           /* グループ割り振り解除の選択 */
Blk_Xfer       = 0           /* ヒープのブロック転送の 1 つの選択 */
PAG            = 0           /* PAG でのヒープ作成の 1 つの選択 */
Alloc_Init     = 0           /* 割り振り初期化の 1 つの選択 */
Init_Value     = 0x00        /* 初期化値 */
```

上記の属性は、_CEE4ALC 割り振りストラテジー・タイプの構造を説明しています。 IBM では、すべての _CEE4ALC 割り振りストラテジー属性に関して説明しているオンライン情報を提供しています。iSeries Information Center の**プログラミング**・カテゴリーにある *API* セクションを参照してください。

単一レベル・ストア・ヒープのインターフェース

バインド可能 API はすべてのヒープ操作に関して提供されています。アプリケーションは、バインド可能 API または言語の組み込み関数、またはその両方を使用して作成することができます。

バインド可能 API は以下のカテゴリーに分けることができます。

- 基本ヒープ操作。これらの操作は、デフォルトのヒープとユーザー作成ヒープに使用することができます。
  - ストレージ解放 (CEEFRST) バインド可能 API は、ヒープ・ストレージの前の割り振りを 1 つ解放します。
  - ヒープ・ストレージ入手 (CEEGTST) バインド可能 API は、ヒープ内のストレージを割り振ります。
  - ストレージ再割り振り (CEECZST) バインド可能 API は、前に割り振られたストレージのサイズを変更します。
- 拡張ヒープ操作。 これらの操作は、ユーザー作成ヒープに使用することができます。
  - ヒープ作成 (CEECRHP) バインド可能 API は新しいヒープを作成します。
  - ヒープ廃棄 (CEEDSHP) バインド可能 API は既存のヒープを廃棄します。

ヒープ・マーク付け (CEEMKHP) バインド可能 API は、CEERLHP バインド可能 API によって解放されるヒープ・ストレージの識別に使用されるトークンを戻します。
ヒープ解放 (CEERLHP) バインド可能 API は、マークが指定された以降に、ヒープ内に割り振られたすべてのストレージを解放します。

- ヒープ割り振りストラテジー
  ヒープ割り振りストラテジー定義 (CEE4DAS) バインド可能 API は、CEECRHP バインド可能 API で作成されたヒープの属性を判別する割り振りストラテジーを定義します。

IBM では、ストレージ管理バインド可能 API に関するオンライン情報を提供しています。 iSeries Information Center の**プログラミング**・カテゴリーにある *API* セクションを参照してください。

ヒープ・サポート

デフォルトで、malloc、calloc、realloc、および new によって提供される動的ストレージは、活動化グループ内のルート・プログラムのストレージ・モデルと同じタイプのストレージになります。ただし、単一レベル・ストレージ・モデルの使用中に、TERASPACE(*YES *TSIFC) コンパイラー・オプションが指定されていれば、テラスペース・ストレージがこれらのインターフェースによって提供されます。同様に、単一レベル・ストア・ストレージ・モデル・プログラムは、_C_TS_malloc、_C_TS_free、_C_TS_realloc および _C_TS_calloc のように、テラスペースを使用して作業するために、バインド可能 API を明示的に使用することができます。

テラスペース・ストレージを使用する方法の詳細については 57 ページの『第 4 章 テラスペースおよび単一レベル・ストア』を参照してください。

CEExxxx ストレージ管理バインド可能 API と ILE C malloc()、calloc()、realloc()、および free() 関数の両方を使用する場合には、以下の規則が適用されます。

- C 関数の malloc()、calloc()、および realloc() を介して割り振られる動的ストレージは、CEEFRST および CEECZST バインド可能 API によって解放または再割り振りを行うことはできません。
- CEEGTST バインド可能 API によって割り振られた動的ストレージは、free() 関数によって解放できます。
- CEEGTST バインド可能 API によって最初に割り振られた動的ストレージは、realloc() 関数によって再割り振りを行うことができます。

COBOL などの他の言語には、ヒープ・ストレージ・モデルはありません。これらの言語は、動的ストレージのバインド可能 API を介して ILE 動的ストレージ・モデルにアクセスできます。

RPG には、デフォルトのヒープにアクセスするための命令コード ALLOC、REALLOC、および DEALLOC と、組み込み関数 %ALLOC および %REALLOC が用意されています。 RPG サポートでは、CEEGTST、CEECZST、および CEEFRST バインド可能 API が使用されます。

第 9 章 例外および条件管理

本章では、例外処理および条件処理について、より詳細に説明しています。本章を読む前に、概要を示す 45 ページの『エラー処理』をお読みください。

OS/400 の例外メッセージ体系は、例外処理と条件処理の両方をインプリメントするために使用されます。例外処理と条件処理が相互作用する場合があります。たとえば、ユーザー作成条件ハンドラー登録 (CEEHDLR) バインド可能 API で登録されている ILE 条件ハンドラーは、プログラム・メッセージ送信 (QMHSNDPM) API で送信される例外メッセージの処理に使用されます。このような相互作用を本章で説明しています。**例外ハンドラー**という用語は、本章では、OS/400 例外ハンドラーまたは ILE 条件ハンドラーのいずれかの意味で使用しています。

処理カーソルおよび再開カーソル

例外の処理に、システムは処理カーソルおよび再開カーソルと呼ばれる 2 つのポインターを使用します。これらのポインターは、例外処理の進行を追跡します。 特定の拡張エラー処理シナリオにおける処理カーソルおよび再開カーソルの使用法の理解が必要です。これらの概念は、以後のトピックで追加のエラー処理機能を説明するのに使用されます。

処理カーソルは、現行の例外ハンドラーを追跡するポインターです。システムは、使用可能な例外ハンドラーを検索すると、各呼び出しスタック項目ごとに定義されている例外ハンドラー・リストの次のハンドラーに処理カーソルを移します。 このリストには、以下のハンドラーが入っています。

- 直接モニター・ハンドラー
- ILE 条件ハンドラー
- HLL 固有ハンドラー

処理カーソルは、例外が処理されるまで、例外ハンドラー・リストのより高い優先順位のハンドラーからより低い優先順位のハンドラーに移動します。1 つの呼び出しスタック項目に定義された例外ハンドラーのいずれによっても例外が処理されない場合、処理カーソルは、前の呼び出しスタック項目に対する最初の (最高の優先順位の) ハンドラーに移動します。

再開カーソルは、例外ハンドラーが例外を処理した後に処理を再開できる現在の位置を追跡するポインターです。通常、システムは再開カーソルを、例外が起こった命令の次の命令にセットします。例外を引き起こしたプロシージャーより上位の呼び出しスタック項目の場合、再開点は、現在、プロシージャーまたはプログラムを中断しているプロシージャー呼び出しまたはプログラム呼び出しの直後になります。再開カーソルを前の再開点に移動するには、再開カーソル移動 (CEEMRCR) バインド可能 API を使用します。

136 ページの図 45 は処理カーソルおよび再開カーソルの例を示しています。

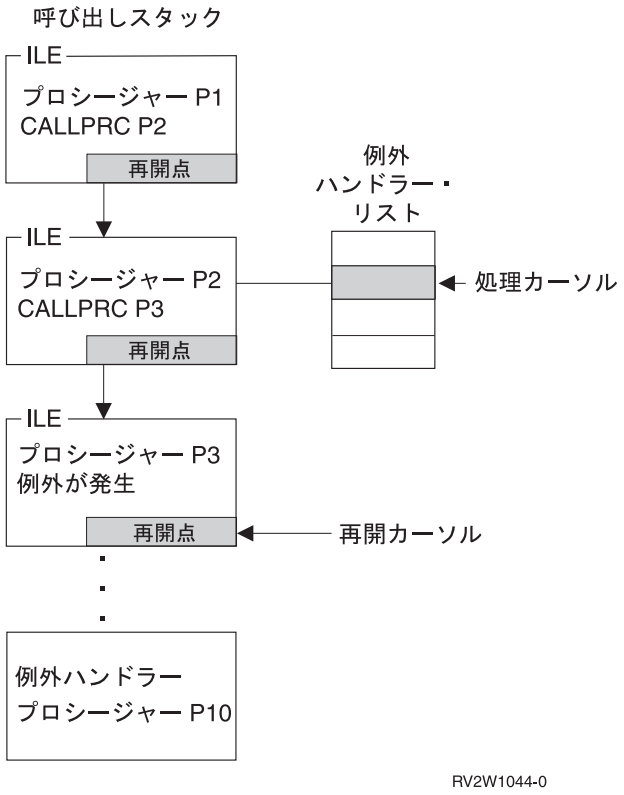

図 45. 処理カーソルおよび再開カーソルの例

処理カーソルは、プロシージャー P2 に関する例外ハンドラー優先順位リストに定義されている 2 番目の例外ハンドラーを現在指しています。ハンドラー・プロシージャー P10 が現在、システムによって呼び出されています。プロシージャー P10 が例外を処理して戻ると、制御は、プロシージャー P3 に定義されている現在の再開カーソル位置に移ります。この例では、プロシージャー P3 が、プロシージャー P2 に例外をパーコレートしたことを想定しています。

例外ハンドラー・プロシージャー P10 は、再開カーソルを再開カーソル移動 (CEEMRCR) バインド可能 API で変更することができます。この API には 2 つのオプションがあります。例外ハンドラーは再開カーソルを以下のいずれかに変更することができます。

- 処理カーソルを含む呼び出しスタック項目
- 処理カーソルの前の呼び出しスタック項目

図 45 では、再開カーソルをプロシージャー P2 または P1 のいずれかに変更することができます。再開カーソルが変更され、例外が処理済みとしてマークされた後で、例外ハンドラーから正常に戻ると、新しい再開点に制御を戻します。

例外ハンドラーのアクション

例外ハンドラーがシステムによって呼び出された場合、例外を処理するいくつかのアクションを行うことができます。たとえば、ILE C 拡張機能は、制御アクション、ブランチ点ハンドラー、およびメッセージ ID によるモニターをサポートします。ここで説明する考えられるアクションは、以下のいずれかのタイプのハンドラーに関連しています。

- 直接モニター・ハンドラー
- ILE 条件ハンドラー
- HLL 固有ハンドラー

処理を再開する方法

処理の続行が可能であると判断した場合には、現在の再開カーソル位置から再開することができます。処理を再開する前に、例外メッセージの処理が終了していることを示すために、例外メッセージの変更が必要です。例外ハンドラーのタイプには、例外メッセージの処理が行われたことを示すために、例外メッセージを明示的に変更しなければならないタイプのものがあります。その他のハンドラーのタイプに関しては、そのハンドラーの呼び出し前に、システムが例外メッセージを変更できます。

直接モニター・ハンドラーの場合、例外メッセージに対して行うアクションを指定することができます。このアクションには、ハンドラーを呼び出すことや、ハンドラーを呼び出す前に例外を処理することや、または例外を処理してプログラムを再開することがあります。アクションがハンドラーを呼び出すことだけである場合、例外メッセージ変更 (QMHCHGEM) API またはバインド可能 API CEE4HC (条件処理) を使用して例外を処理することができます。直接モニター・ハンドラー内の再開点は、再開カーソル移動 (CEEMRCR) バインド可能 API を用いて変更することができます。このような変更を行った後、例外ハンドラーから戻ることによって処理を続行することができます。

ILE 条件ハンドラーの場合、戻りコード値を設定し、システムに戻ることによって処理を続行することができます。 IBM では、ユーザー作成条件ハンドラー登録 (CEEHDLR) バインド可能 API 用の実際の戻りコード値に関して説明しているオンライン情報を提供しています。iSeries Information Center の**プログラミング**・カテゴリーにある *API* セクションを参照してください。

HLL 固有ハンドラーの場合、使用するハンドラーの呼び出し前に、例外メッセージの処理が終了したことを示すために例外メッセージが変更されます。HLL 固有ハンドラーから再開カーソルを変更できるか否かについては、該当の ILE HLL の「プログラマーの手引き」を参照してください。

メッセージをパーコレートする方法

例外メッセージが使用中のハンドラーによって認識されないと判断した場合には、使用可能な次のハンドラーへ例外メッセージをパーコレートすることができます。パーコレーションが起きるためには、例外メッセージは処理済みのメッセージと見なされてはなりません。同じ呼び出しスタック項目または前の呼び出しスタック項目内の他の例外ハンドラーに、例外メッセージを処理する機会が与えられます。例外メッセージをパーコレートする手法は、例外ハンドラーのタイプによって異なります。

直接モニター・ハンドラーの場合、例外メッセージの処理が終了したことを示すためのメッセージの変更は行ってはなりません。例外ハンドラーからの通常の戻りによって、システムはメッセージをパーコレートします。呼び出しスタック項目の例外ハンドラー・リスト内の次の例外ハンドラーへ、メッセージがパーコレートされ

ます。使用中のハンドラーが例外ハンドラー・リストの最後にある場合、前の呼び出しスタック項目内の最初の例外ハンドラーへメッセージがパーコレートされます。

ILE 条件ハンドラーの場合、戻りコード値を設定し、システムに戻ることによって、パーコレート・アクションを伝えます。 IBM では、ユーザー作成条件ハンドラー登録 (CEEHDLR) バインド可能 API 用の実際の戻りコード値に関して説明しているオンライン情報を提供しています。iSeries Information Center の**プログラミング**・カテゴリーにある *API* セクションを参照してください。

HLL 固有ハンドラーの場合、例外メッセージをパーコレートできないことがあります。メッセージをパーコレートできるかどうかは、使用中の HLL が、使用するハンドラーの呼び出し前に、メッセージを処理済みとしてマークするかどうかによって決まります。 HLL 固有ハンドラーを宣言しない場合、HLL は未処理の例外メッセージをパーコレートすることができます。HLL 固有ハンドラーが処理できる例外メッセージについては、該当の ILE HLL の「解説書」を参照してください。

メッセージをプロモートする方法

限定されたある状況のもとで、例外メッセージを別のメッセージへ変更することが選択できます。このアクションは、元の例外メッセージを処理済みとしてマークし、新しい例外メッセージを出して例外処理を再開します。このアクションは、直接モニター・ハンドラーと ILE 条件ハンドラーからのみ実行することができます。

直接モニター・ハンドラーの場合、メッセージ・プロモート (QMHPRMM) API を使用してメッセージをプロモートしてください。システムがプロモートできるのは、状況メッセージ・タイプとエスケープ・メッセージ・タイプだけです。この API を使用して、例外処理を続行するための処理カーソルの位置をある程度制御することができます。iSeries Information Center の**プログラミング**・カテゴリーにある *API* セクションを参照してください。

ILE 条件ハンドラーの場合、戻りコード値をセットして、システムに戻ることによってプロモート・アクションを伝えます。 IBM では、ユーザー作成条件ハンドラー登録 (CEEHDLR) バインド可能 API 用の実際の戻りコード値に関して説明しているオンライン情報を提供しています。iSeries Information Center の**プログラミング**・カテゴリーにある *API* セクションを参照してください。

未処理例外に関するデフォルト・アクション

例外メッセージが制御境界にパーコレートされると、システムはデフォルト・アクションをとります。例外が通知メッセージの場合、システムは、デフォルト応答を送信し、例外を処理し、そして通知メッセージの送信側にプロセスを継続させます。例外が状況メッセージの場合には、システムは、例外を処理し、状況メッセージの送信側にプロセスを継続させます。例外がエスケープ・メッセージの場合には、システムは、エスケープ・メッセージを処理し、機能チェック・メッセージを、再開カーソルが現在ある位置に送信します。未処理例外が機能チェックである場合には、制御境界までのスタック上のすべての項目が取り消され、CEE9901 エスケープ・メッセージが次の優先順位をもつスタック項目に送信されます。

表 9 は、制御境界で例外が未処理の場合にシステムが行うデフォルトの応答を示しています。

表 9. 未処理の例外に対するデフォルトの応答

メッセージ・タイプ	条件の重大度	条件シグナル (CEESGL) バインド可能 API によって生じる条件	他のソースからの例外
状況	0 (情報メッセージ)	未処理条件を戻します。	メッセージをログに記録しないで再開します。
状況	1 (警告)	未処理条件を戻します。	メッセージをログに記録しないで再開します。
通知	0 (情報メッセージ)	適用外	通知メッセージをログに記録し、デフォルト応答を送信します。
通知	1 (警告)	適用外	通知メッセージをログに記録し、デフォルト応答を送信します。
エスケープ	2 (エラー)	未処理条件を戻します。	エスケープ・メッセージをログに記録し、現行の再開点の呼び出しスタック項目に機能チェック・メッセージを送信します。
エスケープ	3 (重大エラー)	未処理条件を戻します。	エスケープ・メッセージをログに記録し、現行の再開点の呼び出しスタック項目に機能チェック・メッセージを送信します。
エスケープ	4 (クリティカル ILE エラー)	エスケープ・メッセージをログに記録し、現行の再開点の呼び出しスタック項目に機能チェック・メッセージを送信します。	エスケープ・メッセージをログに記録し、現行の再開点の呼び出しスタック項目に機能チェック・メッセージを送信します。
機能チェック	4 (クリティカル ILE エラー)	適用外	アプリケーションを終了し、制御境界の呼び出し元にメッセージ CEE9901 を送信します。

注: アプリケーションが未処理の機能チェックによって終了した場合、制御境界が活動化グループ内の最も古い呼び出しスタック項目であると、その活動化グループは削除されます。

ネストされた例外

ネストされた例外は、別の例外の処理中に発生した例外です。この場合、最初の例外の処理は一時的に中断されます。システムは、処理カーソルおよび再開カーソルの位置などのすべての関連情報を保管します。例外処理は、最後に起きた例外に対して再開されます。処理カーソルおよび再開カーソルの新しい位置は、システムによって設定されます。新しい例外が適切に処理された後、元の例外の処理活動が正常に再開されます。

ネストされた例外が発生した場合、以下の項目はいずれも依然として呼び出しスタック上に存在しています。

- 元の例外に関連する呼び出しスタック項目
- 元の例外ハンドラーに関連する呼び出しスタック項目

例外処理ループの可能性を低くするために、システムは、ネストされた例外のパーコレーションを元の例外ハンドラーの呼び出しスタック項目で停止します。次に、システムはネストされた例外を機能チェック・メッセージにプロモートし、その機能チェック・メッセージを同じ呼び出しスタック項目にパーコレートします。ネストされた例外または機能チェック・メッセージを処理しない場合、システムは、異常終了 (CEE4ABN) バインド可能 API を呼び出すことによってアプリケーションを終了します。この場合には、メッセージ CEE9901 が制御境界の呼び出し元に送信されます。

ネストされた例外の処理中に再開カーソルを移動すると、元の例外を暗黙的に変更することができます。このためには、以下のステップを行います。

1. 元の例外を引き起こした呼び出しスタック項目より前の呼び出しスタック項目に再開カーソルを移動します。
2. 使用中のハンドラーから戻ることによって、処理を再開します。

条件処理

ILE 条件は、システムとは独立した方法で表される OS/400 例外メッセージです。ILE 条件を示すために ILE 条件トークンが使用されます。**条件処理**とは、言語固有のエラー処理から独立してエラーを処理するための ILE 機能のことを言います。他のシステムでも、これらの機能を実装しています。条件処理を使用すれば、条件処理を実装したシステム間でのアプリケーションのポータビリティーが容易になります。

ILE 条件処理には、以下の機能があります。

- ILE 条件ハンドラーを動的に登録する機能
- ILE 条件をシグナルする機能
- 条件トークン体系
- バインド可能 ILE API に関するオプショナルの条件トークン・フィードバック・コード

これらの機能について以下のトピックで説明します。

条件を表す方法

ILE **条件トークン**は 12 バイトの複合データ・タイプであり、条件の性質を示す構造化フィールドが入っています。性質とは、条件の重大度、関連メッセージ番号、および条件の特定のインスタンスに固有の情報などです。条件トークンは、条件に関するこれらの情報をシステム、メッセージ・サービス、バインド可能 API、およびプロシージャーに伝えるために使用されます。たとえば、すべての ILE バインド可能 API のオプショナルの fc パラメーターに戻される情報は、条件トークンを使用して伝えられます。

例外がオペレーティング・システムまたはハードウェアによって検出されると、対応する条件トークンがシステムによって自動的に作成されます。条件トークンは、条件トークン作成 (CEENCOD) バインド可能 API を使用しても作成することができます。次に、条件シグナル (CEESGL) バインド可能 API を介してトークンを戻すことによって、条件をシステムにシグナルとして伝えることができます。

条件トークンのレイアウト

図 46 は条件トークンのレイアウトを示しています。各フィールドの開始ビット位置が示されています。

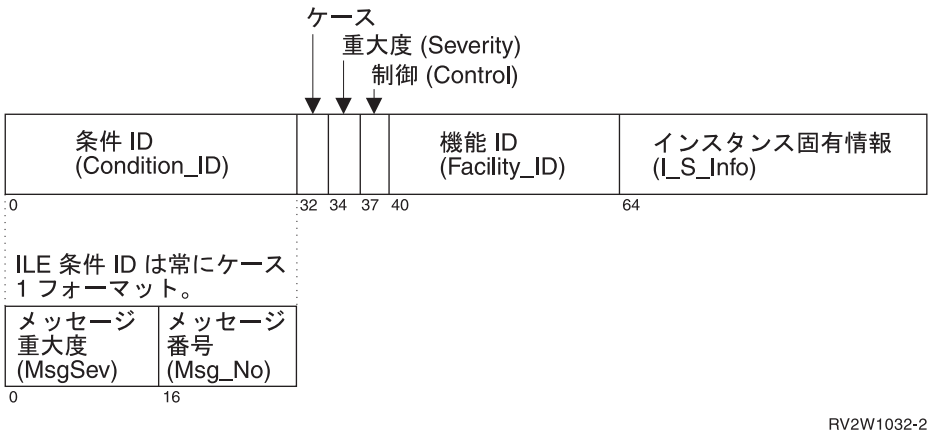

図 46. ILE 条件トークンのレイアウト

各条件トークンには 図 46 に示したコンポーネントが入っています。

条件 ID (Condition_ID)
4 バイトの ID で、Facility_ID と共に、トークンが伝える条件を記述します。ILE バインド可能 API と大部分のアプリケーションは、ケース 1 の条件を生成します。

ケース 2 ビットのフィールドで、トークンの Condition_ID 部分の形式を定義します。ILE 条件は常にケース 1 です。

重大度 (Severity)
3 ビットの 2 進整数で、条件の重大度を示します。Severity フィールドと MsgSev フィールドには同じ情報が入ります。ILE 条件の重大度のリストについては 139 ページの表 9 を参照してください。OS/400 メッセージの重大度については 143 ページの表 11 および 143 ページの表 12 を参照してください。

制御 (Control)
3 ビットのフィールドで、条件処理の様々な様相を記述または制御するフラグが入ります。3 番目のビットは、Facility_ID が IBM によって割り当てられたか否かを示します。

機能 ID (Facility_ID)
3 文字の英数字ストリングで、条件を生成した機能を識別します。Facility_ID は、メッセージの生成がシステムによるものか、または HLL 実行時かを示します。 142 ページの表 10 は ILE で使用される機能 ID をリストしたものです。

インスタンス固有情報 (I_S_Info)
4 バイトのフィールドであり、条件の特定のインスタンスに関連するインスタンス固有情報を示します。このフィールドには、条件トークンに関連するメッセージのインスタンスへの参照キーが含まれます。メッセージ参照キーがゼロの場合、関連メッセージはありません。

メッセージ重大度 (MsgSev)
2 バイトの 2 進整数で、条件の重大度を示します。MsgSev と Severity には同じ情報があります。ILE 条件の重大度のリストについては 139 ページの表 9 を参照してください。OS/400 メッセージの重大度については 143 ページの表 11 および 143 ページの表 12 を参照してください。

メッセージ番号 (Msg_No)
2 バイトの 2 進数で、条件に関連するメッセージを示します。 Facility_ID と Msg_No の組み合わせによって、固有の条件を識別します。

表 10 は、ILE 条件トークンおよび OS/400 メッセージの接頭部で使用される機能 ID を示しています。

表 10. メッセージおよび ILE 条件トークンで使用される機能 ID

機能 ID	機能
CEE	ILE 共通ライブラリー
CPF	OS/400 XPF メッセージ
MCH	OS/400 マシン例外メッセージ

条件トークンのテスト

バインド可能 API から戻された条件トークンを、以下についてテストすることができます。

成功 成功したかどうかをテストするには、最初の 4 バイトがゼロか否かを判別します。最初の 4 バイトがゼロの場合、条件トークンの残りの部分はゼロであり、バインド可能 API の呼び出しが成功したことを示します。

等価トークン
2 つの条件トークンが等価である (すなわち、同じタイプ ではあるが、同じインスタンス ではない条件トークン) か否かを調べるには、各条件トークンの最初の 8 バイトを比較します。これらのバイトは、与えられた条件のすべてのインスタンスに関して同じです。

同等トークン
2 つの条件トークンが同等である (すなわち、1 つの条件の同じインスタンス) か否かを調べるには、各条件トークンの 12 バイトすべてを比較します。最後の 4 バイトは、同じ条件でもインスタンスごとに異なります。

ILE 条件と OS/400 メッセージの関係

メッセージは、ILE で発生するすべての条件に関連しています。条件トークンには、条件に関連するメッセージをメッセージ・ファイルに書き込むために ILE が使用する固有の ID が入っています。

すべての実行時メッセージの形式は、**FFFxxxx** です。

FFF 機能 ID。3 文字の ID で、ILE および ILE 言語のもとで生成されるすべ

てのメッセージによって使用されます。ID および対応する機能のリストについては 142 ページの表 10 を参照してください。

xxxx エラー・メッセージ番号。これは 16 進数で、条件に関連するエラー・メッセージを示します。

表 11 と 表 12 は、ILE 条件の重大度が OS/400 メッセージの重大度にどのようにマップされるかを示しています。

*表 11. ILE 条件の重大度への OS/400 *ESCAPE メッセージの重大度のマッピング*

OS/400 メッセージの重大度から	ILE 条件の重大度へ	OS/400 メッセージの重大度へ
0〜29	2	20
30〜39	3	30
40〜99	4	40

*表 12. ILE 条件の重大度への OS/400 *STATUS と *NOTIFY メッセージの重大度のマッピング*

OS/400 メッセージの重大度から	ILE 条件の重大度へ	OS/400 メッセージの重大度へ
0	0	0
1〜99	1	10

OS/400 メッセージおよびバインド可能 API のフィードバック・コード

バインド可能 API への入力として、フィードバック・コードのコーディング、およびプロシージャー内の戻り (またはフィードバック) コード・チェックとしてのフィードバック・コードの使用を選択することができます。フィードバック・コードは、他のプロシージャーの呼び出しからの戻りの検査に融通性を与えるために用意された条件トークン値です。その後、フィードバック・コードを条件トークンへの入力として使用することができます。 バインド可能 API の呼び出しでフィードバック・コードを指定しない場合に条件が発生すると、バインド可能 API の呼び出し元に例外メッセージが送信されます。

バインド可能 API からのフィードバック情報を受け取るために、アプリケーション内にフィードバック・コード・パラメーターをコーディングした場合、条件が発生すると以下の一連のイベントが発生します。

1. 情報メッセージが API の呼び出し側に送信され、条件に関連するメッセージが伝えられます。
2. 条件が発生したバインド可能 API は、条件に関する条件トークンを作成します。バインド可能 API は、インスタンス固有情報エリアに情報を入れます。 条件トークンのインスタンス固有情報は、情報メッセージのメッセージ参照キーです。このキーは、条件に対処するためにシステムによって使用されます。
3. 検出された条件がクリティカル (重大度が 4) である場合、システムは例外メッセージをバインド可能 API の呼び出し元に送信します。
4. 検出された条件がクリティカルでない (重大度が 4 より小さい) 場合、バインド可能 API を呼び出したルーチンに、条件トークンが戻されます。

5. 条件トークンがアプリケーションに戻された場合、以下のいずれかの処置を行うことができます。
   - 条件トークンを無視して、処理を続行する。
   - 条件シグナル (CEESGL) バインド可能 API を使用して、条件をシグナルとして伝える。
   - メッセージ入手/フォーマット設定/ディスパッチ (CEEMSG) バインド可能 API を使用して表示用メッセージの入手、フォーマット設定、ディスパッチを行う。
   - メッセージ入手 (CEEMGET) バインド可能 API を使用して、メッセージをストレージに保管する。
   - メッセージ・ディスパッチ (CEEMOUT) バインド可能 API を使用して、ユーザー定義のメッセージを指定する宛先にディスパッチする。
   - API の呼び出し元に制御が再び渡ると、情報メッセージは除去され、ジョブ・ログには現れません。

バインド可能 API を呼び出す場合に、フィードバック・コード・パラメーターを省略すると、バインド可能 API はバインド可能 API の呼び出し元に例外メッセージを送信します。

第 10 章 デバッグに関する考慮事項

ソース・デバッガーは、OPM プログラム、ILE プログラム、およびサービス・プログラムのデバッグに使用されます。オリジナル・プログラム・モデル (OPM) プログラムのデバッグには、CL コマンドを引き続き使用することができます。

本章では、ソース・デバッガーに関するいくつかの考慮事項を示します。ソース・デバッガーを使用する方法については、ご使用の ILE 高水準言語 (HLL) に関するオンライン情報および「プログラマーの手引き」を参照してください。特定の作業 (たとえば、モジュールの作成) に使用するコマンドについては、使用している ILE HLL の「プログラマーの手引き」を参照してください。

デバッグ・モード

ソース・デバッガーを使用するには、セッションがデバッグ・モードでなければなりません。**デバッグ・モード**は、通常のシステム機能に加えてプログラム・デバッグ機能が使用できる特殊な環境です。

デバッグ開始 (STRDBG) コマンドを実行すると、そのセッションはデバッグ・モードになります。

デバッグ環境

プログラムは、次の 2 つの環境のどちらにおいてもデバッグできます。

- OPM デバッグ環境。すべての OPM プログラムは、ILE デバッグ環境に明示的に追加された場合を除き、この環境でデバッグすることができます。
- ILE デバッグ環境。すべての ILE プログラムはこの環境でデバッグされます。さらに、OPM プログラムについても、次のすべての基準に該当する場合には、この環境でデバッグすることができます。
  - CL、COBOL、または RPG プログラムである。
  - OPM ソース・デバッグ・データと共にコンパイルされている。
  - その STRDBG コマンドの OPMSRC パラメーターが *YES に設定されている。

ILE デバッグ環境は、ソース・レベルのデバッグ・サポートを提供します。デバッグの機能は、ステートメント、ソース、またはコードのリスト・ビューから直接得られます。

いったん OPM プログラムが ILE デバッグ環境に入ると、システムは、ILE および OPM プログラムのシームレスであるデバッグを同じユーザー・インターフェースを介して提供します。 ILE デバッグ環境において、OPM プログラムでソース・デバッガーを使用する方法については、OPM 言語として使用している同等の ILE 高水準言語 (HLL) (CL、COBOL、または RPG) のオンライン・ヘルプまたは「プログラマーの手引き」を参照してください。

デバッグ・モードへのプログラムの追加

プログラムは、デバッグに先立ってデバッグ・モードに加えなければなりません。OPM プログラム、ILE プログラム、および ILE サービス・プログラムを同時にデバッグ・モードにすることができます。 OPM デバッグ環境では、一度に 20 の OPM プログラムをデバッグ・モードにすることができます。ILE デバッグ環境において、同時にデバッグ・モードにすることができる ILE プログラム、サービス・プログラム、および OPM プログラムの数には制限がありません。ただし、一度にサポートされるデバッグ・データの最大量は、モジュールごとに 16MB です。

プログラムまたはサービス・プログラムをデバッグ・モードに加えるには、それに対する *CHANGE 権限が必要です。プログラムまたはサービス・プログラムは、呼び出しスタックで停止している時点で、デバッグ・モードに加えることができます。

ILE プログラムおよび ILE サービス・プログラムは、ソース・デバッガーによって、一度に 1 モジュールずつアクセスされます。ILE プログラムまたは ILE サービス・プログラムをデバッグしている時点で、他のプログラムまたはサービス・プログラムのモジュールをデバッグする必要が生じることがあります。 2 番目のプログラムまたはサービス・プログラムのモジュールをデバッグする前に、この 2 番目のプログラムをデバッグ・モードに加えなければなりません。

デバッグ・モードを終了すると、すべてのプログラムがデバッグ・モードから除去されます。

プログラム識別情報と最適化がデバッグに与える影響

モジュールが識別可能であるかどうか、および完全に最適化されているかどうかは、デバッグに影響を与えます。

モジュールの**プログラム識別情報**とは、モジュールとともに保管できる、再コンパイルなしで変更可能なデータを指します。**最適化**は、同じ出力を生成するために必要なシステム・リソースの量を削減する最も効率的な処理方法をシステムが探すプロセスです。

プログラム識別情報

モジュールのプログラム識別情報は 2 つのタイプのデータから構成されます。

デバッグ・データ
: *DBGDTA 値によって示されます。このデータはモジュールをデバッグするのに必要です。

作成データ
: *CRTDTA 値によって示されます。このデータはコードを機械命令に変換するのに必要です。モジュールの最適化レベルを変更するには、モジュールにこのデータがなければなりません。

モジュールをコンパイルした後は、これらのデータの除去だけを行うことができます。モジュールの変更 (CHGMOD) コマンドを使用して、モジュールからどちらかのタイプまたは両方のタイプのデータを除去することができます。すべてのプログラム識別情報の除去によって、モジュールのサイズを (圧縮により) 最小限にするこ

とができます。これらのデータを除去した後は、モジュールを再コンパイルしてデータを置き換えない限り、モジュールを変更することはできません。再コンパイルするには、ソース・コードに対する権限が必要です。

最適化レベル

一般的に、モジュールに作成データがある場合、レベルを変更して、そのレベルでソース・コードを最適化し、システムで実行することができます。処理のショートカットがマシン・コードに変換され、モジュール内のプロシージャーのより効率的な実行が可能になります。最適化レベルが高くなるほど、モジュール内のプロシージャーの実行はより効率的になります。

ただし、高い最適化を使用すると、デバッグの過程で、変数の変更ができず、変数の実際の値を表示できない可能性があります。デバッグ時は、最適化レベルを 10 (*NONE) に設定してください。これによって、モジュール内のプロシージャーのパフォーマンスは最低のレベルになりますが、変数を正確に表示することや、変更することができます。デバッグを完了した後で、最適化レベルを 30 (*FULL) または 40 に設定します。これによって、モジュール内のプロシージャーのパフォーマンス・レベルは最高になります。

デバッグ・データの作成および除去

デバッグ・データは各モジュールとともに保管され、モジュールの作成時に生成されます。モジュール内のデバッグ・データなしで作成されたプロシージャーをデバッグするには、そのモジュールをデバッグ・データとともに再作成し、その後、そのモジュールを ILE プログラムまたはサービス・プログラムに再バインドしなければなりません。デバッグ・データをすでに持っている、プログラムまたはサービス・プログラム内の他のすべてのモジュールは再コンパイルする必要はありません。

デバッグ・データをモジュールから除去するには、デバッグ・データなしでモジュールを再作成するか、またはモジュールの変更 (CHGMOD) コマンドを使用します。

モジュールのビュー

使用可能なデバッグ・データのレベルは、ILE プログラムまたは ILE サービス・プログラム内の各モジュールごとに変更することができます。各モジュールを個別にコンパイルするので、異なるコンパイラーおよびオプションを使用して生成される可能性があります。これらのデバッグ・データ・レベルによって、コンパイラーが生成する**ビュー**およびソース・デバッガーが表示するビューが決まります。次の値が可能です。

***NONE**

デバッグ・ビューは生成されません。

***STMT**

デバッガーによってソースは表示されませんが、コンパイラー・リストにリストされたプロシージャー名およびステートメント番号を使用して停止点を追加することができます。このビューで保管されるデバッグ・データは、デバッグに必要な最少量のデータです。

***SOURCE**
モジュールのコンパイルに使用されたソース・ファイルがシステムに存在している場合には、ソース・デバッガーはそのソースを表示します。

***LIST** リスト・ビューがモジュールとともに生成され保管されます。これによって、モジュールの作成に使用されたソース・ファイルがシステムにない場合でも、ソース・デバッガーはソースを表示することができます。このビューは、プログラムを変更する場合、バックアップ・コピーとして役立ちます。ただし、デバッグ・データの量は、他のファイルがリストに拡張される場合は特に、多くなる可能性があります。組み込みが展開されるかどうかは、モジュールが作成されたときに使用されたコンパイラー・オプションによって決まります。展開することのできるファイルには、DDS ファイル、および組み込みファイル (ILE C 組み込みファイル、ILE RPG /COPY ファイル、および ILE COBOL COPY ファイルなど) が含まれます。

***ALL** すべてのデバッグ・ビューが生成されます。リスト・ビューに関しては、デバッグ・データの量がきわめて多くなる可能性があります。

ILE RPG には、ソース・ビューとコピー・ビューを生成するデバッグ・オプションの *COPY もあります。コピー・ビューは、含まれているすべての /COPY ソース・メンバーを持つデバッグ・ビューです。

ジョブ間のデバッグ

ジョブまたはバッチ・ジョブで実行されるプログラムをデバッグするために、別のジョブを使用したい場合があります。これは、デバッガー・パネルの干渉なしにプログラムの機能を調べたい場合に、特に有用です。たとえば、アプリケーションが表示するパネルやウィンドウが、ステップ内のデバッガー・パネルやや停止点で、オーバーレイしたり、オーバーレイされたりすることがあります。この問題は、サービス・ジョブを開始させ、デバッグ中ではないジョブにデバッガーを開始させることにより、避けることができます。この詳細については、「CL プログラミング、SD88-5038)」 のテストに関する付録を参照してください。

OPM および ILE デバッガー・サポート

OPM および ILE デバッガー・サポートによって、OPM プログラムのソース・レベルのデバッグを、ILE デバッガー API を介して行うことができます。 ILE デバッガー API については、iSeries Information Center の**プログラミング**・カテゴリーの中の *API* セクションを参照してください。 OPM および ILE デバッガー・サポートによって、ILE および OPM プログラムのシームレスであるデバッグを同じユーザー・インターフェースを介して行うことができます。このサポートを使用するには、OPM プログラムを RPG、COBOL、または CL コンパイラーでコンパイルする必要があります。 OPTION パラメーターは、コンパイル用に *SRCDBG または *LSTDBG に設定する必要があります。

監視サポート

監視サポートは、指定したストレージの内容が変更された時点で、プログラムの実行を停止する機能を提供します。ストレージはプログラム変数の名前で指定します。プログラム変数はストレージのロケーションに変換され、そのストレージの内容の変更がモニターされます。そのストレージの内容が変更されると、実行は停止

します。 中断時点での中断したプログラムのソースが表示され、強調表示されたソース行はストレージを変更したステートメントの後で実行されます。

監視されていない例外

監視されていない例外が発生すると、実行中のプログラムは機能チェックを出し、メッセージをジョブ・ログに送信します。デバッグ・モードの場合、プログラムのモジュールがデバッグ・データとともに作成されていれば、ソース・デバッガーは「モジュール・ソース表示」画面を表示します。必要に応じて、プログラムはデバッグ・モードに追加されます。該当するモジュールが、影響を受けた行が強調表示されて画面に表示されるので、プログラムをデバッグすることができます。

デバッグに関するグローバリゼーション上の制約事項

以下のいずれかの条件が存在する場合、

- デバッグ・ジョブのコード化文字セット ID (CCSID) が 290、930、または 5026 (カタカナ) である場合
- デバッグに使用された装置記述のコード・ページが 290、930、または 5026 (カタカナ) である場合

デバッグ・コマンド、関数、および 16 進数リテラルは、大文字で入力する必要があります。 たとえば、

```
BREAK 16 WHEN var=X'A1B2'
EVAL var:X
```

カタカナ・コード・ページに関する上記の制約は、デバッグ・コマンドに識別名 (たとえば、EVAL) を使用している場合には、適用されません。ただし、ILE RPG、ILE COBOL、または ILE CL モジュールをデバッグする際には、デバッグ・コマンドの識別名は、ソース・デバッガーによって大文字に変換されるので、異なった形で再表示されることがあります。

第 11 章 データ管理機能の有効範囲指定

本章では、ILE プログラムまたはサービス・プログラムによって使用可能なデータ管理機能リソースについて説明します。本章を読むには 53 ページの『データ管理機能の有効範囲指定の規則』で説明した、データ管理機能の有効範囲指定の概念を理解していなければなりません。

各リソース・タイプの詳細については、各 ILE HLL の「プログラマーの手引き」を参照してください。

共通のデータ管理機能リソース

このトピックでは、データ管理機能の有効範囲指定の規則に従っているすべてのデータ管理機能リソースを示します。各リソースの後に、有効範囲指定の方法に関する簡単な説明があります。各リソースに関する詳細については、参照資料をご覧ください。

オープン・ファイル操作

オープン・ファイル操作によって、オープン・データ・パス (ODP) と呼ばれる一時リソースが作成されます。オープン機能は、HLL オープン verb、QUERY ファイル・オープン (OPNQRYF) コマンド、またはデータベース・ファイルのオープン (OPNDBF) コマンドを使用して開始することができます。ODP の有効範囲は、そのファイルをオープンしたプログラムの活動化グループになります。デフォルトの活動化グループで実行される OPM プログラムまたは ILE プログラムの場合、ODP の有効範囲は呼び出しレベル番号になります。 HLL オープン verb の有効範囲指定を変更するには、指定変更を使用します。すべての指定変更コマンド、OPNDBF コマンド、および OPNQRYF コマンドで有効範囲のオープン (OPNSCOPE) パラメーターを使用することによって、有効範囲を指定することができます。

指定変更

指定変更の有効範囲は、呼び出しレベル、活動化グループ・レベル、またはジョブ・レベルになります。指定変更の有効範囲を指定するには、いずれかの指定変更コマンドで指定変更の有効範囲の指定変更 (OVRSCOPE) パラメーターを使用します。明示的な有効範囲を指定しない場合は、指定変更の有効範囲はシステムが指定変更を出した場所によって決まります。システムが指定変更をデフォルトの活動化グループから出した場合は、指定変更の有効範囲は呼び出しレベルになります。 システムが指定変更をそれ以外の活動化グループから出した場合は、指定変更の有効範囲はその活動化グループのレベルになります。

コミットメント定義

コミットメント定義は、活動化グループとジョブ・レベルの有効範囲指定をサポートします。有効範囲指定レベルは、コミットメント制御開始 (STRCMTCTL) コマンドでコミットメント定義有効範囲 (CTLSCOPE) パラ

メーターを使用して指定します。コミットメント定義についての詳細は、iSeries バックアップおよび回復の手引き、SD88-5008 のトピックを参照してください。

ローカル SQL カーソル

ILE コンパイラー・プロダクト用に SQL プログラムを作成することができます。ILE プログラムによって使用される SQL カーソルの有効範囲は、モジュールまたは活動化グループです。SQL カーソルの有効範囲は、SQL プログラム作成コマンドの SQL 終了 (ENDSQL) パラメーターによって指定することができます。

リモート SQL 接続

SQL カーソルに使用されるリモート接続の有効範囲は、通常の SQL 処理の一部として暗黙的に活動化グループに設定されます。これによって、1 つのソース・ジョブと複数のターゲット・ジョブまたは複数のシステムの間に、複数の会話が存在することが可能になります。

ユーザー・インターフェース・マネージャー

プリント・アプリケーション・オープン (QUIOPNPA) API およびディスプレイ・アプリケーション・オープン API は、アプリケーション有効範囲パラメーターをサポートします。これらの API は、ユーザー・インターフェース・マネージャー (UIM) アプリケーションの有効範囲を活動化グループまたはジョブに設定するために使用することができます。ユーザー・インターフェース・マネージャーの詳細については、iSeries Information Center の**プログラミング**・カテゴリーの中の *API* のセクションを参照してください。

オープン・データ・リンク (オープン・ファイル管理)

リンク使用可能化 (QOLELINK) API はデータ・リンクを使用可能にします。この API を ILE 活動化グループ内から使用すると、データ・リンクの有効範囲は、その活動化グループになります。この API をデフォルトの活動化グループ内から使用すると、データ・リンクの有効範囲は、呼び出しレベルになります。オープン・データ・リンクの詳細については、iSeries Information Center の**プログラミング**・カテゴリーの中の *API* のセクションを参照してください。

共通プログラミング・インターフェース (CPI) コミュニケーションの会話

会話を開始する活動化グループは、その会話を所有します。リンク使用可能化 (QOLELINK) API を介してリンクを使用可能にした活動化グループは、そのリンクを所有します。 IBM では、共通プログラミング・インターフェース (CPI) コミュニケーションの会話に関するオンライン情報を用意しています。iSeries Information Center の**プログラミング**・カテゴリーの中の *API* セクションを参照してください。

階層ファイル・システム

ストリーム・ファイル・オープン (OHFOPNSF) API は、階層ファイル・システム (HFS) ファイルを管理します。活動化グループまたはジョブ・レベルへの有効範囲指定を制御するために、この API でオープン情報 (OPENINFO) パラメーターを使用することができます。階層ファイル・システムの詳細については、iSeries Information Center の**プログラミング**・カテゴリーの中の *API* セクションを参照してください。

コミットメント制御の有効範囲指定

ILE は、コミットメント制御に関して、次の 2 つの変更をもたらします。

- ジョブごとの複数の独立したコミットメント定義。トランザクションは、互いに独立してコミットおよびロールバックが可能です。ILE 以前は、ジョブごとに 1 つだけのコミットメント定義が許可されました。
- 活動化グループが正常に終了したときに変更が保留中の場合には、システムは暗黙的に変更をコミットします。ILE の以前は、システムは変更をコミットしませんでした。

コミットメント制御によって、データベース・ファイルまたはテーブルなどのリソースに対する変更を単一トランザクションとして定義し処理することができます。**トランザクション**は、システム上のオブジェクトへの個別の変更からなる 1 つのグループであり、ユーザーは単一のアトミック変更と見なします。コミットメント制御によって、以下のいずれかの操作がシステムで確実に行われます。

- 個別の変更がグループ全体で起きる (**コミット**操作)
- 個別変更がいずれも起こらない (**ロールバック**操作)

OPM および ILE のプログラムの両者を使用して、コミットメント制御のもとで種々のリソースを変更することができます。

コミットメント制御開始 (STRCMTCTL) コマンドによって、ジョブで実行中のプログラムは、コミットメント制御のもとで変更を行うことができます。STRCMTCTL コマンドを使用してコミットメント制御を開始すると、システムは**コミットメント定義**を作成します。各コミットメント定義は、STRCMTCTL コマンドを出したジョブにのみ認識されます。コミットメント定義には、そのジョブ内でコミットメント制御のもとで変更中のリソースに関する情報が入ります。コミットメント定義内のコミットメント制御情報は、コミットメント・リソースが変更されるたびに、システムによって保守されます。コミットメント定義は、コミットメント制御終了 (ENDCMTCTL) コマンドを使用して終了することができます。コミットメント制御についての詳細は、 iSeries バックアップおよび回復の手引き、SD88-5008 のトピックを参照してください。

コミットメント定義および活動化グループ

複数のコミットメント定義を、1 つのジョブで実行中のプログラムによって開始し、使用することができます。1 つのジョブの各コミットメント定義は、関連するリソースをもっている個別のトランザクションを識別します。これらのリソースは、同じジョブに関して開始された他のすべてのコミットメント定義に関係なく、コミットまたはロールバックを行うことができます。

注: デフォルトの活動化グループ以外の活動化グループに関するコミットメント制御を開始できるのは、ILE プログラムだけです。したがって、ジョブは、1 つ以上の ILE プログラムを実行している場合にのみ、複数のコミットメント定義を使用することができます。

オリジナル・プログラム・モデル (OPM) プログラムはデフォルトの活動化グループで実行されます。デフォルトでは、OPM プログラムは *DFTACTGRP コ

ミットメント定義を使用します。OPM プログラムの場合、STRCMTCTL コマンドで CMTSCOPE(*JOB) を指定することによって、*JOB コミットメント定義を使用することができます。

コミットメント制御開始 (STRCMTCTL) コマンドを使用する場合には、コミットメント定義有効範囲 (CMTSCOPE) パラメーターでコミットメント定義の有効範囲を指定します。コミットメント定義の**有効範囲**は、ジョブ内で実行されるどのプログラムがそのコミットメント定義を使用するかを示します。コミットメント定義のデフォルトの有効範囲は、STRCMTCTL コマンドを出したプログラムの活動化グループです。その活動化グループ内で実行されるプログラムだけが、そのコミットメント定義を使用します。有効範囲が活動化グループであるコミットメント定義は、**活動化グループ・レベル**でのコミットメント定義と呼ばれます。OPM デフォルト活動化グループの活動化グループ・レベルで開始されたコミットメント定義は、デフォルトの活動化グループ (*DFTACTGRP) コミットメント定義と呼ばれます。様々な活動化グループ・レベルのコミットメント定義を、1 つのジョブの様々な活動化グループ内で実行するプログラムによって開始し使用することができます。

コミットメント定義の有効範囲をジョブにすることもできます。この有効範囲値を持つコミットメント定義は、**ジョブ・レベル**のコミットメント定義または *JOB コミットメント定義と呼ばれます。活動化グループ・レベルで開始されたコミットメント定義をもたない活動化グループ内で実行中のプログラムは、ジョブ・レベル・コミットメント定義を使用します。この状況は、ジョブ・レベル・コミットメント定義が、当該ジョブの他のプログラムによって開始済みである場合に起こります。1 つのジョブ用に開始できるジョブ・レベル・コミットメント定義は 1 つだけです。

1 つの活動化グループ内で実行されるプログラムが使用できるコミットメント定義は 1 つだけです。1 つの活動化グループ内で実行されるプログラムは、ジョブ・レベルまたは活動化グループ・レベルのいずれかでコミットメント定義を使用することができます。ただし、同時に両方のレベルのコミットメント定義を使用することはできません。

プログラムがコミットメント制御操作を実行するとき、プログラムは、要求に対してどのコミットメント定義を使用するかを直接指示しません。その代わり、要求を行っているプログラムがどの活動化グループを実行中かに基づいて、システムが、使用すべきコミットメント定義を決定します。1 つの活動化グループ内で実行されるプログラムは、一度に 1 つのコミットメント定義しか使用できないのでこのことが可能になります。

コミットメント制御の終了

ジョブ・レベル・コミットメント定義または活動化グループ・レベル・コミットメント定義のコミットメント制御は、コミットメント制御終了 (ENDCMTCTL) コマンドによって終了することができます。 ENDCMTCTL コマンドは、要求を行ったプログラムの活動化グループに対するコミットメント定義を終了するようにシステムに指示します。 ENDCMTCTL コマンドは、ジョブに対する 1 つのコミットメント定義を終了します。ジョブの他のすべてのコミットメント定義は変更されないまま残ります。

活動化グループ・レベルのコミットメント定義を終了すると、その活動化グループ内で実行中のプログラムは、コミットメント制御のもとで変更を行えなくなります。ジョブ・レベルのコミットメント定義が開始されるか、またはすでに存在している場合、コミットメント制御を指定している新しいファイル・オープン操作は、ジョブ・レベルのコミットメント定義を使用します。

ジョブ・レベルのコミットメント定義を終了すると、そのジョブ・レベルのコミットメント定義を使用していたジョブ内で実行中であったどのプログラムも、コミットメント制御のもとで変更を行えなくなります。STRCMTCTL コマンドを使用してコミットメント制御を再度開始すれば、変更を行うことができます。

活動化グループ終了時のコミットメント制御

以下の条件が同時に存在する場合、

- 活動化グループが終了した
- ジョブが終了していない

システムは活動化グループ・レベルのコミットメント定義を自動的に終了します。以下の 2 つの条件が存在する場合、

- 活動化グループ・レベルのコミットメント定義に関するコミットされていない変更が存在する
- 活動化グループが正常に終了した

システムは、コミットメント定義を終了する前に、コミットメント定義に関する暗黙のコミット操作を実行します。一方、以下のいずれかの条件が存在する場合、

- 活動化グループが異常終了した
- 有効範囲が活動化グループであるコミットメント制御のもとでオープンされたいずれかのファイルをクローズする時点で、システムがエラーを検出した

活動化グループ・レベルのコミットメント定義に対して、終了前に、暗黙のロールバック操作が実行されます。活動化グループが異常終了すると、システムは通知オブジェクトを最新の成功したコミット操作で更新します。コミットおよびロールバックは、保留中の変更に基づきます。保留中の変更が無い場合、ロールバックはありませんが、通知オブジェクトは更新されます。活動化グループが保留中の変更で異常終了した場合には、システムは暗黙的に変更をロールバックします。活動化グループが保留中の変更を正常に終了した場合は、システムは暗黙的にその変更をコミットします。

暗黙のコミット操作またはロールバック操作は、*JOB コミットメント定義または *DFTACTGRP コミットメント定義の活動化グループの終了処理時には実行されません。なぜなら、*JOB コミットメント定義および *DFTACTGRP コミットメント定義は、活動化グループの終了によって終了されないからです。その代わり、これらのコミットメント定義は、ENDCMTCTL コマンドによって明示的に終了されるか、またはジョブの終了時にシステムによって終了されます。

活動化グループの終了時に、システムは、有効範囲が活動化グループであるすべてのファイルを自動的にクローズします。これには、有効範囲が活動化グループで、コミットメント制御のもとでオープンされたすべてのデータベース・ファイルが含まれます。このようなファイルに関するクローズ操作は、活動化グループ・レベルのコミットメント定義に対して実行される暗黙のコミット操作の前に行われます。

したがって、入出力バッファーにあるレコードは、暗黙のコミット操作が実行される前に、まずデータベースに送られます。

暗黙のコミット操作またはロールバック操作の一部として、システムは、各 API コミットメント・リソースごとに API コミット / ロールバックの出口プログラムを呼び出します。各 API コミットメント・リソースは、活動化グループ・レベルのコミットメント定義に関連付けられていなければなりません。API コミット / ロールバック出口プログラムを呼び出した後、システムは、API コミットメント・リソースを自動的に除去します。

以下の条件に該当する場合、

- 活動化グループの終了に伴う終了処理でコミットメント定義に対して暗黙のロールバック操作が実行された
- そのコミットメント定義に対して通知オブジェクトが定義されている

通知オブジェクトが更新されます。

第 12 章 ILE バインド可能アプリケーション・プログラミング・インターフェース

ILE バインド可能アプリケーション・プログラミング・インターフェース (バインド可能 API) は、ILE の重要な一部です。このインターフェースは、特定の高水準言語が提供する機能以上の追加機能を提供する場合があります。たとえば、動的ストレージを操作する方法は、すべての HLL に組み込まれているわけではありません。 組み込まれていない場合、特定のバインド可能 API を使用することによって、HLL 機能を補足することができます。HLL が特定のバインド可能 API と同じ機能を提供する場合には、HLL 固有の機能を使用してください。

バインド可能 API は HLL に依存しません。このことは混合言語のアプリケーションの場合に役立ちます。たとえば、混合言語のアプリケーションで条件管理バインド可能 API だけを使用すれば、このアプリケーションに対する条件を処理するセマンティクスを統一することができます。これによって、複数の HLL 固有の条件ハンドラーを使用するより、条件管理の整合性が保たれます。

バインド可能 API は以下の各機能を含む広範囲の機能を提供します。

- 活動化グループと制御の流れの管理
- 条件管理
- 日付時刻操作
- 動的画面管理
- 数学関数
- メッセージ処理
- プログラム呼び出し管理またはプロシージャー呼び出し管理および操作記述子アクセス
- ソース・デバッガー
- ストレージ管理

ILE バインド可能 API については、iSeries Information Center の**プログラミング・**カテゴリーの中の *API* セクションを参照してください。

使用可能な ILE バインド可能 API

大部分のバインド可能 API は、ILE がサポートするすべての HLL で使用することができます。ILE は以下のバインド可能 API を提供します。

活動化グループおよび制御フローのバインド可能 API

- 異常終了 (CEE4ABN)
- 制御境界検出 (CEE4FCB)
- 活動化グループ出口プロシージャー登録 (CEE4RAGE)
- 呼び出しスタック項目登録終了ユーザー出口プロシージャー (CEERTX)
- 緊急条件終了シグナル (CEETREC)
- 呼び出しスタック項目抹消終了ユーザー出口プロシージャー (CEEUTX)

条件管理バインド可能 API

- 条件トークン構成 (CEENCOD)

条件トークン解除 (CEEDCOD)
条件処理 (CEE4HC)
再開カーソルを戻り点へ移動 (CEEMRCR)
ユーザー作成条件ハンドラー登録 (CEEHDLR)
ILE バージョンおよびプラットフォーム ID 検索 (CEEGPID)
相対呼び出し番号入手 (CEE4RIN)
条件シグナル (CEESGL)
ユーザー条件ハンドラー抹消 (CEEHDLU)

日付時刻バインド可能 API

リリアン日付からの曜日の計算 (CEEDYWK)
リリアン形式への日付の変換 (CEEDAYS)
整数表記から秒表記への変換 (CEEISEC)
文字形式へのリリアン日付の変換 (CEEDATE)
秒表記から文字タイム・スタンプへの変換 (CEEDATM)
秒表記から整数表記への変換 (CEESECI)
タイム・スタンプから秒数への変換 (CEESECS)
現在のグリニッジ標準時間の入手 (CEEGMT)
現在のローカル時間の入手 (CEELOCT)
協定世界時とローカル時間の時間差の入手 (CEEUTCO)
協定世界時の入手 (CEEUTC)
世紀の照会 (CEEQCEN)
各国または各地域のデフォルトの日付時刻ストリングの入手 (CEEFMDT)
各国または各地域のデフォルトの日付ストリングの入手 (CEEFMDA)
各国または各地域のデフォルトの時刻ストリングの入手 (CEEFMTM)
世紀の設定 (CEESCEN)

数学バインド可能 API

各数学バインド可能 API の名前の x は、以下のいずれかのデータ・タイプを示します。

I 32 ビット 2 進整数

S 32 ビット単精度浮動小数点数

D 64 ビット倍精度浮動小数点数

T 32 ビット単精度浮動小数点複素数 (実数部と虚数部の長さはともに 32 ビット)

E 64 ビット長精度浮動小数点複素数 (実数部と虚数部の長さはともに 64 ビット)

絶対値関数 (CEESxABS)
アークコサイン (CEESxACS)
アークサイン (CEESxASN)
アークタンジェント (CEESxATN)
アークタンジェント 2 (CEESxAT2)
共役複素数 (CEESxCJG)
コサイン (CEESxCOS)
コタンジェント (CEESxCTN)
誤差関数およびその補数 (CEESxERx)
e を底とする指数 (CEESxEXP)

指数 (CEESxXPx)
階乗 (CEE4SIFAC)
浮動小数点複素数除算 (CEESxDVD)
浮動小数点複素数乗算 (CEESxMLT)
ガンマ関数 (CEESxGMA)
双曲線アークタンジェント (CEESxATH)
双曲線コサイン (CEESxCSH)
双曲線サイン (CEESxSNH)
双曲線タンジェント (CEESxTNH)
複素数の虚数部 (CEESxIMG)
対数ガンマ関数 (CEESxLGM)
10 を底とする対数 (CEESxLG1)
2 を底とする対数 (CEESxLG2)
e を底とする対数 (CEESxLOG)
モジュラー算術 (CEESxMOD)
近似整数 (CEESxNIN)
近似完全数 (CEESxNWN)
差の正数 (CEESxDIM)
サイン (CEESxSIN)
平方根 (CEESxSQT)
タンジェント (CEESxTAN)
符号の移動 (CEESxSGN)
切り捨て (CEESxINT)

他の数学バインド可能 API
基本乱数の発生 (CEERAN0)

メッセージ処理バインド可能 API
メッセージのディスパッチ (CEEMOUT)
メッセージの入手 (CEEMGET)
メッセージの入手、フォーマット設定、およびディスパッチ (CEEMSG)

プログラム呼び出しまたはプロシージャー呼び出しバインド可能 API
ストリング情報入手 (CEEGSI)
操作記述子の情報検索 (CEEDOD)
省略された引き数のテスト (CEETSTA)

ソース・デバッガー・バインド可能 API
プログラムによるデバッグ・ステートメント発行の許可 (QteSubmitDebugCommand)
セッションによるソース・デバッガー使用の可能化 (QteStartSourceDebug)
1 つのビューから別のビューへのマッピング (QteMapViewPosition)
モジュールのビューの登録 (QteRegisterDebugView)
モジュールのビューの除去 (QteRemoveDebugView)
ソース・デバッグ・セッションの属性の検索 (QteRetrieveDebugAttribute)
プログラムのモジュールおよびビューのリストの検索 (QteRetrieveModuleViews)
プログラムの停止位置の検索 (QteRetrieveStoppedPosition)
指定のビューからのソース・テキストの検索 (QteRetrieveViewText)

ソース・デバッグ・セッションの属性の設定 (QteSetDebugAttribute)
ジョブのデバッグ・モードの解除 (QteEndSourceDebug)

ストレージ管理バインド可能 API

ヒープ作成 (CEECRHP)
ヒープ割り振りストラテジー定義 (CEE4DAS)
ヒープ廃棄 (CEEDSHP)
ストレージの解放 (CEEFRST)
ヒープ・ストレージ入手 (CEEGTST)
ヒープ・マーク付け (CEEMKHP)
ストレージ再割り振り (CEECZST)
ヒープ解放 (CEERLHP)

動的画面マネージャー・バインド可能 API

動的画面マネージャー (DSM) バインド可能 API は、画面入出力インターフェースのセットで、ILE 高水準言語用の表示画面を作成し管理する動的な方法を提供します。

DSM API は以下の機能グループに分けることができます。

- **低レベル・サービス**

  低レベル・サービス API は、 5250 データ・ストリーム・コマンドに対する直接インターフェースを提供します。 API が使用されるのは、表示画面の状態を照会し操作する場合、表示画面と対話する入力バッファーおよびコマンド・バッファーを作成し照会し操作する場合、そしてフィールドを定義し表示画面にデータを書き込む場合です。

- **ウィンドウ・サービス**

  ウィンドウ・サービス API は、ウィンドウの作成、削除、移動、およびサイズ変更のために、また、セッション中の複数ウィンドウの並行管理のために使用されます。

- **セッション・サービス**

  セッション・サービス API は、セッションの作成、照会、および操作のために、また、セッションに対する入出力操作の実行のために使用できる汎用ページング・インターフェースを提供します。

IBM では、DSM バインド可能 API に関するオンライン情報を提供しています。 iSeries Information Center の**プログラミング**・カテゴリーの中の *API* セクションを参照してください。

第 13 章 拡張最適化技法

本章では、ILE プログラムおよびサービス・プログラムを最適化するために使用できる、以下の手法について説明します。

- 『プログラム・プロファイル作成』
- 170 ページの『プロシージャー間分析 (IPA)』
- 177 ページの『ライセンス内部コードのオプション』

プログラム・プロファイル作成

プログラム・プロファイル作成は、プログラム実行中に収集した統計データに基づいて、プロシージャーまたはプロシージャー内のコードを ILE プログラムおよびサービス・プログラム内でリオーダーするための拡張最適化技法です。このリオーダーによって、命令キャッシュ使用率を向上させ、このプログラムによるページングを削減することができるので、パフォーマンスが向上します。プログラムのセマンティック動作は、プログラム・プロファイル作成によって影響されません。

プログラム・プロファイル作成によって実現されるパフォーマンス向上は、アプリケーションのタイプに依存しています。一般的に言って、実行時間または入出力処理の実行に時間を費やすよりも、大半の時間をアプリケーション・コードそれ自身に費やすプログラムの方がより大幅な向上を期待できます。プロファイル・データを適用したときに作成されるプログラム・コードのパフォーマンスは、典型的な用途におけるプログラムの最も重要な部分を識別することで、変換プログラムが正しく最適化できているかどうかによって左右されます。したがって、そのプログラムを実行する環境で使用する予定のデータと類似した入力データを使用して、エンド・ユーザーがタスクを実行している間、プロファイル・データを収集することが重要です。

プログラム・プロファイル作成は、次の条件に該当する ILE プログラムやサービス・プログラムでのみ使用可能です。

- そのプログラムが、V4R2M0 以降のリリース用に作成されている。
- プログラムが V5R2M0 よりも前のリリースに合わせて作成されたものである場合、そのプログラムのターゲット・リリースは、現行システムのリリースと同じでなければならない。
- そのプログラムが、*FULL (30) またはそれ以上の最適化レベルを使用してコンパイルされている。V5R2M0 以降のシステムでは、最適化レベルが 30 未満のバインドされたモジュールを使用することも可能ですが、それらのモジュールは、アプリケーション・プロファイルの作成には関与しません。

注: 最適化要件のためには、プログラム・プロファイル作成を使用する前に完全にプログラムをデバッグする必要があります。

プロファイル作成のタイプ

以下の 2 つの方法でプログラムのプロファイルを作成できます。

- ブロック順
- プロシージャー順およびブロック順

ブロック順プロファイルは、条件付きブランチの各ブランチが取られる回数を記録します。ブロック順プロファイル作成データがプログラムに適用されると、最適化変換プログラムによって、プロファイル・ベースのさまざまな最適化がプロシージャー内で実行されます。こうした最適化の一環として、プロシージャー内の最も実行頻度の高いコード・パスがプログラム・オブジェクト内で隣接するように、コードの順序を並び替えます。このリオーダーにより、命令キャッシュや命令プリフェッチ単位などのプロセッサー・コンポーネントの利用効率が高まり、パフォーマンスが向上します。

プロシージャー順プロファイルは、各プロシージャーがプログラム内の他のプロシージャーを呼び出す回数を記録します。最も頻繁に呼び出されるプロシージャーが一緒にパッケージされるように、プログラム内のプロシージャーがリオーダーされます。このリオーダーにより、メモリー・ページングが削減され、パフォーマンスが向上します。

プログラムに対しブロック順だけのプロファイル作成の適用を選択することもできますが、最大パフォーマンスを得られるようにするために、両方のタイプを適用することをお勧めします。

プログラム・プロファイル作成の方法

プログラム・プロファイル作成は、5 つのステップで行います。

1. プロファイル作成データの収集をするためのプログラムを使用可能にする。
2. プログラム・プロファイル作成の開始 (STRPGMPRF) コマンドを使用して、システム上でプログラム・プロファイル作成データの収集を開始する。
3. 使用頻度の高いコードのパスでプログラムを実行させることによって、プロファイル作成データを収集する。使用頻度の高いコードのパスの最適化を実施するために、プログラム・プロファイル作成はプログラムの実行中に収集された統計データを使用するので、このデータはアプリケーションの典型的なケースで収集されることが重要です。
4. プログラム・プロファイル作成の終了 (ENDPGMPRF) コマンドを使用して、システムに関するプログラム・プロファイル作成データの収集を終了する。
5. 収集されたプロファイル作成データに基づいて最適なパフォーマンスを得るようにコードをリオーダーするように、収集されたプロファイル作成データをプログラムに適用する。

プログラムでプロファイル作成データが収集できるようにする

プログラムにバインドされたモジュールの少なくとも 1 つがプロファイル作成データの収集が可能である場合、そのプログラムはプロファイル作成データの収集が可能です。プロファイル作成データの収集を可能にするには、1 つ以上の *MODULE オブジェクトをプロファイル作成データの収集可能に変更し、その後これらのモジュールを使用してプログラムを作成または更新するか、あるいはプログラムの作成

後にプロファイル作成データを収集するように変更します。いずれの技法をとっても、モジュールがバインドされたプログラムはプロファイル作成データの収集が可能になります。

使用している ILE 言語によっては、プロファイル作成データの収集を可能にするようにモジュールを作成するオプションがコンパイラー・コマンドにある場合があります。ILE 言語が最低 *FULL (30) の最適化レベルをサポートしていれば、モジュールの変更 (CHGMOD) コマンドの、プロファイル作成データ (PRFDTA) パラメーターで *COL を指定することによってプロファイル作成データを収集するように ILE モジュールを変更することができます。

プログラムが、作成後にプログラム変更 (CHGPGM) またはサービス・プログラムの変更 (CHGSRVPGM) コマンドを使用してプロファイル作成データを収集できるようにするには、識別可能プログラムに関して以下のことを行う必要があります。

- プロファイル作成データ (PRFDTA) パラメーターで *COL を指定する。この指定は、プログラム内でバインドされた、以下のすべてのモジュールに影響を与えます。
  - V4R2M0 以降のリリースに合わせて作成されているモジュール。 V5R2M0 よりも前のシステムを使用している場合、プロファイル作成データを収集できるようにするためには、このプログラムを、プログラムを作成したシステムと同じリリース・レベルのシステムに配置する必要があります。バインドされたモジュールにも同じ制約が適用されます。
  - 最適化レベルが 30 以上であるモジュール。 V5R2M0 以降のリリースでは、最適化レベルが 30 未満のバインドされたモジュールを使用することも可能ですが、それらのモジュールは、アプリケーション・プロファイルの作成には関与しません。

注: V5R2M0 よりも前のリリースのシステムでアプリケーション・プロファイル作成データを収集することのできるプログラムは、そのデータを V5R2M0 以降のシステムに適用することができますが、その結果が最適なものになるとはかぎりません。 V5R2M0 以降のシステムにプロファイル作成データを適用したり、作成されたプログラムをそのようなシステムで使用したりする場合には、V5R2M0 以降のシステムでそのプログラムに関するプロファイル作成データを収集できるようにする必要があります。

モジュールまたはプログラムをプロファイル作成データ収集可能にするには、オブジェクトを再作成する必要があります。したがって、モジュールまたはプログラムをプロファイル作成データ収集可能にするのに必要な時間は、オブジェクトを強制的に再作成する (FRCCRT パラメーター) のに必要な時間と同程度です。さらに、最適化変換プログラムによって生成された追加のマシン・インストラクションのために、オブジェクトのサイズは大きくなります。

プログラムまたはモジュールのプロファイル作成データの収集を可能にすると、プログラム識別情報は以下のいずれかが生じるまで除去できません。

- 収集されたプロファイル作成データがプログラムに適用される。
- そのプログラムまたはモジュールが、プロファイル作成データの収集ができないように変更される。

モジュールまたはプログラムがプロファイル作成データ収集を使用することが可能になっているかどうかを判別するには、DETAIL(*BASIC) を指定して、モジュールの表示 (DSPMOD)、プログラム表示 (DSPPGM) またはサービス・プログラムの表示 (DSPSRVPGM) コマンドを使用します。プログラムまたはサービス・プログラムの場合、DETAIL(*MODULE) からオプション 5 (記述の表示) を使用すると、どのバインドされたモジュールがプロファイル作成データを収集することが可能になっているかを判別できます。詳細については 168 ページの『プログラムまたはモジュールのプロファイルが作成されているかまたは収集が可能になっているかどうかを知る方法』のトピックを参照してください。

注: プログラムがすでにプロファイル作成データ (プログラム実行中に収集された統計データ) を収集している場合、このデータはプログラムが再度プロファイル作成データ収集可能にされた時点で消去されます。詳細については 166 ページの『プロファイル作成データの収集が可能にされたプログラムの管理』を参照してください。

プロファイル作成データの収集

プログラム・プロファイル作成は、プロファイル作成データの収集が可能になったプログラムを実行するマシン上で開始する必要があります。これは、そのプログラムがプロファイル作成データのカウントを更新できるようにするためです。これによって、大規模で長時間実行するアプリケーションを開始でき、プロファイル作成データを収集する前に定常状態に達することができます。これによって、データ収集がいつ発生するかを制御できます。

プログラム・プロファイル作成の開始 (STRPGMPRF) コマンドを使用して、マシン上でプログラム・プロファイル作成を開始します。マシン上のプログラム・プロファイル作成収集を終了するには、プログラム・プロファイル作成の終了 (ENDPGMPRF) コマンドを使用します。IBM は、これらのコマンドをいずれも *EXCLUDE の共通権限で出荷します。プログラム・プロファイル作成は、マシンの IPL が行われる時点で暗黙的に終了します。

プログラム・プロファイル作成が開始されると、実行中の、プロファイル作成データの収集が可能になったすべてのプログラムまたはサービス・プログラムは、プロファイル作成データ・カウントを更新します。これは、STRPGMPRF コマンドが出される前にプログラムが活動化されていたかどうかにかかわらず行われます。

プロファイル作成データを収集中のプログラムがマシン上の複数のジョブによって呼び出すことができる場合は、プロファイル作成データ・カウントは、これらのジョブすべてによって更新されます。これが望ましくない場合は、プログラムの重複のコピーを別個のライブラリー内に作成してそのコピーを代用する必要があります。

注:

1. プログラム・プロファイル作成がマシンで開始された場合、プロファイル作成データ収集が可能になったプログラムの実行時に、プロファイル作成データ・カウントが増やされます。したがって、以前にこのプログラムが実行された後にこれらのカウントを消去しなかった場合には、「失効した」プロファイル作成データ・カウントが追加される可能性があります。プロファイル作成データ・カウン

トを強制的に消去するには、いくつかの方法があります。詳細については 166 ページの『プロファイル作成データの収集が可能にされたプログラムの管理』を参照してください。

2. プロファイル作成データ・カウントは、増えるたびに DASD に書き込まれるわけではありません。そのようにすると、プログラムの実行時間が大幅に長くなってしまうためです。プロファイル作成データ・カウントが DASD に書き込まれるのは、プログラムが自然にページアウトされた時に限られます。プロファイル作成データ・カウントが DASD に書き込まれたことを確認するには、プールの消去 (CLRPOOL) コマンドを使用して、プログラムの実行に使用されているストレージ・プールを消去してください。

収集されたプロファイル作成データの適用

収集されたプロファイル作成データの適用するには、以下を行います。

1. パフォーマンスを最適化するために、収集されたプロファイル作成データ (プロシージャー順プロファイル作成データ) を使用して、プログラムのプロシージャーをリオーダーするようにマシンに指示する。
2. パフォーマンスを最適化するために、収集されたプロファイル作成データ (基本ブロック・プロファイル作成データ) を使用して、プログラム内のプロシージャーの中でコードをリオーダーするように、マシンに指示する。
3. プログラムがプロファイル作成データを収集できるようになっていたときに追加されたマシン命令を、プログラムから除去する。これにより、プログラムはプロファイル作成データを収集することができなくなります。
4. 収集されたプロファイル作成データを、以下の識別可能なデータとしてそのプログラムに保管する。
   - *BLKORD (基本ブロック・プロファイル・プログラム識別情報)
   - *PRCORD (プロシージャー順プロファイル・プログラム識別情報)

収集データが一度プログラムに適用されると、再び適用することはできません。プロファイル作成データを再び適用するには 162 ページの『プログラム・プロファイル作成の方法』に概説されているステップを実行する必要があります。**すでに適用されたプロファイル作成データは、プログラムがプロファイル作成データ収集可能になったときに、すべて廃棄されます。**

すでに収集したデータを再び適用したい場合は、プロファイル作成データを適用する前にプログラムのコピーを作成することをお勧めします。各タイプのプロファイル (ブロック順、またはブロックおよびプロシージャー順) のいずれが有利かを試している場合には、これが望ましい方法です。

プロファイル作成データを適用するには、プログラム変更 (CHGPGM) またはサービス・プログラムの変更 (CHGSRVPGM) コマンドを使用します。プロファイル作成データ (PRFDTA) パラメーターには、以下を指定します。

- ブロック順プロファイル作成データの場合、*APYBLKORD
- ブロック順およびプロシージャー順の両方のプロファイル作成データの場合、*APYALL または *APYPRCORD

IBM では、*APYALL の使用をお勧めしています。

プログラムにプロファイル作成データを適用すると、2 つの形式のプログラム識別情報が追加作成され、そのプログラムとともに保管されます。これらの追加プログラム識別情報は、プログラム変更 (CHGPGM) およびサービス・プログラムの変更 (CHGSRVPGM) コマンドを使用して除去することができます。

- *BLKORD プログラム識別情報は、ブロック順プロファイル・プログラム識別情報がプログラムに適用されたときに、暗黙的に追加されます。これによって、マシンは、プログラムが再作成される場合に備えて、プログラムのために適用されたブロック順プロファイル・プログラム識別情報を保存することができるようになります。
- プロシージャー順プロファイル作成データをプログラムに適用すると、暗黙的に *PRCORD および *BLKORD プログラム識別情報が追加されます。これによって、マシンは、プログラムが再作成または更新される場合に備えて、プログラムのために適用されたプロシージャー順プロファイル作成データを保存することができます。

たとえば、ブロック順プロファイル作成データをプログラムに適用して、その後、*BLKORD プログラム識別情報を除去します。プログラムは、ブロック順にプロファイルが作成されたままです。しかし、プログラムの再作成の原因となる変更が行われた場合、プログラムは、ブロック順にはプロファイル作成されないようになります。

注: *CRTDTA プログラム識別情報を除去すると、*BLKORD プログラム識別情報も暗黙的に除去されます。これは、*BLKORD プログラム識別情報が必要なのは、プログラムが再作成されるときに限られるからです。*CRTDTA プログラム識別情報が除去されると、プログラムの再作成は行えなくなるため、*BLKORD も必要ではなくなり、一緒に除去されます。しかし、*PRCORD プログラム識別情報は除去されません。

プロファイル作成データの収集が可能にされたプログラムの管理

プログラム変更 (CHGPGM) またはサービス・プログラムの変更 (CHGSRVPGM) コマンドを使用して、プロファイル作成データ収集が可能になっているプログラムを変更した場合、その変更によってプログラムの再作成が必要となる場合には、プロファイル作成データ・カウントが暗黙的にゼロになります。たとえば、最適化レベル *FULL から最適化レベル 40 までのプロファイル作成データの収集が可能になったプログラムを変更すると、収集されたプロファイル作成データは、すべて暗黙的に消去されます。プロファイル作成データの収集が可能になったプログラムを復元し、FRCOBJCVN(*YES *ALL) がオブジェクト復元 (RSTOBJ) コマンドで指定された場合も同様です。

同様に、プログラムの更新 (UPDPGM) またはサービス・プログラムの更新 (UPDSRVPGM) コマンドを使用してプロファイル作成データの収集が可能になったプログラムを更新すると、その結果のプログラムがプロファイル作成データの収集が可能になったままの場合、プロファイル作成データ・カウントが暗黙的に消去されます。たとえば、プログラム P1 にはモジュール M1 および M2 が含まれているとします。P1 にバインドされたモジュール M1 はプロファイル作成データ収集可能ですが、M2 はそうでないとします。モジュールの 1 つが可能になっている限り、モジュール M1 または M2 をもつプログラム P1 を更新すると、プロファイル作成データ収集可能のままのプログラムになります。プロファイル作成データ・

カウントはすべて消去されます。しかし、モジュールの変更 (CHGMOD) コマンドのプロファイル作成データ (PRFDTA) パラメーターで *NOCOL を指定してモジュール M1 がもはやプロファイル作成データ収集可能でなくなるように変更された場合には、M1 を使用してプログラム P1 を更新すると、プログラム P1 はもはやプロファイル作成データ収集可能ではなくなります。

プログラム変更 (CHGPGM) またはサービス・プログラムの変更 (CHGSRVPGM) コマンドのプロファイル作成データ (PRFDTA) パラメーターで *CLR オプションを指定して、プログラムから明示的にプロファイル・カウントを消去することができます。*CLR オプションを使用するには、プログラムを活動化してはならないことに注意してください。

プログラムにプロファイル作成データの収集を行わせたくない場合には、以下のいずれかのステップを行う必要があります。

- プログラム変更 (CHGPGM) コマンドのプロファイル作成データ (PRFDTA) パラメーターに *NOCOL を指定する。
- サービス・プログラムの変更 (CHGSRVPGM) コマンドのプロファイル作成データ (PRFDTA) パラメーターに *NOCOL を指定する。

これによって、プログラムはプロファイル作成データの収集が可能になる前の状態に戻ります。CHGMOD コマンドまたはモジュールの再コンパイルして、モジュールの PRFDTA の値を *NOCOL に変更して、モジュールをプログラムに再バインドすることもできます。

プロファイル作成データが適用されたプログラムの管理

適用されたプロファイル作成データをもつプログラムが、プログラム変更 (CHGPGM) またはサービス・プログラムの変更 (CHGSRVPGM) コマンドを使用して変更された場合、以下の条件が両方とも当てはまるならば、適用されたプロファイル作成データは失われます。

- 変更によってプログラムの再作成が必要となる。

  注: プロファイル作成データが適用されているプログラムの最適化レベルは、30 未満に変更することはできません。これは、プロファイル作成データが最適化レベルによって異なるからです。

- 必要なプロファイル・プログラム識別情報が除去された。

さらに、プログラムをプロファイル作成データ収集可能にする変更要求の場合には、プロファイル・プログラム識別情報が除去されたかどうかにかかわらず、適用されたプロファイル作成データはすべて失われます。このような要求によって、プログラムはプロファイル作成データ収集可能になります。

以下に例を挙げます。

- プログラム A は、プロシージャー順およびブロック順のプロファイル作成データが適用されています。*BLKORD プログラム識別情報はプログラムから除去されましたが、*PRCORD プログラム識別情報は除去されていません。CHGPGM コマンドが実行されて、プログラム A のパフォーマンス収集属性が変更され、また、プログラムの再作成も必要となります。この変更要求によって、プログラム A は、もはやブロック順にプロファイルは作成されません。しかし、プロシージャー順プロファイル作成データは適用されたままです。

- プログラム A は、プロシージャー順およびブロック順のプロファイル作成データが適用されています。*BLKORD および *PRCORD プログラム識別情報はプログラムから除去されています。 CHGPGM コマンドが実行されて、プログラム A のユーザー・プロファイル属性が変更され、また、プログラムの再作成も必要となります。この変更要求によって、プログラム A はもはやブロック順またはプロシージャー順にプロファイルが作成されていません。プログラム A は、プロファイル作成データが適用される前の状態に戻ります。
- プログラム A は、ブロック順のプロファイル作成データが適用されています。*BLKORD プログラム識別情報がプログラムから除去されました。CHGPGM コマンドが実行されて、プログラムのテキストが変更されますが、プログラムを再作成する必要は**ありません**。この変更の後でも、プログラム A のプロファイル作成はブロック順で行われます。
- プログラム A は、プロシージャー順およびブロックのプロファイル作成データが適用されています。これは、プログラムから *PRCORD および *BLKORD のプログラム識別情報を除去しません。プログラムがプロファイル作成データを収集できるようにするために、CHGPGM コマンドを実行してください (これにより、プログラムが再作成されます)。この変更要求によって、プログラム A はもはやブロック順またはプロシージャー順にプロファイルが作成されていません。これは、そのプログラムをプロファイル作成データが適用されたことがなかった場合と同様の状態にします。これは、プロファイル作成データのすべてのカウントを消去して、プログラムがプロファイル作成データを収集できるようにします。

(*APYALL、*APYBLKORD、または *APYPRCORD を使用することによって) プロファイル作成データが適用されているプログラムは、ただちに CHGPGM または CHGSRVPGM コマンドで PRFDTA(*NOCOL) を指定して、プロファイルを作成しないプログラムに変更することはできません。このようになっているのは、プロファイル作成データを不注意によって喪失しないようにするための安全策です。本当にプロファイルを作成しないようにしたい場合には、まず最初にプログラムを PRFDTA(*COL) に変更することによって正しく既存プロファイルを除去してから、PRFDTA(*NOCOL) に変更する必要があります。

プログラムまたはモジュールのプロファイルが作成されているかまたは収集が可能になっているかどうかを知る方法

プログラム・プロファイル作成データ属性を判別するには、プログラム表示 (DSPPGM) またはサービス・プログラムの表示 (DSPSRVPGM) コマンドを DETAIL(*BASIC) を指定して使用します。「プロファイル作成データ」の値は、以下のいずれかです。

- *NOCOL - プログラムは、プロファイル作成データ収集が可能になっていません。
- *COL - プログラム内の 1 つ以上のモジュールのプロファイル作成データの収集が可能になっています。この値は、プロファイル作成データが実際に収集されたかどうかを示すものではありません。
- *APYALL - ブロック順およびプロシージャー順プロファイル作成データがこのプログラムに適用されています。プロファイル作成データの収集はもはや可能ではありません。

- *APYBLKORD - ブロック順プロファイル作成データがこのプログラム内の 1 つ以上のバインドされたモジュールのプロシージャーに適用されています。 これが適用されるのは、すでにプロファイル作成データの収集が可能になったバインド済みモジュールのみです。プロファイル作成データの収集はもはや可能ではありません。
- *APYPRCORD - プロシージャー順プログラム・プロファイル作成データがこのプログラムに適用されています。プロファイル作成データの収集はもはや可能ではありません。

プログラムにプロシージャー順プロファイルだけを適用するには、以下のようにしてください。

- まず、*APYALL または *APYPRCORD (これは *APYALL と同じです) を指定してプロファイルを作成します。
- そして、*BLKORD プログラム識別情報を除去して、プログラムを再作成します。

プログラムにバインドされたモジュールのプログラム・プロファイル作成データ属性を表示するには、DSPPGM または DSPSRVPGM DETAIL(*MODULE) を使用します。プログラムにバインドされたモジュールでオプション 5 を指定して、モジュール・レベルでこのパラメーターの値を調べてください。「プロファイル作成データ」の値は、以下のいずれかです。

- *NOCOL - このバインドされたモジュールは、プロファイル作成データの収集が可能になっていません。
- *COL - このバインドされたモジュールは、プロファイル作成データの収集が可能になっています。この値は、プロファイル作成データが実際に収集されたかどうかを示すものではありません。
- *APYBLKORD - ブロック順プロファイル作成データがこのバインドされたモジュールの 1 つ以上のプロシージャーに適用されています。 プロファイル作成データの収集はもはや可能ではありません。

さらに DETAIL(*MODULE) は、以下のフィールドを表示して、プログラム・プロファイル作成データ属性によって影響されたプロシージャーの数を示します。

- プロシージャーの数 - モジュール内のプロシージャーの総数。
- 再順序づけされたプロシージャー・ブロックの数 - 基本ブロック順にリオーダーされた該当モジュール内のプロシージャーの数。
- 測定されたプロシージャー・ブロック順序の数 - ブロック順プロファイル作成データが適用されたとき、ブロック順プロファイル作成データを収集した、このバインドされたモジュール内のプロシージャーの数。ベンチマークの実行時に、あるプロシージャーがベンチマーク内で実行されなかったために、その特定のプロシージャーのデータが収集されなかった可能性があります。したがって、このカウントはベンチマーク内で実行されたプロシージャーの数を反映しています。

DSPMOD コマンドを使用して、モジュールのプロファイル作成属性を判別します。「プロファイル作成データ」の値は、以下のいずれかです。*APYBLKORD を表示することは決してありません。基本ブロック・データが適用できるのは、プログラムにバインドされたモジュールのみだからです。スタンドアロン・モジュールには決して適用されません。

- *NOCOL - モジュールは、プロファイル作成データの収集が可能になっていません。
- *COL - モジュールは、プロファイル作成データの収集が可能になっています。

プロシージャー間分析 (IPA)

このトピックでは、CRTPGM および CRTSRVPGM コマンドの IPA オプションを介して使用可能な、プロシージャー間分析 (IPA) 処理について概説します。

コンパイル時に、最適化変換プログラムは、プロシージャー内分析とプロシージャー間分析の両方を実行します。プロシージャー内分析は、コンパイル単位内の各関数の最適化を、その関数とコンパイル単位でのみ使用可能な情報を使用して実行するメカニズムです。プロシージャー間分析は、関数の境界を超えて最適化を実行するメカニズムです。最適化変換プログラムは、プロシージャー間分析を実行しますが、1 つのコンパイル単位内にのみ限定されます。IPA コンパイラー・オプションによって実行されるプロシージャー間分析は、上記の限定されたプロシージャー間分析を改善しました。IPA オプションを介してプロシージャー間分析を実行すると、IPA はプログラム全体に渡って最適化を実行します。さらに、コンパイル時に最適化変換プログラムを使用して、他の方法では実行できない最適化も実行します。最適化変換プログラムまたは IPA オプションは、以下のタイプの最適化を実行します。

- *複数のコンパイル単位に渡るインライン化*。インライン化は、特定の関数呼び出しを、関数の実際のコードと置換します。インライン化は、呼び出しのオーバーヘッドを除去するだけでなく、関数全体を呼び出し元に露出することになるので、コンパイラーはコードをさらに最適化することができます。
- *プログラムの区画化*。プログラムの区画化では、関数を再順序付けして、参照の局所性を活用することによって、パフォーマンスを改善します。区画化では、頻繁に相互に呼び出しを行う関数をメモリー内で接近させて置くようにします。プログラムの区画への分割の詳細については 176 ページの『IPA によって作成される区画』を参照してください。
- *グローバル変数の合体*。コンパイラーは、グローバル変数を 1 つ以上の構造に書き込んでおき、構造の先頭からのオフセットを計算することで、これらの変数にアクセスしています。これにより、変数アクセスのためのコストを低減させ、データの局所性を活用できます。
- *コードの直線化*。コードを直線化することにより、プログラムのフローが合理化されます。
- *到達不能コードの除去*。到達不能コードの除去によって、関数内の到達不能コードが除去されます。
- *到達不能関数の呼び出しグラフのプルーニング*。到達不能関数の呼び出しグラフのプルーニングによって、100% インライン化されているか、またはまったく参照されないコードが除去されます。
- *プロシージャー内での定数の伝搬および設定の伝搬*。IPA は、浮動小数点定数と整数定数をその使用法に合わせて伝搬し、定数式はコンパイル時に計算します。また、いくつかの定数のうちの 1 つであると分かっている変数使用は、結果として、複数の条件とスイッチをフォールディングします。

- *プロシージャー内ポインターの別名分析*。IPA は、ポインターの使用法に対する定義を追跡して、ポインター参照解除で使用または定義する可能性があるメモリー・ロケーションに関する情報がより詳細になるようにしています。これにより、コンパイラーの他の部分が、このような参照解除に関連するコードをさらに最適化できるようになります。IPA は、データ・ポインターおよび関数ポインターの定義を追跡します。ポインターが単一のメモリー・ロケーションまたは関数のみしか参照できない場合、IPA は、そのメモリー・ロケーションまたは関数を明示的に参照するようにポインターを書き直します。
- *プロシージャー内コピーの伝搬*。IPA は、式を伝搬し、一部の変数は、変数の使用法を定義します。これによって、さらに定数式のフォールディングを行う機会が与えられます。これは、冗長な変数コピーも除去します。
- *プロシージャー内到達不能コードおよび保管の除去*。IPA は到達不能な変数の定義を、その定義を作成するための計算と共に除去します。
- *参照 (アドレス) 引き数の値引き数への変換*。IPA は、仮パラメーターが呼び出し先プロシージャーで指定されていない場合に、参照 (アドレス) 引き数を値引き数へ変換します。
- *静的変数の自動 (スタック) 変数への変換*。IPA は、静的変数の使用が単一のプロシージャー呼び出しに限定されている場合は、その静的変数を自動 (スタック) 変数に変換します。

通常、IPA を使用して最適化したコードの実行時間は、コンパイル時にのみ最適化したコードの実行時間より速くなります。しかし、すべてのアプリケーションが IPA 最適化に適しているわけではありません。また、IPA の使用によって実現されるパフォーマンスの向上は、アプリケーションによって異なります。アプリケーションによっては、プロシージャー間分析を使用しても、パフォーマンスが向上しない可能性があります。実際、まれには、プロシージャー間分析を使用すると、アプリケーションのパフォーマンスが低下する場合があります。このような場合には、プロシージャー間分析を使用しないことをお勧めします。プロシージャー間分析によって実現されるパフォーマンス向上は、アプリケーションのタイプに依存します。パフォーマンスが向上する可能性が高いアプリケーションは、以下の特性を持つアプリケーションです。

- 多数の関数を含んでいる。
- 多数のコンパイル単位を含んでいる。
- 呼び出し元と同じコンパイル単位にない多数の関数を含んでいる。
- 多数の入出力操作を実行しない。

プロシージャー間の最適化は、次の条件に該当する ILE プログラムやサービス・プログラムでのみ使用可能です。

- 特に V4R4M0 以降のリリースで、プログラムまたはサービス・プログラムにバインドされるモジュールを作成した場合。
- 20 (*BASIC) 以上の最適化レベルで、プログラムまたはサービス・プログラムにバインドされるモジュールをコンパイルした場合。
- プログラムまたはサービス・プログラムにバインドされるモジュールに、それに関連付けられている IL データがある場合。中間言語 (IL) データをモジュールと一緒に保持するには、モジュール作成オプション MODCRTOPT(*KEEPILDTA) を使用してください。

注: 最適化の要件を満たすためには、プロシージャー間分析を使用する前に、完全にプログラムをデバッグする必要があります。

IPA を使用してプログラムを最適化する方法

IPA を使用して、プログラム・オブジェクトまたはサービス・プログラム・オブジェクトを最適化するには、以下のステップを実行してください。

1. プログラムまたはサービス・プログラムに必要なすべてのモジュールを、MODCRTOPT(*KEEPILDTA) および最適化レベル 20 以上 (できれば 40) を指定してコンパイルする。 DETAIL(*BASIC) パラメーターを指定して DSPMOD コマンドを使用することにより、単一のモジュールが正しいオプションを指定してコンパイルされていることを確認できます。 IL データが存在している場合には、**中間言語データ**・フィールドに *YES の値が入ります。**最適化レベル**・フィールドは、モジュールの最適化レベルを示します。
2. CRTPGM または CRTSRVPGM コマンドで IPA(*YES) を指定する。バインドの IPA 部分が実行している間、システムは IPA が進行中であることを示す状況メッセージを表示します。

IPA がプログラムを最適化する方法をさらに詳細に定義するには、以下のパラメーターを使用します。

- 追加の IPA サブオプション情報を提供するには、IPACTLFILE(*IPA-control-file*) を指定してください。制御ファイル内で指定できるオプションのリストについては 『IPA 制御ファイルの構文』を参照してください。

CRTPGM コマンドで IPA(*YES) を指定した場合には、プログラムに対する更新も可能にすることはできません (すなわち、ALWUPD(*YES) を指定することはできません)。これは、CRTSRVPGM コマンドの ALWLIBUPD パラメーターに関しても同じです。IPA(*YES) と一緒に指定する場合、このパラメーターは ALWLIBUPD(*NO) でなければなりません。

IPA 制御ファイルの構文

IPA 制御ファイルは、追加の IPA 処理ディレクティブを含むストリーム・ファイルです。制御ファイルはファイルの 1 つのメンバーであってもよく、QSYS.LIB 命名規則を使用します (たとえば、`/qsys.lib/mylib.lib/xx.file/yy.mbr`)。IPACTLFILE パラメーターは、このファイルのパス名を示します。

制御ファイルのディレクティブの構文が無効である場合、IPA はエラー・メッセージを出します。

制御ファイルの中で、以下のディレクティブを指定することができます。

exits=name[,name]

関数のリストを指定します。各関数は常にプログラムを終了させます。このような関数の呼び出しは、プログラムに戻ることがないので、最適化 (たとえば、保管と復元の手順の除去) が可能です。これらの関数は、IL データに関連しているプログラムの他の部分を呼び出すものであってはなりません。

inline=attribute

インラインで処理させたい関数をコンパイラーに識別させる方法を指定します。このディレクティブに関して、以下の属性を指定することができます。

auto 関数をインライン化できるかどうかを、インラインしきい値およびインラインしきい値に基づいて、インライン化プログラムが判断する必要があることを指定します。noinline ディレクティブは、自動インライン化を指定変更します。これはデフォルトです。

noauto

inline ディレクティブを使用して名前を指定した関数のみのインライン化を IPA が考慮する必要があることを指定します。

name[,name]

インライン化したい関数のリストを指定します。指定した関数が必ずしもインライン化されるわけではありません。

name[,name] from name[,name]

関数が特定の関数または関数のリストから呼び出される場合に、インライン化の望ましい候補である関数のリストを指定します。指定した関数が必ずしもインライン化されるわけではありません。

inline-limit=num

インライン化が停止するまでに関数が成長できる最大相対サイズを (抽象コード単位で) 指定します。抽象コード単位は、関数内の実行可能コードのサイズに比例します。この数値を大きくすれば、コンパイラーは、より大きなサブプログラムまたはより多くのサブプログラム呼び出し、あるいはその両方をインライン化することができます。このディレクティブは、inline=auto がオンの場合のみ適用できます。デフォルト値は 8192 です。

inline-threshhold=size

自動インライン化の候補にできる関数の最大相対サイズを (抽象コード単位で) 指定します。このディレクティブは、inline=auto がオンの場合のみ適用できます。デフォルトのサイズは 1024 です。

isolated=name[,name]

isolated 関数のリストを指定します。isolated 関数は、可視の関数にアクセス可能なグローバル変数を直接的に (あるいは、その呼び出しチェーン内の別の関数を介して間接的に) 参照または変更することのない関数です。IPA は、サービス・プログラムからバインドされた関数は、isolated 関数であると想定します。

lowfreq=name[,name]

頻繁に呼び出されないと予想される関数の名前を指定します。このような関数は一般的にはエラー処理関数またはトレース関数です。IPA は、このような関数の呼び出しの最適化をあまり行わないことによって、プログラムの他の部分をより高速にすることができます。

missing=attribute

missing 関数のプロシージャー間動作を指定します。 missing 関数は、IL データに関連付けられていない関数で、unknown、safe、isolated、または pure ディレクティブに明示的に指定されていない関数です。これらのディレクティブは、IL データに関連付けられていないライブラリー・ルーチンに対する呼び出しで、IPA が安全に実行できる最適化の程度を指定します。

IPA にとっては、これらの関数内のコードは可視ではありません。すべてのユーザー参照が、ユーザー・ライブラリーまたは実行時ライブラリーを使用して確実に解決されるようにする必要があります。

このディレクティブのデフォルトの設定値は unknown です。 *unknown* は、IPA に、このような missing 関数への呼び出しを介して使用および変更する可能性があるデータに関して、また、このような呼び出しを介して間接的に呼び出される可能性がある関数に関して、悲観的な想定を行うように指定します。このディレクティブに関して、以下の属性を指定することができます。

unknown
missing 関数が ″unknown″ であることを指定します。下記の unknown ディレクティブの説明を参照してください。これがデフォルトの属性です。

safe missing 関数が ″safe″ であることを指定します。下記の safe ディレクティブの説明を参照してください。

isolated
missing 関数が ″isolated″ であることを指定します。上記の isolated ディレクティブの説明を参照してください。

pure missing 関数が ″pure″ であることを指定します。下記の pure ディレクティブの説明を参照してください。

noinline=name[,name]
コンパイラーがインライン化しない関数のリストを指定します。

noinline=name[,name] from name[,name]
関数が特定の関数または関数のリストから呼び出される場合に、コンパイラーがインライン化しない関数のリストを指定します。

partition=small| medium|large|unsigned-integer
IPA が作成する各プログラム区画のサイズを指定します。区画のサイズは、リンクに必要な時間および生成されるコードの品質に直接比例します。区画サイズが大きい場合、リンクに必要な時間は長くなりますが、生成されるコードの品質は一般的によりよくなります。

このディレクティブのデフォルトは medium です。

さらに詳細な度合いを求めるために、符号なしの整数値を使用して、区画サイズを指定することができます。この整数は抽象コード単位であり、その意味はリリースごとに異なる可能性があります。この整数は、非常に短期間の調整作業で、または区画数を固定にする必要がある状況でのみ使用してください。

pure=name[,name]
pure 関数のリストを指定します。これらの関数は safe および isolated である関数です。 pure 関数の内部状態は監視不能です。これは、関数の特定の呼び出しの戻り値が、この関数の以前または将来の呼び出しとは独立していることを意味します。

safe=name[,name]
safe 関数のリストを指定します。これらの関数は、IL データが関連付けら

れている関数を直接的または間接的に呼び出さない関数です。safe 関数はグローバル変数を参照および変更する可能性があります。

unknown=name[,name]

unknown 関数のリストを指定します。これらの関数は、safe、isolated、または pure でない関数です。

IPA の使用上の注意

- IPA を使用した場合、バインド時間が増加する可能性があります。アプリケーションのサイズおよびプロセッサーの速度によっては、バインド時間が大幅に増加する可能性があります。
- IPA によって、従来のバインディングと比較するとかなり大きなバインド済みのプログラムおよびサービス・プログラムが生成される可能性があります。
- IPA のプロシージャー間最適化によって、プログラムのパフォーマンスが大幅に向上する可能性がありますが、エラーがあるにもかかわらずに機能していたプログラムが失敗する場合があります。
- IPA は関数をインライン化してコンパイルするので、相対スタック・フレーム・オフセット (たとえば、QMHRCVPM) を受け入れる API を使用する場合には、注意が必要です。
- 関数をインラインでコンパイルするために、IPA はそれ自身のインライナーを使用し、バックエンドのインライナーは使用しません。コンパイル・コマンドで INLINE オプションを使用するような、バックエンドのインライナーに指定されたパラメーターはすべて無視されます。IPA インライナーのためのパラメーターは、IPA 制御ファイルで指定されます。

IPA の制約事項および制限

- IPA が最適化したバインド済みのプログラムまたはサービス・プログラムには、UPDPGM または UPDSRVPGM を使用できません。
- IPA が最適化したプログラムまたはサービス・プログラムを、通常のソース・デバッグ機能を使用してデバッグすることはできません。この理由は、IPA が IL データ内にデバッグ情報を保持しないからです。実際には、出力区画の生成時にすべてのデバッグ情報が破棄されます。したがって、ソース・デバッガーは、IPA プログラムまたはサービス・プログラムを処理しません。
- 出力区画には 10,000 個という限界があります。この限界に達すると、バインドが失敗し、システムはメッセージを送信します。この限界に到達した場合には、CRTPGM または CRTSRVPGM コマンドを再び実行して、より大きな区画サイズを指定してください。172 ページの『IPA 制御ファイルの構文』の partition ディレクティブを参照してください。
- プログラムに SQL データが含まれている場合、プログラムに適用される可能性がある、IPA に関するいくつかの制限があります。使用するコンパイラーが、IL データを保持するオプションを許可している場合、これらの制限は適用されません。使用するコンパイラーが、IL データを保持するオプションを許可しない場合、SQL データを含んでいるプログラムに IPA を使用するには、下記のステップを実行する必要があります。たとえば、SQL ステートメントが組み込まれてい

る C プログラムがあるとします。通常は、このソースを CRTSQLCI コマンドを使用してコンパイルしますが、このコマンドには MODCRTOPT(*KEEPILDTA) オプションがありません。

以下のステップを実行して、SQL データと IL データの両方が組み込まれた *MODULE を作成してください。

1. SQL C ソース・ファイルを CRTSQLCI コマンドでコンパイルします。OPTION(*NOGEN) および TOSRCFILE(QTEMP/QSQLTEMP) コンパイラー・オプションを指定してください。このステップで、SQL ステートメントがプリコンパイルされ、SQL プリコンパイラー・データが、元のソース・ファイルの関連するスペースに置かれます。このステップではさらに、C ソースが、一時ソース物理ファイル QTEMP/QSQLTEMP 内の、同じ名前を持つメンバーに置かれます。
2. コンパイラー・コマンドに MODCRTOPT(*KEEPILDTA) を指定して、QTEMP/QSQLTEMP 内の C ソース・ファイルをコンパイルします。このアクションによって、SQL C *MODULE オブジェクトが作成され、プリプロセッサー・データが、元のソース・ファイルの関連するスペースからモジュール・オブジェクトに伝搬されます。この *MODULE オブジェクトには IL データも含まれています。この時点で、IPA(*YES) パラメーターを指定した CRTPGM または CRTSRVPGM コマンドに *MODULE オブジェクトを指定することができます。

- IPA は、最適化レベル 10 (*NONE) でコンパイルしたモジュールを最適化できません。 IPA には、より高い最適化レベルでのみ使用可能な IL データ内の情報が必要です。
- IPA は、IL データを含んでいないモジュールを最適化できません。このため、IPA は、MODCRTOPT(*KEEPILDTA) オプションを提供するコンパイラーを使用して作成されたモジュールのみを最適化できます。 現在、これには、C および C++ コンパイラーが含まれます。
- 一般的には main 関数であるプログラム・エントリー・ポイントのあるプログラム・モジュールの場合、上記のような正しい属性を持っていなければなりません。さもないと、IPA は失敗します。サービス・プログラムの場合は、エクスポートされた関数が入っているモジュールの少なくとも 1 つが上記のような正しい属性を持っている必要があります。さもないと、IPA は失敗します。プログラムまたはサービス・プログラム内の他のモジュールも正しい属性を持っていることが望ましいのですが、必須ではありません。 IPA は正しい属性を持たないすべてのモジュールを受け入れますが、それらは最適化されません。

IPA によって作成される区画

IPA によって作成される最終的なプログラムまたはサービス・プログラムは、区画から構成されます。IPA は区画ごとに *MODULE を作成します。区画には以下の 2 つの目的があります。

- 区画は、関連するコードを同じストレージ領域に集めることによって、プログラム内の参照の局所性を向上させます。
- 区画は、区画のオブジェクト・コード生成時のメモリー要件を低減します。

区画には以下の 3 つのタイプがあります。

- 初期設定区画。これには初期設定コードおよびデータが含まれます。

- 1 次区画。これには、プログラムの 1 次入り口点に関する情報が含まれます。
- 2 次区画またはその他の区画。

IPA は、各タイプの区画の数を以下の方法で決定します。

- IPACTLFILE パラメーターによって指定された、制御ファイル内の 'partition' ディレクティブ。このディレクティブは、各区画の大きさを指示します。
- プログラム呼び出しグラフ内の接続性。接続性とは、プログラム内の関数間の呼び出しの量のことです。
- 異なるコンパイル単位に指定されたコンパイラー・オプション間の競合の解決。IPA は、すべてのコンパイル単位に共通のオプションを適用することによって、競合を解決します。この方法で解決できない場合には、元のオプションの効果が個別の区画で維持されるようなコンパイル単位を強制します。

  このような例の 1 つは、*ライセンス内部コード・オプション* (LICOPT) です。2 つのコンパイル単位の LICOPT が競合している場合、IPA は、このようなコンパイル単位からの関数を同じ出力区画に組み合わせることができません。区画マップ (Partition Map) リスト・セクションの例については、 201 ページの『区画マップ (Partition Map)』 を参照してください。 IPA は一時ライブラリーに区画を作成し、関連する *MODULE をバインドして、最終的なプログラムまたはサービス・プログラムを作成します。IPA は、ランダムな接頭部を使用して区画の *MODULE 名を作成します (たとえば、QD0068xxxx。この xxxx の範囲は 0000 ～ 9999)。

  このため、DSPPGM または DSPSRVPGM 内のいくつかのフィールドが実行されない可能性があります。「プログラム入り口プロシージャー・モジュール」は、元の *MODULE 名ではなく、*MODULE 区画名を示します。そのモジュールの「ライブラリー」フィールドは、元のライブラリー名ではなく、一時ライブラリー名を示します。さらに、プログラムまたはサービス・プログラムにバウンダリーされた各モジュールの名前は、生成された区画名になります。 IPA によって最適化されたプログラムまたはサービス・プログラムの場合、DSPPGM または DSPSRVPGM によって表示される「プログラム属性」フィールドは、そのプログラムまたはサービス・プログラムのすべてのバインド済みモジュールの属性フィールドと同様で、IPA になります。

注: IPA が区画への分割を実行している場合、IPA は関数名またはデータ名の接頭部として @nnn@ または XXXX@nnn@ を付ける場合があります。ここで、XXXX は区画名、また nnn はソース・ファイル番号です。これによって、静的関数名および静的データ名の固有性が維持されます。

ライセンス内部コードのオプション

ライセンス内部コード (LIC)・オプション (LICOPT) はコンパイラー・オプションであり、コードの生成方法やパッケージ方法を変えるために、ライセンス内部コードに渡されます。渡されるコンパイラー・オプションは、モジュール、ILE プログラム・オブジェクト、あるいはコンパイルされた Java プログラム用に生成されたコードを対象にします。一部のオプションは、コードの最適化を微調整するために使用することができます。一部のオプションは、プログラムのデバッグに使用することができます。このセクションでは、ILE に関連するライセンス内部コードのオプションについてのみ説明します。 Java についての詳細は、CRTJVAPGM コマンド

の LICOPT パラメーターに関するオンライン・ヘルプ情報を参照してください。その他の情報源としては、Information Center の IBM Developer Kit for Java トピックがあります。

現在定義されているオプション

現在、ILE 用に定義されているライセンス内部コード・オプションは以下のとおりです。

[No]AlwaysTryToFoliate

最適化レベル 40 でコンパイルしている場合に、実行時呼び出しスタックに保持されているスタック・フレーム数の削減を試みる、**呼び出しフォーリエーション (call foliation)** と呼ばれる最適化を、より積極的に試みるように、最適化変換プログラムに伝えます。この利点は、必要なスタック・フレームがより少なくて済む場合があり、その場合、参照の局所性が向上し、まれには、実行時のスタック・オーバーフローの可能性を低減することができます。欠点は、プログラム障害が発生した場合のデバッグ時に、呼び出しスタックに残っている情報が少なくなる可能性があります。このオプションは、デフォルトではオフになっています。

[No]CallTracingAtHighOpt

このオプションを使用すれば、最適化レベルが 40 の場合でも、スタックが必要なプロシージャーのプロローグとエピローグにそれぞれ、呼び出しと戻りのトラップを挿入するように要求することができます。デフォルトでは、呼び出しと戻りのトラップは、最適化レベルが 40 の場合には、プロシージャーに挿入されません。呼び出しと戻りのトラップを挿入する利点は、ジョブ・トレース (TRCJOB) を使用できるようになることであり、欠点は実行時のパフォーマンスが低下する可能性があることです。 このオプションは、デフォルトではオフになっています。

[No]Compact

このオプションを使用すれば、可能な場合に、実行速度を犠牲にして、コード・サイズを削減することができます。これは、コードを複製または拡張してインライン化するという最適化を禁止することによって行われます。このオプションは、デフォルトではオフになっています。

[No]CreateSuperblocks

このオプションは、スーパーブロックの形成を制御するものです。スーパーブロックとは、大規模な拡張基本ブロックのことであり、スーパーブロック・ヘッダー以外に制御フロー項目が含まれません。このオプションはまた、トレース・アンローリングやトレース・ピーリングなどのような、スーパーブロックで行われる特定の最適化も制御します。スーパーブロックの形成や最適化が行われると、大量のコードが重複する可能性があります。この LICOPT を使用することにより、こうした最適化を使用不可にすることができます。この LICOPT は、プロファイル・データが適用されているときにのみ有効です。このオプションは、デフォルトではオンになっています。

[No]DetectConvertTo8BytePointerError

このオプションはテラスペース・ストレージ・モデル・プログラムにのみ適用されます (たとえば、コンパイルされるソース言語に対する CRT コマンドで `STGMDL(*TERASPACE)` が使用されている場合)。このオプションを使用すると追加のコードが生成され、16 バイト・ポインターから 8 バイト・ポインターへの変

換の一部として、実行時に 16 バイト・ポインターに単一レベル・ストア (SLS) ・アドレスが入っていれば検出します。8 バイト・ポインターがポイントできるのはテラスペースのみであるため、SLS アドレスは 8 バイト・ポインターには保管されません。 16 バイト・ポインターから 8 バイト・ポインターへの変換に対して SLS 検出が有効になっていない場合に 16 バイト・ポインターに SLS アドレスが入っていると、その後の 8 バイト・ポインターの使用で参照されるテラスペース内のロケーションが一定でないか、MCH0601 例外が発生する可能性があります。それとは逆に、検出できるようになっていれば、MCH0609 例外により、明確に問題が知らされます。この検出は、SLS および継承ストレージ・モデル・プログラムではデフォルトで有効になります。テラスペース・ストレージ・モデル・プログラムでは、この検出はプログラム呼び出しの一部として呼び出されるプログラム・エントリー・プロシージャー (PEP) 内でのみ、デフォルトで有効になります。ただし、他のプロシージャー内では有効にはなりません。

ある特定のポインター変換操作は、ポインターの変換を行なうための言語組み込み関数として、テラスペース・アドレスの検索 (retrieve teraspace address (RETTSADR)) マシン・インターフェース命令を使用することによっても検出できます。

このオプションは、デフォルトではオフになっています。

[No]EnableInlining

このオプションは、最適化変換プログラムにより、プロシージャーのインライン化を制御します。プロシージャーのインライン化とは、プロシージャーへの呼び出しをプロシージャー・コードのインライン・コピーによって置き換えることです。このオプションは、デフォルトではオンになっています。

[No]FoldFloat

コンパイル時にシステムが浮動小数点の定数式の値を求めることが可能であることを指定します。この LICOPT は、「浮動小数点定数フォールディング」モジュール作成オプションを指定変更します。この LICOPT を指定しない場合には、このモジュール作成オプションが使用されます。

LoopUnrolling=<option>

LoopUnrolling オプションは、最適化変換プログラムによって実行されるループ・アンローリングの量を制御するために使用します。有効な値は、0 (ループ・アンローリングを使用不可にする)、1 (コードの重複を削減することを重点にして小規模なループをアンロールする)、および 2 (ループを積極的にアンロールする) です。オプション 2 を使用すると、生成されるコードのサイズが大幅に増大する可能性があります。デフォルト値は 1 です。

[No]Maf

浮動小数点の乗算・加算命令の生成が可能です。この命令は、乗算と加算の演算を結合させて、中間結果の丸め操作を行わないようにしたものです。実行パフォーマンスは改善されますが、計算結果に影響を与えることがあります。この LICOPT は、「乗算・加算使用」モジュール作成オプションを指定変更します。この LICOPT を指定しない場合には、このモジュール作成オプションが使用されます。

[No]MinimizeTeraspaceFalseEAOs

現在のハードウェアのロード命令と保管命令は、16 MB 境界超え、すなわち、**有効アドレス・オーバーフロー** (EAO) を検出します。EAO の検査は、アドレ

ス算術演算の一部としても実行されます。生成された同じコードが、テラスペース・アドレスと単一レベル・ストア (SLS) ・アドレスの両方を処理する必要があるので、有効なテラスペースの使用が、偽の EAO を生じる可能性があります。このような EAO 条件は問題を示していませんが、このような条件の処理によって、処理オーバーヘッドが大幅に増加します。
MinimizeTeraspaceFalseEAOs LICOPT によって、プログラムのために生成されるハードウェア命令シーケンスに相違が生じます。まず、通常の場合には少し遅くなりますが、大部分の EAO の発生を除去する、別のアドレス演算命令シーケンスが生成されます。さらに、通常の場合にはより速いコードを生成するが、偽の EAO の頻度が増加する可能性がある特定の最適化が禁止されます。この LICOPT を使用する必要がある場合の例は、モジュール内で実行される大部分のアドレス演算で、共通基底アドレスからのテラスペース・アドレスとオフセット値の加算結果が 16 MB より頻繁に大きくなる場合です。このオプションは、デフォルトではオフになっています。

[No]OrderedPtrComp
このオプションを使用すれば、符号なし整数値としてポインターを比較し、順序付け (等しい、より小、より大) の結果を常に生成することができます。このオプションを使用した場合、異なるスペースを参照するポインターは、順序付け不能で比較されません。このオプションは、デフォルトではオフになっています。

[No]PredictBranchesInAbsenceOfProfiling
プロファイル作成データが提供されていない場合、このオプションを使用して、コードの最適化をガイドする静的ブランチ予測を実行します。プロファイル作成データが提供されている場合には、このオプションの指定に関係なく、ブランチの可能性を予測するために、プロファイル作成データが使用されます。このオプションは、デフォルトではオフになっています。

[No]PtrDisjoint
このオプションは、積極的な入力ベースの別名の精錬を可能にします。これにより最適化変換プログラムが冗長な負荷を大幅に除去できるようになり、ランタイム・パフォーマンスを改善できます。ポインターの内容が非ポインター・タイプを通してアクセスされない場合は、アプリケーションは安全にこのオプションを使用できます。以下の C の式は、ポインターの値に対する、アンセーフなアクセス方法を示したものです。

```
void* spp;
... = ((long long*) &spp) [1]; // Access low order 8 bytes of 16-byte pointer
```

デフォルト: NoPtrDisjoint

TargetProcessorModel=<option>
targetProcessorModel オプションは、指定されたプロセッサー・モデルに対する最適化を行なうよう変換プログラムに指示します。このオプションを指定して作成されたプログラムは、サポートされるすべてのハードウェア・モデルで実行できますが、指定のプロセッサー・モデルではより速く実行されます。有効な値は、スター型プロセッサーの場合は 0、 POWER4™ プロセッサーの場合は 2 です。デフォルトは、プログラム・オブジェクトに関連するターゲット・リリースによって異なります。 V5R2M0 以降、デフォルト値は 2 です。それより前のリリースでは、デフォルト値は 0 です。

それぞれのオプションごとに、肯定形と否定形の差異があることに注意してください。否定形は接頭部 'no' で始まります。否定形のバリアントは、そのオプションが適用されないことを意味します。ブール・オプションには必ずこのように 2 つのバリアントがあり、オプションをオンにするだけでなく、明示的にオフにすることができます。デフォルト・オプションがオンのものについては、オフにする機能が必要です。 オプションのデフォルトは、いずれもリリースによって変更されることがあります。

アプリケーション

プログラムをコンパイルする時に、コンパイラー・オプションとしてライセンス内部コード・オプションを指定することができます。

オプションを既存のモジュールに適用するために、CHGMOD (モジュールの変更)、CHGPGM (プログラム変更)、および CHGSRVPGM (サービス・プログラムの変更) コマンドで、これらのオプションを指定することもできます。コマンドでのパラメーター名は、LICOPT です。 以下は、ライセンス内部コードのオプションをモジュールに適用する例です。

```
> CHGMOD MODULE(TEST) LICOPT('maf')
```

CHGPGM または CHGSRVPGM で使用する場合は、システムは指定されたライセンス内部コードのオプションを、ILE プログラム・オブジェクトに含まれるすべてのモジュールに適用します。 以下は、ライセンス内部コードのオプションを ILE プログラム・オブジェクトに適用する例です。

```
> CHGPGM PGM(TEST) LICOPT('nomaf')
```

以下は、ライセンス内部コードのオプションをサービス・プログラムに適用する例です。

```
> CHGSRVPGM SRVPGM(TEST) LICOPT('maf')
```

制約事項

ライセンス内部コードのオプションを適用するプログラムやモジュールのタイプにはいくつかの制限があります。

- OPM プログラムにはライセンス内部コードのオプションを適用できません。
- モジュール、ILE プログラム、またはサービス・プログラムのオブジェクトは、初めからリリース V4R5M0 以降用に作成されたものでなければなりません。
- V4R5 以降のプログラムまたはサービス・プログラム内で、V4R5 より前の結合モジュールにライセンス内部コードのオプションを適用することはできません。このことは、プログラム内で LICOPT が適用された他の結合モジュールには影響しません。

構文

CHGMOD、CHGPGM、および CHGSRVPGM コマンドで、LICOPT パラメーター値の大文字小文字は区別しません。たとえば、以下の 2 つのコマンド呼び出しは同じ結果になります。

```
> CHGMOD MODULE(TEST) LICOPT('nomaf')
> CHGMOD MODULE(TEST) LICOPT('NoMaf')
```

複数のライセンス内部コード・オプションを一緒に指定する場合は、オプションをコンマで区切る必要があります。また、システムは、オプションの前後のスペースをすべて無視します。以下に例を挙げます。

```
> CHGMOD MODULE(TEST) LICOPT('Maf,NoFoldFloat')
> CHGMOD MODULE(TEST) LICOPT('Maf,   NoFoldFloat')
> CHGMOD MODULE(TEST) LICOPT('   Maf   ,   NoFoldFloat   ')
```

ブール・オプションの場合、2 つの相反するバリアントを同時に指定することはできません。たとえば、以下のようなコマンドは指定できません。

```
> CHGMOD MODULE(TEST) LICOPT('Maf,NoMaf')   <- 無効です!
```

ただし、同じオプションを複数回指定することはできます。たとえば、以下のものは有効です。

```
> CHGMOD MODULE(TEST) LICOPT('Maf, NoFoldFloat, Maf')
```

リリースの互換性

このシステムでは、ユーザーは、ライセンス内部コードのオプションを適用済みのモジュール、ILE プログラム、およびサービス・プログラムを、V4R5M0 より前のどのリリースにも移すことは許可されません。実際に、オブジェクトをメディアまたは保管ファイルに保管しようとした場合、このシステムでは以前のターゲット・リリースを指定することはできません。

OS/400 は、今後のリリースで (あるいは、PTF を適用した所定のリリース内で)、新規のライセンス内部コードのオプションを定義する場合があります。新規のオプションは、それをサポートする最初のリリース、あるいはそれ以降のリリースのシステムで使用することができます。新規のオプションが適用されているモジュール、ILE プログラム、およびサービス・プログラムはいずれも、そのオプションをサポートしていないリリースへ移すことができます。ただし、そのリリースは V4R5M0 以降でなければなりません。コマンドの LICOPT パラメーターでそのオプションが指定されていない場合は、システムはそれを無視し、サポートされていないライセンス内部コードのオプションを再変換されたオブジェクトに適用することはありません。システムが CHGMOD、CHGPGM、または CHGSRVPGM コマンドで LICOPT(*SAME) を使用してオブジェクトを再作成する場合には、このタイプの再作成が行われる可能性があります。また、システムがオブジェクトを自動的に変換する場合にも、このタイプの再作成が行われます。これによって再作成が妨げられることはありません。逆に、CHGMOD、CHGPGM、あるいは CHGSRVPGM コマンドの LICOPT パラメーターで、サポートされていない同じオプションを指定しようとするといずれも失敗します。

モジュールおよび ILE プログラムのライセンス内部コード・オプションの表示

DSPMOD、DSPPGM、および DSPSRVPGM コマンドは、適用されたライセンス内部コードのオプションを表示します。 DSPMOD は「モジュール情報」セクションでそのオプションを表示します。たとえば、次のとおりです。

```
ライセンス内部コード・オプション  . . . . . :   maf
```

DSPPGM および DSPSRVPGM は、プログラム内のそれぞれ独立したモジュールに適用されているライセンス内部コードのオプションを、「モジュール属性」セクションで各モジュールごとに表示します。

同じライセンス内部コードのオプションを何度も指定すると、最後のものを除いてすべて、そのオプションの頭に '+' 記号が付けられます。たとえば、あるモジュール・オブジェクトにライセンス内部コードのオプションを適用するために、以下のようなコマンドを使用するとします。

```
> CHGMOD MODULE(TEST) LICOPT('maf, maf,   Maf')
```

これによって、DSPMOD は以下のように表示します。

```
    ライセンス内部コード・オプション  . . . . . :   +maf,+maf,Maf
```

'+' は、ユーザーが同じオプションを重複して指定したことを示します。

ライセンス内部コードのオプションの頭に '*' 記号が付いて表示される場合は、そのオプションがモジュールやプログラムに適用されることはありません。これは、オブジェクトの再作成を最後に行ったシステムでは、そのオプションがサポートされていなかったためです。詳細については 182 ページの『リリースの互換性』のセクションを参照してください。たとえば、以下のコマンドを使用して、新しいオプションが最初に N+1 リリースのシステムに適用されたとします。

```
> CHGMOD MODULE(TEST) LICOPT('NewOption')
```

モジュールをそのオプションをサポートしていないリリース N のシステムに戻し、その後、次のコマンドを使用してモジュール・オブジェクトをそこで再作成します。

```
> CHGMOD MODULE(TEST) FRCCRT(*YES) LICOPT(*SAME)
```

DSPMOD で表示されるライセンス内部コードのオプションは以下のようになります。

```
    ライセンス内部コード・オプション  . . . . . :   *NewOption
```

'*' は、そのオプションがモジュールに適用されないことを意味します。

第 14 章 共用ストレージの同期

共用ストレージは、並行して実行している複数のスレッド間の通信に対し、効率のよい手段を提供します。この章では、共用ストレージに関連する多くの問題について説明しています。説明の主な焦点は、共用ストレージにアクセスする場合に生じる可能性があるデータの同期の問題と、その解決方法です。

共用ストレージに関連するプログラミングの問題は ILE に特有の問題ではありませんが、オリジナルの MI 言語よりも高い頻度で生じる可能性があります。これは、ILE におけるマルチプログラミングのアプリケーション・プログラミング・インターフェースに関する広範なサポートによります。

共用ストレージ

ここでの説明で*共用ストレージ* という用語は、複数のスレッドからアクセスされる何らかのスペース・データを指しています。この定義には、個々のバイトにまで直接アクセス可能なストレージが含まれ、また以下のクラスのストレージを含めることができます。

- MI スペース・オブジェクト
- 他の MI オブジェクトの 1 次関連スペース
- POSIX 共用メモリー・セグメント
- 暗黙のプロセス・スペース、すなわち、自動ストレージ、静的ストレージ、および活動ベースのヒープ・ストレージ
- テラスペース

これらのスペースの存続期間に関係なく、並行処理が可能な複数のスレッドによってアクセスされると、システムはこれらのスペースを共用ストレージであると想定します。

共用ストレージの問題

共用ストレージを活用するアプリケーションを作成する場合には、予期しないデータ値をもたらす 2 つのタイプの問題、すなわち、*競合状態* とストレージ・アクセス*順序付け問題* を回避しなければなりません。

- 競合状態は、プログラムの種々の結果が連携して機能する 2 つまたはそれ以上のスレッドの相対的なタイミングにのみ依存する可能性がある場合に生じます。

  競合状態は、予測可能で、しかも正しく機能するよう、相互作用するスレッドの処理を同期化することにより避けることができます。本章では、ストレージの同期化を中心に説明していますが、スレッド実行の同期化とストレージの同期化の技法は、大きな範囲で重なり合っています。このような理由で、本章で後述する問題の例では、競合状態については手短に触れています。
- ストレージ・アクセス順序付け問題は、ストレージ同期またはメモリー整合性の問題としても知られています。このような問題が生じるのは、連携して機能する 2 つまたはそれ以上のスレッドが、共用ストレージに対する更新が特定の順序で

行われることに依存し、しかも各ストレージへのアクセスが同期化されていない場合です。たとえば、あるスレッドが 2 つの共用変数に値を保管し、別のスレッドが、それらの値の更新を特定の順序で監視することに暗黙的に依存する場合です。

共用ストレージ・アクセスの順序付けの問題は、共用ストレージを読み書きするスレッドに対して、システムがストレージ同期化のアクションを確実に行うことにより回避することができます。これらのアクションのいくつかについて、以下のトピックで説明します。

共用ストレージ・アクセスの順序付け

複数のスレッドがストレージを共用する場合、1 つのスレッドによって行われた共用ストレージのアクセス (読み取りおよび書き込み) を、他のスレッドが行われた順序で監視できる保証はありません。このことは、共用ストレージに対して読み取りまたは書き込みを行うスレッドにある形式の明示的なストレージの同期化を行わせることによって、防止することができます。

ストレージの同期が必要なのは、2 つまたはそれ以上のスレッドが共用ストレージに並行してアクセスを試み、しかもそれらのスレッドのロジックが、共用ストレージへのアクセスに一定の順序付けが必要であることを意味している場合です。共用ストレージの更新が監視される順序が重要でない場合、ストレージの同期は必要ではありません。1 つのスレッドは常にそれ自体のストレージ (共用または非共用のストレージ) の更新を順序どおりに監視します。オーバーラップしている共用ストレージのロケーションをアクセスするすべてのスレッドは、同じ順序でそれぞれのアクセスを監視するはずです。

競合状態とストレージのアクセス順序付けの問題が、どのように予期しない結果を生むかを示している次の単純な例について考えてみます。

```
                volatile int X = 0;
                volatile int Y = 0;
  スレッド A                          スレッド B
--------------                    -------------
    Y = 1;                          print(X);
    X = 1;                          print(Y);
```

以下の表は、スレッド B によって印刷される可能性がある結果を要約しています。

X	Y	問題のタイプ	説明
0	0	競合状態	スレッド B はスレッド A による変更に先立って変数を読み取っている
0	1	競合状態	スレッド B は、Y に対する更新は認知しているが、スレッド A による X の更新を認知する前に X を印刷している
1	1	競合状態	スレッド B はスレッド A による更新の後で両方の変数を読み取っている

X	Y	問題のタイプ	説明
1	0	ストレージ・アクセス順序付け	スレッド B はスレッド A による X の更新を識別しているが、Y に対する更新はまだ認知していない。明示的なデータ同期化のアクションを行わないと、このような順不同のストレージ・アクセスが生じる可能性があります。

問題の例 1: 1 つの書き込みと複数の読み取り

通常、順序の狂った共用ストレージ・アクセスの可能性は、マルチスレッドのプログラム・ロジックの正確性に影響しません。ただし、場合によっては、スレッドが他のスレッドによるストレージの更新を認知する順序が、プログラムの正確さにとって重大であることがあります。

ある形式の明示的なデータ同期化を必要とする典型的な場合を考えてみます。共用ストレージのあるロケーション (オーバーラップしていない) へのアクセスの制御に、共用ストレージの他のロケーションの状態が使用される (プログラム・ロジックの規則に基づいて) 場合です。たとえば、1 つのスレッドがいくつかの共用データ (DATA) を初期化すると想定します。さらに、そのスレッドは次に共用フラグ (FLAG) を設定して、他のすべてのスレッドに対して共用データが初期化されていることを示すものと想定します。

```
初期化するスレッド                  その他のすべてのスレッド
-------------------                 --------------------------
     DATA = 10                      FLAG の値が 1 になるまでループ
     FLAG = 1                       DATA を使用する
```

このような場合、共用を行うスレッドは、共用ストレージのアクセスに関して順序の強制が必要になります。これを行わないと、他のスレッドは、初期化を行うスレッドによる共用ストレージの更新を正しくない順序で認知することがあります。これにより、他のスレッドの一部またはすべてが、DATA から消去されていない値を読み取ることになります。

例 1 のソリューション

前述の例の問題を解決する望ましい方式は、データとフラグの値との間の依存性を完全に回避することです。より堅固なスレッド同期化スキームを使用して、これを行うことができます。スレッド同期化の多くの技法を採用することができますが、この問題に最も適しているのはセマフォーの使用です。 (セマフォーに対するサポートは、AS/400 バージョン 3 リリース 2 から使用可能です。)

以下のロジックは、次の想定に該当する場合に当てはまります。

- プログラムは、連携するスレッドの開始に先立ってそのセマフォーを作成している。
- プログラムは、そのセマフォーを 1 のカウントに初期化している。

```
初期化するスレッド                  その他のすべてのスレッド
-------------------                 --------------------------
DATA = 10                           セマフォー・カウントが 0 になるまで待機
セマフォーを減算する                DATA を使用する
```

ストレージ同期化のアクション

共用ストレージのアクセスの順序付けが必要な場合、順序付けの制約が必要なすべてのスレッドは、明示的なアクションを行って、共用ストレージ・アクセスを同期化しなければなりません。このようなアクションを、*ストレージ同期化アクション*と呼びます。

スレッドで行われる同期化アクションにより、そのスレッドのロジック・フローのコードでその同期化アクションよりも前に現れる共用ストレージ・アクセスは、その同期化アクションよりも後に現れる共用ストレージ・アクセスに先立って完了することが保証されます。これは、他のスレッドにおける同期化アクションについて当てはまります。言い換えると、1 つのスレッドが 2 つの共用ロケーションに 2 つの書き込みを行い、しかもそれらの書き込みが同期化アクションで分離されている場合、システムは以下を行います。すなわち、最初の書き込みは、次の同期化アクションの時点以前で、しかも 2 番目の書き込みが使用可能になる時点に先立って、他のスレッドに対して使用可能になることが保証されます。

2 つの共用ロケーションからの 2 つの読み取りがストレージの同期化アクションで分離されている場合には、2 番目の読み取りは最初の読み取りとほとんど同じ時点の値を読み取ります。これは、他のスレッドが共用ストレージに書き込みを行う際に順序付けを強制する場合にのみ当てはまります。

以下のスレッドの同期化アクションは、ストレージ同期化アクションでもあります。

メカニズム	同期化アクション	使用可能な最初の VRM
オブジェクト・ロック	ロック、アンロック	すべて
スペース・ロケーションのロック	ロック、アンロック	すべて
Mutex	ロック、アンロック	V3R1M0
セマフォー	ポスト、待機	V3R2M0
スレッド 条件	待機、シグナル、ブロードキャスト	V4R2M0
データ待ち行列	エンキュー、デキュー	すべて
メッセージ待ち行列	エンキュー、デキュー	V3R2M0
比較交換	ターゲットへの正常な保管	V3R1M0
ロック値の検査 (CHKLKVAL)	ターゲットへの正常な保管	V5R3M0
ロック値のクリア (CLRLKVAL)	常時	V5R3M0

さらに、次の MI 命令は、ストレージ同期化アクションを構成しますが、同期化スレッドには使用できません。

メカニズム	同期化アクション	使用可能な最初の VRM
SYNCSTG MI 命令	常時	V4R5M0

複数のスレッド間で共用ストレージ・アクセスの順序付けを完全に行うためには、アクセスの順序付けに依存するすべてのスレッドで適切な同期化アクションを使用する必要があるということを忘れないでください。 これは、共用データの読み取りおよび書き込みの両方に当てはまります。 読み取りと書き込み間のこの合意により、基盤のマシンで用いる最適化がどのようなものであっても、アクセスの順序はそのままで、変わることはありません。

問題の例 2: 2 つの競合する書き込みまたは読み取り

同期化を必要とする共通する他の問題は、以下の例のように、複数のスレッドが形式どおりでないロック・プロトコルの強制を試みる場合の問題です。この例では、2 つのスレッドが共用ストレージのデータを操作しています。両方のスレッドは、アクセスを逐次化する目的で共用フラグを使用して、2 つの共用データ項目の読み取りと書き込みを繰り返して試みます。

```
                スレッド A                                    スレッド B
  ------------------------------------          ------------------------------------
 /* 共用データに対し何らかの作業を行う */          /* 共用データに対し何らかの作業を行う */
  for (int i=0; i<10; ++i) {                    for (int i=0; i<10; ++i) {
    /* ロックされたフラグがクリアされるまで待機 */     /* ロックされたフラグがクリアされるまで待機 */
    while (locked == 1) {                         while (locked == 1) {
     sleep(1);                                      sleep(1);
    }                                            }

    locked = 1;  /* ロックを設定する */              locked = 1;     /* ロックを設定する */

   /* 共用データを更新する */                      /* 共用データを更新する */
   data1 += 5;                                   data1 += 4;
   data2 += 10;                                  data2 += 6;

    locked = 0;  /* ロックをクリアする */           locked = 0;     /* ロックをクリアする */
  }                                            }
```

この例は、共用メモリーに関する 2 つの問題を示しています。

競合状態

ここで使用されているロック・プロトコルは、データ競合状態を回避していません。両方のジョブは同時に、ロック・フラグがクリアの状態であると認識することがあり、その結果、両方のジョブはデータを更新するロジックに進むことがあります。そのような場合、どのようなデータ値が読み取られ、増分され、書き込まれるかについては何らの保証もありません。種々の結果が生じる可能性があります。

ストレージ・アクセスの順序付けの問題

上述の競合状態は、しばらく無視してください。両方のジョブによって使用されているロックと共用データの更新のロジックには、フィールドの更新についての暗黙の順序付けに関する前提条件が含まれている点に注意してください。特に、各スレッドは、他のスレッドはデータに対する変更を認識する前に、ロック・フラグは 1 に設定されているのを認識するはずであるという想定を前提としています。さらに、各スレッドは、0 のロック・フラグの値を認識する前にデータの変更を認識するはずであるという想定を前提としています。本章で前述したように、このような前提は無効です。

例 2 のソリューション

競合状態を回避し、ストレージ・アクセスの順序付けを強制するには、上で列挙した同期化メカニズムのいずれかによって共用データへのアクセスを逐次化しなければなりません。複数のスレッドが共用リソースを競合するこの例の場合には、何らかの形式のロックの使用が適しています。以下では、スペース・ロケーションのロックを使用するソリューションについて説明し、その後で比較交換 (compare-and-swap) のメカニズムによる別のソリューションを示しています。

```
            スレッド A
-------------------------------------
for (i=0; i<10; ++i) {
  /* 共用データの排他ロックを取得する。
     ロックが認可されるまで待機状態に
     なる。 */
  locksl( LOCK_LOC, _LENR_LOCK );

  /* 共用データを更新する */
  data1 += 5;
  data2 += 10;

  /* 共用データをアンロックする */
  unlocksl( LOCK_LOC, _LENR_LOCK );
}
```

```
            スレッド B
-------------------------------------
 for (i=0; i<10; ++i) {
/* 共用データの排他ロックを取得する。
   ロックが認可されるまで待機状態に
   なる。*/
locksl( LOCK_LOC, _LENR_LOCK );

   /* 共用データを更新する */
   data1 += 4;
   data2 += 6;

/* 共用データをアンロックする */
unlocksl( LOCK_LOC, _LENR_LOCK );
}
```

ロックを用いて共用データへのアクセスを制限することにより、 1 時点で該当のデータにアクセスできるのは、確実に 1 つだけのスレッドになります。これで、競合状態は解決します。共用ストレージの 2 つのロケーション間の順序付けに関する依存性はもはや存在しないので、このソリューションは、ストレージ・アクセスの順序付けの問題も解決しています。

代替ソリューション: ロック値の検査/ロック値のクリアの使用

最初のソリューションで使用されたスペース・ロケーションのロックには、このような単純な例では必要としない多くの機能が含まれています。たとえば、スペース・ロケーションのロックは、ある時間間隔内でそのロックを入手できない場合に、処理の再開を許すタイムアウトの値をサポートしています。また、スペース・ロケーションのロックは、共用ロックのいくつもの組み合わせをサポートしています。これらは重要な機能ですが、ある程度のパフォーマンスのオーバーヘッドの犠牲を伴います。

代替方法として、*ロック値の検査* および*ロック値のクリア* を使用する方法があります。これら 2 つの MI 命令を一緒に使用すると、特にロックの競合がそれほど多くない場合には、非常に簡単で高速なロック手順を実装することができます。

このソリューションでは、システムは CHKLKVAL を使用してロックの取得を試みます。 (ロックがすでに使用中であることをシステムが検出したために) その試みが失敗すると、スレッドは、しばらく待機してから再度取得を試みることを、ロックが取得できるまで繰り返します。システムは、共用データを更新した後で、CLRLKVAL を使用してロックを解放します。この例では、スレッドが共用データ項目に加えて、8 バイトのロケーションによるアドレスも共用しているものと想定しています。このコードでは、そのロケーションを変数 LOCK で参照しています。さらに、ロックは、静的初期化またはなんらかの同期化前初期化によりゼロに初期設定されていると想定しています。

```
            スレッド A
---------------------------------
/* 共用データに対し何らかの作業を行う */
for (i=0; i<10; ++i) {

  /* Attempt to acquire the lock using
     CHKLKVAL.  By convention, use value
     1 to indicate locked, 0 to indicate
     unlocked. */
  while (_CHKLKVAL(&LOCK, 0, 1) == 1) {
    sleep(1);  /* wait a bit and try again */
  }

  /* 共用データを更新する */
  data1 += 5;
  data2 += 10;

  /* Unlock the shared data.  Use of
     CLRLKVAL ensures other jobs/threads
     see update to shared data prior to
     release of the lock. */
  _CLRLKVAL(&LOCK, 0);
}
```

```
            スレッド B
---------------------------------
/* 共用データに対し何らかの作業を行う */
for (i=0; i<10; ++i) {

        /* Attempt to acquire the lock using
           CHKLKVAL.  By convention, use value
           1 to indicate locked, 0 to indicate
           unlocked. */
        while (_CHKLKVAL(&LOCK, 0, 1) == 1) {
          sleep(1);  /* wait a bit and try again */
        }

  /* 共用データを更新する */
  data1 += 4;
  data2 += 6;

        /* Unlock the shared data.  Use of
           CLRLKVAL ensures other jobs/threads
           see update to shared data prior to
           release of the lock. */
        _CLRLKVAL(&LOCK, 0);
      }
```

ここで、スレッドは、ロック値の検査を使用してロック変数が競合していないかどうかをテストし、更新を行ってから、ロック値のクリアを使用してロック変数をアンロック状態にリセットしています。これは、当初の問題で経験した競合状態を解決します。また、ストレージ・アクセスの順序付けの問題も処理しています。すでに述べたように、ロック値の検査およびロック値のクリアをこのような方法で使用すると、アクションが同期化されます。共用データの読み取りに先立ってロック値の検査を使用してロックを設定すると、スレッドが最新の更新データを必ず読み取るようになります。共用データの更新後に、ロック値のクリアを使用してロックをクリアすると、その更新は、次に同期化アクションが行われた後で、どのスレッドでも読み取れるようになります。

付録 A. CRTPGM、CRTSRVPGM、UPDPGM、または UPDSRVPGM コマンドからの出力リスト

この付録では、バインド・プログラム (バインダー) のリストの例を示し、バインド・プログラム言語の使用の結果として起こり得るエラーについて説明します。

バインド・プログラムのリスト

プログラムの作成 (CRTPGM)、サービス・プログラムの作成 (CRTSRVPGM)、プログラムの更新 (UPDPGM)、およびサービス・プログラムの更新 (UPDSRVPGM) の各コマンドのバインド・プログラムのリストは、ほとんど同じです。この付録では 95 ページの『バインド・プログラム言語の例』のサービス・プログラム FINANCIAL の作成に使用された CRTSRVPGM コマンドからのバインド・プログラムのリストを示します。

CRTPGM、CRTSRVPGM、UPDPGM、または UPDSRVPGM の各コマンドの DETAIL パラメーターには、次の 3 つのタイプのリストを指定することができます。

*BASIC
*EXTENDED
*FULL

基本リスト

CRTPGM、CRTSRVPGM、UPDPGM、または UPDSRVPGM の各コマンドに DETAIL(*BASIC) を指定すると、そのリストは以下によって構成されます。

- CRTPGM、CRTSRVPGM、UPDPGM、または UPDSRVPGM コマンドに指定された値
- 簡略要約表
- バインディング・プロセスのいくつかの処理に要した時間を示すデータ

図 47、図 48、および 194 ページの図 49 は、これらの情報を示しています。

```
                                サービス・プログラムの作成                      ページ   1

サービス・プログラム . . . . . :   FINANCIAL
  ライブラリー . . . . . . . . . :     MYLIB
エクスポート . . . . . . . . . . :   *SRCFILE
エクスポート・ソース・ファイル . . . . . :   QSRVSRC
  ライブラリー . . . . . . . . . :     MYLIB
エクスポート・ソース・メンバー . . . . . :   *SRVPGM
活動化グループ . . . . . . . . . . :   *CALLER
作成オプション . . . . . . . . . . :   *GEN        *NODUPPROC  *NODUPVAR   *DUPWARN
明細のリスト . . . . . . . . . . :   *FULL
更新可能 . . . . . . . . . . . . :   *YES
バインド済み *SRVPGM ライブラリー名更新可能  . . . :   *NO
ユーザー・プロファイル . . . . . . . . . :   *USER
既存のサービス・プログラムの置き換え . . . . . :   *YES
権限 . . . . . . . . . . . . . . :   *LIBCRTAUT
ターゲット・リリース . . . . . . . . . :   *CURRENT
再初期設定可能 . . . . . . . . . . . :   *NO
ストレージ・モデル . . . . . . . . . :   *SNGLVL
プロシージャー間分析 . . . . . . . . . :   *NO
IPA 制御ファイル . . . . . . . . . . :   *NONE
IPA 置き換え IL データ . . . . . . . . . :   *NO
テキスト . . . . . . . . . . . . . :
モジュール      ライブラリー    モジュール      ライブラリー    モジュール      ライブラリー    モジュール      ライブラリー
MONEY           MYLIB           CALCS           MYLIB
RATES           MYLIB           ACCTS           MYLIB

サービス・      ライブラリー    サービス・      ライブラリー    サービス・      ライブラリー    サービス・      ライブラリー
プログラム                      プログラム                      プログラム                      プログラム
*NONE
バインド・      ライブラリー    バインド・      ライブラリー    バインド・      ライブラリー    バインド・      ライブラリー
ディレクトリー                  ディレクトリー                  ディレクトリー                  ディレクトリー
*NONE
```

図 47. CRTSRVPGM コマンドに指定された値

```
                                サービス・プログラムの作成                      ページ   3
                                      簡略要約表
プログラム入力プロシージャー . . . . . :   0
複数強定義 . . . . . . . . . . . . . :   0
未解決の参照 . . . . . . . . . . . . . :   0

    * * * * *  簡  略  要  約  表  の  終  わ  り  * * * * *
```

図 48. 簡略要約表

```
                                サービス・プログラムの作成                      ページ  23
5722SS1 V5R3M0  040525
                                      バインド統計
記号収集 CPU 時間  . . . . . . . . . . . . . . . . . . . . :              .018
記号分析解決 CPU 時間  . . . . . . . . . . . . . . . . . . :              .006
バインド・ディレクトリー分析解決 CPU 時間  . . . . . . . . . :              .403
BIND プログラム言語コンパイル CPU 時間 . . . . . . . . . . . :              .040
リスト作成 CPU 時間  . . . . . . . . . . . . . . . . . . . . :             1.622
プログラム／サービス・プログラム作成 CPU 時間  . . . . . . . . :              .178
合計 CPU 時間  . . . . . . . . . . . . . . . . . . . . . . . :             2.761
合計経過時間 . . . . . . . . . . . . . . . . . . . . . . . . :            11.522

                    * * * * *  バ  イ  ン  ド  統  計  の  終  わ  り  * * * * *

*CPC5D0B -  サービス・プログラム FINANCIAL がライブラリー MYLIB に作成された。

          * * * * *  サ  ー  ビ  ス  ・  プ  ロ  グ  ラ  ム  作  成  リ  ス  ト  の  終  わ  り  * * * * *
```

図 49. バインディング統計

拡張リスト

CRTPGM、CRTSRVPGM、UPDPGM、または UPDSRVPGM の各コマンドに DETAIL(*EXTENDED) を指定すると、リストには DETAIL(*BASIC) によって示されるすべての情報に加えて拡張要約表が含まれます。拡張要約表には、解決されたインポート (参照) の数、および処理されたエクスポート (定義) の数が示されます。 CRTSRVPGM または UPDSRVPGM コマンドの場合、このリストには、使用されたバインド・プログラム言語、生成されたシグニチャー、およびどのインポート (参照) がどのエクスポート (定義) に一致したかについても示されます。図 50、196 ページの図 51、および 197 ページの図 52 は追加のデータの例を示しています。

```
                                サービス・プログラムの作成                    ページ    2
5722SS1 V5R3M0  040525
                                                拡張要約表
   有効な定義 . . . . . . . . . . . . . . . :          418
     強 . . . . . . . . . . . . . . . . . . :          418
     弱 . . . . . . . . . . . . . . . . . . :            0
   解決済み参照 . . . . . . . . . . . . . . :           21
     終了強定義 . . . . . . . . . . . . . . :           21
     終了弱定義 . . . . . . . . . . . . . . :            0

     * * * * *  拡  張  要  約  表  の  終  わ  り  * * * * *
```

図 50. 拡張要約リスト

```
                                サービス・プログラムの作成                    ページ   4
5722SS1 V5R3M0  040525
                                BIND プログラムの情報リスト
モジュール . . . . . :   MONEY
  ライブラリー . . . . . :     MYLIB
  バインド . . . . . :     *YES
  変更日／時刻 . . . . . :     04/03/12  00:37:52
  テラスペース記憶域使用可能
    化 . . . . . :     *YES
  ストレージ・モデル . . . . :     *SNGLVL
      数値      記号        参照      識別コード                    タイプ    有効範囲    エクスポート    キー
     00000001   定義                  main                          PROC      MODULE      強
     00000002   定義                  Amount                        PROC      SRVPGM      強
     00000003   定義                  Payment                       PROC      SRVPGM      強
     00000004   参照        0000017F  Q LE AG_prod_rc Data
     00000005   参照        0000017E  Q LE AG_user_rc Data
     00000006   参照        000000AC  _C_main Proc
     00000007   参照        00000180  Q LE leDefaultEh Proc
     00000008   参照        00000181  Q LE mhConversionEh Proc
     00000009   参照        00000125  _C_exception_router Proc

モジュール . . . . . :   RATES
  ライブラリー . . . . . :     MYLIB
  バインド . . . . . :     *YES
      数値      記号        参照      識別コード                    タイプ    有効範囲    エクスポート    キー
     0000000A   定義                  Term                          PROC      SRVPGM      強
     0000000B   定義                  Rate                          PROC      SRVPGM      強
     0000000C   参照        0000017F  Q LE AG_prod_rc Data
     0000000D   参照        0000017E  Q LE AG_user_rc Data
     0000000E   参照        00000180  Q LE leDefaultEh Proc
     0000000F   参照        00000181  Q LE mhConversionEh Proc
     00000010   参照        00000125  _C_exception_router Proc

モジュール . . . . . :   CALCS
  ライブラリー . . . . . :     MYLIB
  バインド . . . . . :     *YES
      数値      記号        参照      識別コード                    タイプ    有効範囲    エクスポート    キー
     00000011   定義                  Calc1                         PROC      MODULE      強
     00000012   定義                  Calc2                         PROC      MODULE      強
     00000013   参照        0000017F  Q LE AG_prod_rc Data
     00000014   参照        0000017E  Q LE AG_user_rc Data
     00000015   参照        00000180  Q LE leDefaultEh Proc
     00000016   参照        00000181  Q LE mhConversionEh Proc
     00000017   参照        00000125  _C_exception_router Proc

モジュール . . . . . :   ACCTS
  ライブラリー . . . . . :     MYLIB
  バインド . . . . . :     *YES
      数値      記号        参照      識別コード                    タイプ    有効範囲    エクスポート    キー
     00000018   定義                  OpenAccount                   PROC      SRVPGM      強
     00000019   定義                  CloseAccount                  PROC      SRVPGM      強
     0000001A   参照        0000017F  Q LE AG_prod_rc Data
     0000001B   参照        0000017E  Q LE AG_user_rc Data
     0000001C   参照        00000180  Q LE leDefaultEh Proc
     0000001D   参照        00000181  Q LE mhConversionEh Proc
     0000001E   参照        00000125  _C_exception_router Proc
```

図 51. バインド・プログラム情報リスト (1/2)

```
サービス・プログラム . . . . :   QC2SYS
  ライブラリー . . . . . . . :     *LIBL
  バインド . . . . . . . . . :     *NO
      数値      記号        参照      識別コード                    タイプ    有効範囲    エクスポート    キー
     0000001F   定義                  system                        PROC                  強

サービス・プログラム . . . . :   QLEAWI
  ライブラリー . . . . . . . :     *LIBL
  バインド . . . . . . . . . :     *YES
      数値      記号        参照      識別コード                    タイプ    有効範囲    エクスポート    キー
     0000017E   定義                  Q LE AG_user_rc               データ                強
     0000017F   定義                  Q LE AG_prod_rc               データ                強
     00000180   定義                  Q LE leDefaultEh              PROC                  強
     00000181   定義                  Q LE mhConversionEh           PROC                  強
```

図 51. バインド・プログラム情報リスト (2/2)

```
                                サービス・プログラムの作成                          ページ  14

                                BIND プログラム言語のリスト
STRPGMEXP PGMLVL(*CURRENT)
  EXPORT SYMBOL('Term')
  EXPORT SYMBOL('Rate')
  EXPORT SYMBOL('Amount')
  EXPORT SYMBOL('Payment')
  EXPORT SYMBOL('OpenAccount')
  EXPORT SYMBOL('CloseAccount')
ENDPGMEXP
******** エクスポート・インターフェース識別値 : 00000000ADCEFEE088738A98DBA6E723
STRPGMEXP PGMLVL(*PRV)
  EXPORT SYMBOL('Term')
  EXPORT SYMBOL('Rate')
  EXPORT SYMBOL('Amount'
  EXPORT SYMBOL('Payment')
ENDPGMEXP
******** エクスポート・インターフェース識別値 : 0000000000000000000ADC89D09E0C6E7

    * * * * * B I N D プ ロ グ ラ ム 言 語 リ ス ト の 終 わ り * * * * *
```

図 52. バインド・プログラム言語リスト

フル・リスト

CRTPGM、CRTSRVPGM、UPDPGM、または UPDSRVPGM の各コマンドに DETAIL(*FULL) を指定すると、リストには DETAIL(*EXTENDED) によって示されるすべての詳細情報に加えて相互参照リストが含まれます。 198 ページの図 53 は、追加されるデータの例を示しています。

相互参照表

識別コード	定義	参照	参照	タイプ	ライブラリー	オブジェクト
.	.			.	.	.
.	.			.	.	.
.	.			.	.	.
xlatewt	000000DD			*SRVPGM	*LIBL	QC2UTIL1
yn	00000140			*SRVPGM	*LIBL	QC2UTIL2
y0	0000013E			*SRVPGM	*LIBL	QC2UTIL2
y1	0000013F			*SRVPGM	*LIBL	QC2UTIL2
Amount	00000002			*MODULE	MYLIB	MONEY
Calc1	00000011			*MODULE	MYLIB	CALCS
Calc2	00000012			*MODULE	MYLIB	CALCS
CloseAccount	00000019			*MODULE	MYLIB	ACCTS
CEECRHP	000001A0			*SRVPGM	*LIBL	QLEAWI
CEECZST	0000019F			*SRVPGM	*LIBL	QLEAWI
CEEDATE	000001A9			*SRVPGM	*LIBL	QLEAWI
CEEDATM	000001B1			*SRVPGM	*LIBL	QLEAWI
CEEDAYS	000001A8			*SRVPGM	*LIBL	QLEAWI
CEEDCOD	00000187			*SRVPGM	*LIBL	QLEAWI
CEEDSHP	000001A1			*SRVPGM	*LIBL	QLEAWI
CEEDYWK	000001B3			*SRVPGM	*LIBL	QLEAWI
CEEFMDA	000001AD			*SRVPGM	*LIBL	QLEAWI
CEEFMDT	000001AF			*SRVPGM	*LIBL	QLEAWI
CEEFMTM	000001AE			*SRVPGM	*LIBL	QLEAWI
CEEFRST	0000019E			*SRVPGM	*LIBL	QLEAWI
CEEGMT	000001B6			*SRVPGM	*LIBL	QLEAWI
CEEGPID	00000195			*SRVPGM	*LIBL	QLEAWI
CEEGTST	0000019D			*SRVPGM	*LIBL	QLEAWI
CEEISEC	000001B0			*SRVPGM	*LIBL	QLEAWI
CEELOCT	000001B4			*SRVPGM	*LIBL	QLEAWI
CEEMGET	00000183			*SRVPGM	*LIBL	QLEAWI
CEEMKHP	000001A2			*SRVPGM	*LIBL	QLEAWI
CEEMOUT	00000184			*SRVPGM	*LIBL	QLEAWI
CEEMRCR	00000182			*SRVPGM	*LIBL	QLEAWI
CEEMSG	00000185			*SRVPGM	*LIBL	QLEAWI
CEENCOD	00000186			*SRVPGM	*LIBL	QLEAWI
CEEQCEN	000001AC			*SRVPGM	*LIBL	QLEAWI
CEERLHP	000001A3			*SRVPGM	*LIBL	QLEAWI
CEESCEN	000001AB			*SRVPGM	*LIBL	QLEAWI
CEESECI	000001B2			*SRVPGM	*LIBL	QLEAWI
CEESECS	000001AA			*SRVPGM	*LIBL	QLEAWI
CEESGL	00000190			*SRVPGM	*LIBL	QLEAWI
CEETREC	00000191			*SRVPGM	*LIBL	QLEAWI
CEEUTC	000001B5			*SRVPGM	*LIBL	QLEAWI
CEEUTCO	000001B7			*SRVPGM	*LIBL	QLEAWI
CEE4ABN	00000192			*SRVPGM	*LIBL	QLEAWI
CEE4CpyDvfb	0000019A			*SRVPGM	*LIBL	QLEAWI
CEE4CpyIofb	00000199			*SRVPGM	*LIBL	QLEAWI
CEE4CpyOfb	00000198			*SRVPGM	*LIBL	QLEAWI
CEE4DAS	000001A4			*SRVPGM	*LIBL	QLEAWI
CEE4FCB	0000018A			*SRVPGM	*LIBL	QLEAWI
CEE4HC	00000197			*SRVPGM	*LIBL	QLEAWI
CEE4RAGE	0000018B			*SRVPGM	*LIBL	QLEAWI
CEE4RIN	00000196			*SRVPGM	*LIBL	QLEAWI
OpenAccount	00000018			*MODULE	MYLIB	ACCTS
Payment	00000003			*MODULE	MYLIB	MONEY
Q LE leBdyCh	00000188			*SRVPGM	*LIBL	QLEAWI
Q LE leBdyEpilog	00000189			*SRVPGM	*LIBL	QLEAWI
Q LE leDefaultEh	00000180	00000007	0000000E	*SRVPGM	*LIBL	QLEAWI
	00000015		0000001C			
Q LE mhConversionEh	00000181	00000008	0000000F	*SRVPGM	*LIBL	QLEAWI
	00000016		0000001D			
Q LE AG_prod_rc	0000017F	00000004	0000000C	*SRVPGM	*LIBL	QLEAWI
	00000013	0000001A				
Q LE AG_user_rc	0000017E	00000005	0000000D	*SRVPGM	*LIBL	QLEAWI
		00000014	0000001B			
Q LE HdlrRouterEh	0000018F			*SRVPGM	*LIBL	QLEAWI
Q LE RtxRouterCh	0000018E			*SRVPGM	*LIBL	QLEAWI
Rate	0000000B			*MODULE	MYLIB	RATES
Term	0000000A			*MODULE	MYLIB	RATES

図 53. 相互参照リスト

IPA リストの構成要素

以下のセクションで、リストの IPA 構成要素について説明します。

- オブジェクト・ファイル・マップ (Object File Map)
- コンパイラー・オプション・マップ (Compiler Options Map)
- インライン報告書 (Inline Report)
- グローバル記号マップ (Global Symbols Map)
- 区画マップ (Partition Map)
- ソース・ファイル・マップ (Source File Map)
- メッセージ (Messages)
- メッセージの要約 (Message Summary)

IPA(*YES) および DETAIL(*BASIC または *EXTENDED) を指定した場合、CRTPGM または CRTSRVPGM コマンドは、インライン報告書 (Inline Report) を除く上記のすべてのセクションを生成します。IPA(*YES) および DETAIL(*FULL) を指定した場合のみ、CRTPGM または CRTSRVPGM コマンドは、インライン報告書 (Inline Report) を生成します。

オブジェクト・ファイル・マップ (Object File Map)

オブジェクト・ファイル・マップ (Object File Map) は、IPA への入力として使用されたオブジェクト・ファイルの名前を表示します。ソース・ファイル・マップ (Source File Map) などの他のリスト・セクションでは、このリスト・セクションに表示される FILE ID 番号を使用します。

コンパイラー・オプション・マップ (Compiler Options Map)

コンパイラー・オプション・マップ (Compiler Options Map) リスト・セクションは、処理される各コンパイル単位について、IL データ内に指定されたコンパイラー・オプションを示します。それぞれのコンパイル単位ごとに、IPA 処理に関係のあるオプションを表示します。コンパイラー・オプション、#pragma ディレクティブを介して、またはデフォルト値として、これらのオプションを指定することができます。

インライン報告書 (Inline Report)

インライン報告書 (Inline Report) リスト・セクションは、IPA インライン化プログラムによって実行されるアクションについて記述します。この報告書で、「サブプログラム」という用語は C/C++ 関数または C++ メソッドと同じことです。この要約には、以下の情報が含まれます。

- 各定義済みサブプログラムの名前。 IPA は、サブプログラム名をアルファベット順にソートします。
- サブプログラムに対するアクションの理由。
  - サブプログラムに #pragma noinline が指定された。
  - サブプログラムに #pragma inline が指定された。
  - IPA がサブプログラムに自動インライン化を実行した。
  - サブプログラムにインライン化を実行する理由がなかった。
  - 区画の競合があった。

  - IL データが存在していなかったので、IPA がサブプログラムにインライン化を実行できなかった。
- サブプログラムに対するアクション。
  - IPA がサブプログラムにインライン化を少なくとも 1 回実行した。
  - 初期サイズ制約のために、IPA がサブプログラムにインライン化を実行しなかった。
  - サイズ制約を超える拡張のために、IPA がサブプログラムにインライン化を実行しなかった。
  - サブプログラムがインライン化の候補であったが、IPA はインライン化を実行しなかった。
  - サブプログラムがインライン化の候補であったが、参照されなかった。
  - サブプログラムが直接再帰プログラムであるか、またはパラメーターが一致していない呼び出しがあった。
- インライン化後の元のサブプログラムの状況。
  - サブプログラムはもう参照されないためまた、静的で内部であると定義されているので、IPA がサブプログラムを廃棄した。
  - IPA が以下のさまざまな理由でサブプログラムを廃棄しなかった。
    - サブプログラムが外部である。(サブプログラムがコンパイル単位の外側から呼び出される可能性がある。)
    - このサブプログラムに対するサブプログラム呼び出しが残っている。
    - サブプログラムのアドレスが使用されている。
- サブプログラムの初期相対サイズ (抽象コード単位で)。
- インライン化後のサブプログラムの最終相対サイズ (抽象コード単位で)。
- サブプログラム内の呼び出しの数、および IPA がサブプログラム内へインライン化したこれらの呼び出しの数。
- コンパイル単位内でサブプログラムが他のサブプログラムによって呼び出された回数、および IPA がサブプログラムをインライン化した回数。
- 選択されているモード、および指定されたしきい値と限界の値。アプリケーション全体の中では固有でない可能性がある静的関数の名前には、接頭部として `@nnn@` または `XXXX@nnn@` が付けられます。ここで、`XXXX` は区画名、また `nnn` はソース・ファイル番号です。

詳細呼び出し構造には、以下のような各サブプログラムに固有の情報が含まれます。

- 当該サブプログラムが呼び出すサブプログラム。
- 当該サブプログラムを呼び出すサブプログラム。
- 当該サブプログラムがインライン化される先のサブプログラム。

インライン化プログラムを、選択したモードで使用する必要がある場合には、これらの情報によって、プログラムをよりよく分析することができます。この報告書内のカウントには、IPA 以外のプログラムからの IPA プログラムの呼び出しは含まれません。

グローバル記号マップ (Global Symbols Map)

グローバル記号マップ (Global Symbols Map) リスト・セクションは、グローバル変数の合体最適化処理によって、グローバル・データ構造のメンバーにグローバル記号がマップされる方法を示します。このセクションには、記号情報とファイル名情報が含まれます (ファイル名情報は正確でない場合があります)。さらに、行番号情報も使用可能である場合があります。

区画マップ (Partition Map)

区画マップ (Partition Map) リスト・セクションは、IPA が作成する各オブジェクト・コード区画について説明します。このセクションは以下の情報を提供します。

- 各区画を生成するための理由。
- オブジェクト・コードを生成するために使用されたオプション。
- 区画に含まれている関数およびグローバル・データ。
- 区画を作成するために使用されたソース・ファイル。

ソース・ファイル・マップ (Source File Map)

ソース・ファイル・マップ (Source File Map) リスト・セクションは、オブジェクト・ファイルに組み込まれたソース・ファイルを示します。

メッセージ (Messages)

IPA は、エラーまたはエラーの可能性を検出すると、1 つ以上の診断メッセージを発行し、メッセージ (Messages) リスト・セクションを生成します。このリスト・セクションには、IPA 処理中に発行されたメッセージの要約が含まれます。メッセージは重大度別にソートされます。メッセージ (Messages) リスト・セクションは、各メッセージが最初に表示されたリスト・ページ番号を表示します。このセクションは、メッセージ・テキスト、およびオプショナルで、ファイル名、行 (既知の場合)、および桁 (既知の場合) に関する情報も表示します。

メッセージの要約 (Message Summary)

メッセージの要約 (Message Summary) リスト・セクションは、メッセージの合計数および重大度レベル別のメッセージ数を表示します。

サービス・プログラムの例のリスト

194 ページの図 49、 196 ページの図 51、および 198 ページの図 53 は 101 ページの図 36 のサービス・プログラム FINANCIAL を作成する際に DETAIL(*FULL) を指定した場合に生成されるリストのデータの一部を示しています。各図は、バインディング統計、バインド・プログラム情報リスト、および相互参照リストを示しています。

サービス・プログラムの例のバインド・プログラム情報リスト

バインド・プログラム情報リスト (196 ページの図 51) には、以下のデータおよび列見出しが示されます。

- 処理されたモジュールまたはサービス・プログラムの名前とライブラリー。

  *バインド*の欄の値が、モジュール・オブジェクトに関して *YES の場合、そのモジュールがコピーによってバインドされること示します。*バインド* の欄の値がサービス・プログラムに関して *YES の場合、そのサービス・プログラムが参照によってバインドされることを示します。*バインド* の欄の値がモジュール・オブジ

ェクトまたはサービス・プログラムに関して *NO の場合、そのオブジェクトがバインドに組み込まれないことを示します。その理由は、そのオブジェクトが未解決のインポートを満足するエクスポートを提供しなかったからです。

- 数値

  処理された各モジュールまたはサービス・プログラムごとに、各エクスポート (定義) またはインポート (参照) に関連付けられた固有の ID を示します。

- 記号

  この欄はエクスポート (定義) またはインポート (参照) としての記号名を示します。

- 参照

  この欄に示される番号は、インポート要求を満たすエクスポート (定義) の固有の ID です。 たとえば 196 ページの図 51 で、インポート 00000005 の固有の ID は、エクスポート 0000017E の固有の ID に一致します。

- ID

  これは、エクスポートまたはインポートされる記号の名前です。固有の ID 00000005 でインポートされる記号名は Q LE AG_user_rc です。 固有の ID 0000017E でエクスポートされる記号名も Q LE AG_user_rc です。

- タイプ

  記号名がプロシージャーの場合には、Proc として、また 記号名がデータ項目の場合には、Data として示されます。

- 有効範囲

  モジュールの場合、この欄は、エクスポートされた記号名がモジュール・レベルでアクセスされるか、またはサービス・プログラムへの共通インターフェースでアクセスされるかを示します。プログラムを作成している場合、エクスポートされた記号名は、モジュール・レベルでのみアクセス可能です。サービス・プログラムを作成している場合、エクスポートされた記号名は、モジュール・レベルでもサービス・プログラム (SrvPgm) レベルでもアクセス可能です。エクスポートされた記号が共通インターフェースの一部である場合、*有効範囲 (Scope)* 欄の値は SrvPgm でなければなりません。

- エクスポート

  この欄は、モジュールまたはサービス・プログラムからエクスポートされたデータ項目の強さを示します。

- キー

  この欄には、ウイーク・エクスポートに関する追加情報が入ります。通常、この欄はブランクです。

サービス・プログラムの例の相互参照リスト

198 ページの図 53 の相互参照リストは、バインド・プログラム情報に示されるデータに関する別の表示方法です。 相互参照リストには以下の列見出しがあります。

- ID

  記号解決時に処理されたエクスポートの名前です。

- 定義

  各エクスポートに関連する固有の ID です。

- 参照

この欄の番号は、該当のエクスポート (定義) に解決されたインポート (参照) の固有の ID を示します。

- タイプ

  そのエクスポートのエクスポート元が *MODULE オブジェクトであるか、または *SRVPGM オブジェクトであるかを示します。

- ライブラリー

  コマンドまたはバインディング・ディレクトリーに指定されたライブラリー名です。

- オブジェクト

  エクスポート (定義) を提供したオブジェクトの名前です。

サービス・プログラムの例のバインディング統計

194 ページの図 49 は、サービス・プログラム FINANCIAL の作成に関する一連の統計を示しています。統計は、バインド・プログラムが作成要求を処理した際の各処理の所要時間を示します。このセクションに示されるデータについては間接的な制御しか行うことができません。処理のオーバーヘッドには測定できない部分があります。したがって、*合計 CPU 時間* の欄にリストされる値は、それよりも前の各欄にリストされた時間の合計よりも大きくなります。

バインド・プログラム言語のエラー

システムがサービス・プログラムの作成の過程でバインド・プログラム言語を処理する際、エラーが発生することがあります。 サービス・プログラムの作成 (CRTSRVPGM) コマンドに DETAIL(*EXTENDED) または DETAIL(*FULL) を指定すると、スプール・ファイルにエラーが記録されます。

以下の通知メッセージが出されることがあります。

- インターフェース識別値が埋め込まれました
- インターフェース識別値が切り捨てられました

以下の警告エラーが出されることがあります。

- 現行エクスポート・ブロック限界インターフェース。
- 重複エクスポート・ブロック。
- 前のエクスポートでの重複記号。
- レベル検査は複数回停止できない。無視される。
- 複数の「現行」エクスポート・ブロックは使用できず、「前」と見なされる。

以下の重大エラーが発生することがあります。

- 現行エクスポート・ブロックが空である。
- エクスポート・ブロックが完了していない。ENDPGMEXP の前にファイルの終わりが見つかった。
- エクスポート・ブロックは開始していない。STRPGMEXP が必要である。
- エクスポート・ブロックはネストできない。ENDPGMEXP が抜けている。
- エクスポートはエクスポート・ブロックの中に存在しなければならない。

- 異なるエクスポート・ブロックのインターフェース識別値が同じです。エクスポートを変更する必要があります。
- ワイルドカード仕様と一致するものが複数ある。
- 「現行」エクスポート・ブロックがない。
- ワイルドカード仕様と一致するものがない。
- 前のエクスポート・ブロックが空である。
- インターフェース識別値がバリアント文字を含んでいる。
- LVLCHK(*NO) では、SIGNATURE(*GEN) が必要。
- インターフェース識別値構文が正しくない。
- 記号名が必要である。
- 記号がサービス・プログラム・エクスポートとして許可されない。
- 記号が定義されていない。
- 構文が正しくない。

インターフェース識別値が埋め込まれました

図 54 は、このメッセージを含むバインド・プログラム言語リストを示しています。

```
                                          BIND プログラム言語のリスト
            STRPGMEXP  PGMLVL(*CURRENT) SIGNATURE('SHORT SIGNATURE')
******** インターフェース識別値が埋め込まれた
            EXPORT SYMBOL('PROC_2')
            ENDPGMEXP
******** エクスポート・インターフェース識別値 : E2889699A340A289879581A3A4998540

    * * * * * B I N D プ ロ グ ラ ム 言 語 リ ス ト の 終 わ り * * * * *
```

図 54. 与えられたインターフェース識別値は 16 バイトより短いので埋め込まれました

これは通知メッセージです。

訂正処置

変更は必要ありません。

このメッセージを避けるためには、インターフェース識別値の長さを正確に 16 バイトにします。

インターフェース識別値が切り捨てられた

205 ページの図 55 は、このメッセージを含むバインド・プログラム言語リストを示しています。

```
                                                BIND プログラム言語のリスト
            STRPGMEXP  PGMLVL(*CURRENT) SIGNATURE('THIS SIGNATURE IS VERY LONG')
******** インターフェース識別値が切り捨てられた
            EXPORT SYMBOL('PROC_2')
            ENDPGMEXP
******** エクスポート・インターフェース識別値 : E38889A240A289879581A3A499854089

    * * * * * B I N D プ ロ グ ラ ム 言 語 リ ス ト の 終 わ り * * * * *
```

図 55. 与えられたデータの最初の 16 バイトだけがインターフェース識別値として使用されます

これは通知メッセージです。

訂正処置

変更は必要ありません。

このメッセージを避けるためには、インターフェース識別値の長さを正確に 16 バイトにします。

現行エクスポート・ブロック限界インターフェース

図 56 は、このエラーを含むバインド・プログラム言語リストを示しています。

```
                                                BIND プログラム言語のリスト
 STRPGMEXP  PGMLVL(*CURRENT)
   EXPORT     SYMBOL(A)
   EXPORT     SYMBOL(B)
 ENDPGMEXP
******** エクスポート・インターフェース識別値 : 00000000000000000000000000000CD2
 STRPGMEXP  PGMLVL(*PRV)
   EXPORT     SYMBOL(A)
   EXPORT     SYMBOL(B)
   EXPORT     SYMBOL(C)
 ENDPGMEXP
******** エクスポート・インターフェース識別値 : 00000000000000000000000000000CDE3
******** 現行エクスポート・ブロック限界インターフェース。

    * * * * * B I N D プ ロ グ ラ ム 言 語 リ ス ト の 終 わ り * * * * *
```

*図 56. PGMLVL(*PRV) は PGMLVL(*CURRENT) よりも多い記号をエクスポートしています*

これは警告エラーです。

PGMLVL(*PRV) エクスポート・ブロックが、PGMLVL(*CURRENT) エクスポート・ブロックよりも多い記号を指定しました。

他のエラーがなければ、サービス・プログラムは作成されます。

以下のいずれにも該当する場合、

- PGMLVL(*PRV) が C という名前のプロシージャーを以前にサポートしていた。
- 新しいサービス・プログラムのもとでは、プロシージャー C は、もはやサポートされていない。

このサービス・プログラムのプロシージャー C を呼び出す ILE プログラムまたはサービス・プログラムはいずれも、実行時にエラーを生じます。

訂正処置

1. PGMLVL(*CURRENT) エクスポート・ブロックに、PGMLVL(*PRV) エクスポート・ブロックより多くのエクスポートされる記号があることを確認してください。
2. CRTSRVPGM コマンドを再実行します。

この例では、EXPORT SYMBOL(C) が誤って STRPGMEXP PGMLVL(*CURRENT) ブロックではなく、PGMLVL(*PRV) ブロックに追加されています。

重複エクスポート・ブロック

図 57 は、このエラーを含むバインド・プログラム言語リストを示しています。

```
                                        BIND プログラム言語のリスト
 STRPGMEXP  PGMLVL(*CURRENT)
   EXPORT     SYMBOL(A)
   EXPORT     SYMBOL(B)
 ENDPGMEXP
******** エクスポート・インターフェース識別値 : 00000000000000000000000000000CD2
 STRPGMEXP  PGMLVL(*PRV)
   EXPORT     SYMBOL(A)
   EXPORT     SYMBOL(B)
 ENDPGMEXP
******** エクスポート・インターフェース識別値 : 00000000000000000000000000000CD2
******** 重複エクスポート・ブロック。

    * * * * * B I N D プ ロ グ ラ ム 言 語 リ ス ト の 終 わ り * * * * *
```

図 57. 重複する STRPGMEXP-ENDPGMEXP ブロック

これは警告エラーです。

複数の STRPGMEXP-ENDPGMEXP ブロックが同じ順序で同じ記号をエクスポートしました。

他のエラーがなければ、サービス・プログラムは作成されます。重複するインターフェース識別値は、作成されるサービス・プログラムに 1 回だけ組み込まれます。

訂正処置

1. 以下のいずれかの訂正処置を行います。
   - PGMLVL(*CURRENT) エクスポート・ブロックが正しいかどうかを確認します。必要ならば、更新します。
   - 重複するエクスポート・ブロックを除去します。
2. CRTSRVPGM コマンドを再実行します。

この例では、PGMLVL(*CURRENT) が指定されている STRPGMEXP コマンドの EXPORT SYMBOL(B) の後に以下のソース行を追加します。

```
EXPORT  SYMBOL(C)
```

前のエクスポートでの重複記号

図 58 は、重複記号エラーを含むバインド・プログラム言語リストを示しています。

```
                                   BIND プログラム言語のリスト
 STRPGMEXP  PGMLVL(*CURRENT)
   EXPORT     SYMBOL(A)
   EXPORT     SYMBOL(B)
   EXPORT     SYMBOL(A)
******** 前のエクスポートでの重複記号
 ENDPGMEXP
******** エクスポート・インターフェース識別値 : 00000000000000000000000000CDED3

    * * * * * B I N D プ ロ グ ラ ム 言 語 リ ス ト の 終 わ り * * * * *
```

図 58. 重複するエクスポート記号

これは警告エラーです。

サービス・プログラムからエクスポートされる記号が 1 つの STRPGMEXP-ENDPGMEXP ブロックに複数回指定されています。

他のエラーがなければ、サービス・プログラムは作成されます。最初の重複する記号だけが、サービス・プログラムからエクスポートされます。すべての重複する記号が、生成されるインターフェース識別値に影響します。

訂正処置

1. 重複するソース行の 1 つを、バインド・プログラム言語ソース・ファイルから除去します。
2. CRTSRVPGM コマンドを再実行します。

この例では、2 番目の EXPORT SYMBOL(A) を除去します。

レベル検査は複数回停止できない。無視される。

図 59 は、このエラーを含むバインド・プログラム言語リストを示しています。

```
                                   BIND プログラム言語のリスト
 STRPGMEXP  PGMLVL(*CURRENT)
   EXPORT     SYMBOL(A)
   EXPORT     SYMBOL(B)
 ENDPGMEXP
******** エクスポート・インターフェース識別値 : 00000000000000000000000000000000
 STRPGMEXP  PGMLVL(*PRV)
******** レベル検査は複数回停止できない－無視される。
   EXPORT     SYMBOL(A)
 ENDPGMEXP
******** エクスポート・インターフェース識別値 : 000000000000000000000000000000C1

    * * * * * B I N D プ ロ グ ラ ム 言 語 リ ス ト の 終 わ り * * * * *
```

*図 59. 複数の STRPGMEXP コマンドに LVLCHK(*NO) が指定されています*

これは警告エラーです。

複数の STRPGMEXP ブロックで LVLCHK(*NO) が指定されました。

他のエラーがなければ、サービス・プログラムは作成されます。2 番目以降の LVLCHK(*NO) は LVLCHK(*YES) であると想定されます。

訂正処置

1. 1 つの STRPGMEXP ブロックだけに LVLCHK(*NO) を指定したことを確認してください。
2. CRTSRVPGM コマンドを再実行します。

この例では、PGMLVL(*PRV) エクスポート・ブロックが、LVLCHK(*NO) を指定する唯一のエクスポート・ブロックであるべきです。PGMLVL(*CURRENT) エクスポート・ブロックから LVLCHK(*NO) の値を除去します。

複数の「現行」エクスポート・ブロックは使用できず、「前」と見なされる

図 60 は、このエラーを含むバインド・プログラム言語リストを示しています。

```
                                        BIND プログラム言語のリスト
 STRPGMEXP  PGMLVL(*CURRENT)
   EXPORT     SYMBOL(A)
   EXPORT     SYMBOL(B)
   EXPORT     SYMBOL(C)
 ENDPGMEXP
******** エクスポート・インターフェース識別値 : 00000000000000000000000000000CDE3
 STRPGMEXP
   EXPORT     SYMBOL(A)
******** 複数の「現行」エクスポート・ブロックは使用できず，「前」と見なされる
   EXPORT     SYMBOL(B)
 ENDPGMEXP
******** エクスポート・インターフェース識別値 : 00000000000000000000000000000CD2

    * * * * * B I N D プ ロ グ ラ ム 言 語 リ ス ト の 終 わ り * * * * *
```

*図 60. 複数の PGMLVL(*CURRENT) の値の指定*

これは警告エラーです。

複数の STRPGMEXP コマンドで、PGMLVL(*CURRENT) の値が指定されているか、またはデフォルトとして PGMLVL(*CURRENT) が指定されています。PGMLVL(*CURRENT) の値をもつ 2 番目以降のエクスポート・ブロックは PGMLVL(*PRV) であると想定されます。

他のエラーがなければ、サービス・プログラムは作成されます。

訂正処置

1. 該当するソース・テキストを STRPGMEXP PGMLVL(*PRV) に変更します。
2. CRTSRVPGM コマンドを再実行します。

この例では、2 番目の STRPGMEXP が変更すべきソース・テキストです。

現行エクスポート・ブロックが空である

図 61 は、このエラーを含むバインド・プログラム言語リストを示しています。

```
                                BIND プログラム言語のリスト
 STRPGMEXP  PGMLVL(*CURRENT)
 ENDPGMEXP
******** エクスポート・インターフェース識別値 : 00000000000000000000000000000000
***ERROR 現行エクスポート・ブロックが空である。

    * * * * * B I N D プ ロ グ ラ ム 言 語 リ ス ト の 終 わ り * * * * *
```

*図 61. STRPGMEXP PGMLVL(*CURRENT) ブロックからエクスポートされる記号がありません*

これは重大エラーです。

*CURRENT エクスポート・ブロックからエクスポートすべき記号がありません。

サービス・プログラムは作成されません。

訂正処置

1. 以下のいずれかの訂正処置を行います。
   - エクスポートすべき記号名を追加します。
   - 空の STRPGMEXP-ENDPGMEXP ブロックを除去し、PGMLVL(*CURRENT) として別の STRPGMEXP-ENDPGMEXP ブロックを作成します。
2. CRTSRVPGM コマンドを実行します。

この例では、バインド・プログラム言語ソース・ファイルの STRPGMEXP コマンドと ENDPGMEXP コマンドの間に以下のソース行を追加します。

```
EXPORT SYMBOL(A)
```

エクスポート・ブロックが完了していない。ENDPGMEXP の前にファイルの終わりが見つかった

図 62 は、このエラーを含むバインド・プログラム言語リストを示しています。

```
                                BIND プログラム言語のリスト
 STRPGMEXP  PGMLVL(*CURRENT)
***ERROR 構文が正しくない。
***ERROR エクスポート・ブロックが完了していない－ ENDPGMEXP の前にファイルの終りが見つかった。

    * * * * * B I N D プ ロ グ ラ ム 言 語 リ ス ト の 終 わ り * * * * *
```

図 62. ENDPGMEXP コマンドが見つかる前に、ソース・ファイルの終わりが検出されました

これは重大エラーです。

ファイルの終わりに到達する前に、ENDPGMEXP が検出されませんでした。

サービス・プログラムは作成されません。

訂正処置

1. 以下のいずれかの訂正処置を行います。
   - 適切な場所に ENDPGMEXP コマンドを追加します。
   - 対応する ENDPGMEXP コマンドをもたない STRPGMEXP コマンドを除去し、エクスポートを指定したすべて記号名を除去します。
2. CRTSRVPGM コマンドを実行します。

この例では、STRPGMEXP コマンドの後に以下の行を追加します。

```
  EXPORT  SYMBOL(A)
ENDPGMEXP
```

エクスポート・ブロックは開始していない。STRPGMEXP が必要である。

図 63 は、このエラーを含むバインド・プログラム言語リストを示しています。

```
                                        BIND プログラム言語のリスト
 ENDPGMEXP
***ERROR エクスポート・ブロックは開始していない－ STRPGMEXP が必要である。
***ERROR 「現行」エクスポート・ブロックがない。

    * * * * * B I N D プ ロ グ ラ ム 言 語 リ ス ト の 終 わ り * * * * *
```

図 63. STRPGMEXP コマンドが欠落しています

これは重大エラーです。

ENDPGMEXP コマンドの前に STRPGMEXP コマンドがありません。

サービス・プログラムは作成されません。

訂正処置

1. 以下のいずれかの訂正処置を行います。
   - STRPGMEXP コマンドを追加します。
   - エクスポートされるすべての記号と ENDPGMEXP コマンドを除去します。
2. CRTSRVPGM コマンドを実行します。

この例では、バインド・プログラム言語ソース・ファイルの ENDPGMEXP コマンドの前に、以下の 2 つのソース行を追加します。

```
STRPGMEXP
  EXPORT  SYMBOL(A)
```

エクスポート・ブロックはネストできない。ENDPGMEXP が抜けている

211 ページの図 64 は、このエラーを含むバインド・プログラム言語リストを示しています。

```
                                        BIND プログラム言語のリスト
  STRPGMEXP  PGMLVL(*CURRENT)
    EXPORT     SYMBOL(A)
    EXPORT     SYMBOL(B)
  STRPGMEXP  PGMLVL(*PRV)
 ***ERROR エクスポート・ブロックはネストできない－ ENDPGMEXP が抜けている。
    EXPORT     SYMBOL(A)
  ENDPGMEXP
 ******** エクスポート・インターフェース識別値 : 00000000000000000000000000000C1

    * * * * * B I N D プ ロ グ ラ ム 言 語 リ ス ト の 終 わ り * * * * *
```

図 64. ENDPGMEXP コマンドが欠落しています

これは重大エラーです。

2 番目の STRPGMEXP コマンドの前に ENDPGMEXP コマンドがありません。

サービス・プログラムは作成されません。

訂正処置

1. 以下のいずれかの訂正処置を行います。
   - 2 番目の STRPGMEXP コマンドの前に ENDPGMEXP コマンドを追加します。
   - STRPGMEXP コマンドとエクスポートする記号名を除去します。
2. CRTSRVPGM コマンドを実行します。

この例では、バインド・プログラム・ソース・ファイルの 2 番目の STRPGMEXP コマンドの前に ENDPGMEXP コマンドを追加します。

エクスポートはエクスポート・ブロックの中に存在しなければならない

図 65 は、このエラーを含むバインド・プログラム言語リストを示しています。

```
                                        BIND プログラム言語のリスト
  STRPGMEXP  PGMLVL(*CURRENT)
    EXPORT     SYMBOL(A)
    EXPORT     SYMBOL(B)
  ENDPGMEXP
 ******** エクスポート・インターフェース識別値 : 00000000000000000000000000000CD2
    EXPORT     SYMBOL(A)
 ***ERROR エクスポートはエクスポート・ブロックの中に存在しなければならない。

* * * * * B I N D プ ロ グ ラ ム 言 語 リ ス ト の 終 わ り * * * * *
```

図 65. エクスポートすべき記号名が STRPGMEXP-ENDPGMEXP ブロックの外側にあります

これは重大エラーです。

エクスポートすべき記号が、STRPGMEXP-ENDPGMEXP ブロック内に定義されていません。

サービス・プログラムは作成されません。

訂正処置

1. 以下のいずれかの訂正処置を行います。
   - エクスポートすべき記号を移動して、STRPGMEXP-ENDPGMEXP ブロック内に入れます。
   - 当該記号を除去します。
2. CRTSRVPGM コマンドを実行します。

この例では、エラーのソース行をバインド・プログラム言語ソース・ファイルから除去します。

異なるエクスポート・ブロックのインターフェース識別値が同じです。エクスポートを変更する必要があります

これは重大エラーです。

異なる記号をエクスポートする複数の STRPGMEXP-ENDPGMEXP ブロックから同じインターフェース識別値が生成されました。このエラー条件はほとんど発生しません。 エクスポートされるかなりの数の記号のセットに対して、このエラーは 3.4E28 回の試行ごとに 1 回しか発生しません。

サービス・プログラムは作成されません。

訂正処置

1. 以下のいずれかの訂正処置を行います。
   - PGMLVL(*CURRENT) ブロックからエクスポートする他の記号を追加します。

     推奨する方法は、すでにエクスポート済みの記号を指定することです。これによって、重複記号に関する警告エラーが出されますが、インターフェース識別値は確実に固有のものになります。 別の方法は、エクスポート済みでない別のエクスポート記号を追加することです。
   - モジュールからエクスポートすべき記号の名前を変更し、対応する変更をバインド・プログラム言語ソース・ファイルに行います。
   - プログラム・エクスポート・リストの開始 (STRPGMEXP) コマンドに SIGNATURE パラメーターを使用してインターフェース識別値を指定します。
2. CRTSRVPGM コマンドを実行します。

ワイルドカード仕様と一致するものが複数ある

213 ページの図 66 は、このエラーを含むバインド・プログラム言語リストを示しています。

```
                                        BIND プログラム言語のリスト
 STRPGMEXP  PGMLVL(*CURRENT)
   EXPORT  ("A"<<<)
*** エラー－ワイルドカード仕様と一致するものが複数ある
   EXPORT  ("B"<<<)
 ENDPGMEXP
******** エクスポート・インターフェース識別値 : 00000000000000000000000000000FFC2

    * * * * * B I N D プ ロ グ ラ ム 言 語 リ ス ト の 終 わ り * * * * *
```

図 66. ワイルドカードの指定と複数のエクスポート記号が一致します

これは重大エラーです。

エクスポートに対し指定されたワイルドカードが、エクスポート可能な複数の記号と一致します。

サービス・プログラムは作成されません。

訂正処置

1. 一致するエクスポートが 1 つだけになるように、ワイルドカードの指定をより特定化します。
2. CRTSRVPGM コマンドを実行します。

「現行」エクスポート・ブロックがない

図 67 は、このエラーを含むバインド・プログラム言語リストを示しています。

```
                                        BIND プログラム言語のリスト
 STRPGMEXP  PGMLVL(*PRV)
   EXPORT  SYMBOL(A)
 ENDPGMEXP
******** エクスポート・インターフェース識別値 : 00000000000000000000000000000C1
***ERROR 「現行」エクスポート・ブロックがない。

    * * * * * B I N D プ ロ グ ラ ム 言 語 リ ス ト の 終 わ り * * * * *
```

*図 67. PGMLVL(*CURRENT) エクスポート・ブロックがありません*

これは重大エラーです。

バインド・プログラム言語ソース・ファイルに STRPGMEXP PGMLVL(*CURRENT) がありません。

サービス・プログラムは作成されません。

訂正処置

1. 以下のいずれかの訂正処置を行います。
   - PGMLVL(*PRV) を PGMLVL(*CURRENT) に変更します。
   - 正しい *CURRENT エクスポート・ブロックである STRPGMEXP-ENDPGMEXP ブロックを追加します。
2. CRTSRVPGM コマンドを実行します。

この例では、PGMLVL(*PRV) を PGMLVL(*CURRENT) に変更します。

ワイルドカード仕様と一致するものがない

図 68 は、このエラーを含むバインド・プログラム言語リストを示しています。

```
                                        BIND プログラム言語のリスト
 STRPGMEXP  PGMLVL(*CURRENT)
   EXPORT  ("Z"<<<)
*** エラー－ワイルドカード仕様と一致するものがない
   EXPORT  ("B"<<<)
 ENDPGMEXP
******** エクスポート・インターフェース識別値 : 00000000000000000000000000000FFC2

    * * * * * B I N D プ ロ グ ラ ム 言 語 リ ス ト の 終 わ り * * * * *
```

図 68. ワイルドカードの指定と一致するエクスポート記号がありません

これは重大エラーです。

エクスポートに対し指定されたワイルドカードと一致するエクスポート可能な記号はありません。

サービス・プログラムは作成されません。

訂正処置

1. エクスポートしたい記号と一致するワイルドカードを指定します。
2. CRTSRVPGM コマンドを実行します。

前のエクスポート・ブロックが空である

図 69 は、このエラーを含むバインド・プログラム言語リストを示しています。

```
                                        BIND プログラム言語のリスト
 STRPGMEXP  PGMLVL(*CURRENT)
   EXPORT     SYMBOL(A)
   EXPORT     SYMBOL(B)
 ENDPGMEXP
******** エクスポート・インターフェース識別値 : 00000000000000000000000000000CD2
 STRPGMEXP  PGMLVL(*PRV)
 ENDPGMEXP
******** エクスポート・インターフェース識別値 : 00000000000000000000000000000C1
***ERROR 前のエクスポート・ブロックが空である。

    * * * * * B I N D プ ロ グ ラ ム 言 語 リ ス ト の 終 わ り * * * * *
```

*図 69. PGMLVL(*CURRENT) エクスポート・ブロックがありません*

これは重大エラーです。

STRPGMEXP PGMLVL(*PRV) がありますが、記号が指定されていません。

サービス・プログラムは作成されません。

訂正処置

1. 以下のいずれかの訂正処置を行います。
   - 空の STRPGMEXP-ENDPGMEXP ブロックに記号を追加します。

- 空の STRPGMEXP-ENDPGMEXP ブロックを除去します。

2. CRTSRVPGM コマンドを実行します。

この例では、バインド・プログラム言語ソース・ファイルから空の STRPGMEXP-ENDPGMEXP ブロックを除去します。

インターフェース識別値に可変文字が入ってる

図 70 は、このエラーを含むバインド・プログラム言語リストを示しています。

```
                                        BIND プログラム言語のリスト
            STRPGMEXP  SIGNATURE('\!CDEFGHIJKLMOP')
***ERROR インターフェース識別値に可変文字が入っている
            EXPORT  SYMBOL('PROC_2')
            ENDPGMEXP
******** エクスポート・インターフェース識別値 : E05A838485868788899191929394959697

    * * * * * B I N D プ ロ グ ラ ム 言 語 リ ス ト の 終 わ り * * * * *
```

図 70. インターフェース識別値に可変文字が入っています

これは重大エラーです。

インターフェース識別値はすべてのコード化文字セット ID (CCSID) にない文字を含んでいます。

サービス・プログラムは作成されません。

訂正処置

1. バリアント文字を除去します。
2. CRTSRVPGM コマンドを実行します。

この場合には、¥! を除去する必要があります。

LVLCHK(*NO) では SIGNATURE(*GEN) が必要

図 71 は、このエラーを含むバインド・プログラム言語リストを示しています。

```
                                        BIND プログラム言語のリスト
            STRPGMEXP  SIGNATURE('ABCDEFGHIJKLMOP')  LVLCHK(*NO)
            EXPORT  SYMBOL('PROC_2')
***ERROR LVLCHK(*NO) では SIGNATURE(*GEN) が必要
            ENDPGMEXP
******** エクスポート・インターフェース識別値 : E05A838485868788899191929394959697

    * * * * * B I N D プ ロ グ ラ ム 言 語 リ ス ト の 終 わ り * * * * *
```

*図 71. LVLCHK(*NO) が指定されると、明示的インターフェース識別値は無効です*

これは重大エラーです。

LVLCHK(*NO) が指定されると、SIGNATURE(*GEN) が必要となります。

サービス・プログラムは作成されません。

訂正処置

1. 以下のいずれかの訂正処置を行います。
   - SIGNATURE(*GEN) を指定します。
   - LVLCHK(*YES) を指定します。
2. CRTSRVPGM コマンドを実行します。

インターフェース識別値構文が正しくない

図 72 は、このエラーを含むバインド・プログラム言語リストを示しています。

```
                                                BIND プログラム言語のリスト
            STRPGMEXP  SIGNATURE('"ABCDEFGHIJKL "')
***ERROR インターフェース識別値構文が正しくない
***ERROR インターフェース識別値構文が正しくない
***ERROR 構文が正しくない。
***ERROR 構文が正しくない。
            EXPORT  SYMBOL('PROC_2')
            ENDPGMEXP

    * * * * * B I N D プ ロ グ ラ ム 言 語 リ ス ト の 終 わ り * * * * *
```

図 72. インターフェース識別値として指定された値が無効です

これは重大エラーです。

インターフェース識別値が無効な文字を含んでいます。

サービス・プログラムは作成されません。

訂正処置

1. インターフェース識別値から無効な文字を除去します。
2. CRTSRVPGM コマンドを実行します。

この場合、シグニチャー・フィールドから文字 ″ を除去します。

記号名が必要である

図 73 は、このエラーを含むバインド・プログラム言語リストを示しています。

```
                                                BIND プログラム言語のリスト
 STRPGMEXP  PGMLVL(*CURRENT)
   EXPORT     SYMBOL(A)
   EXPORT     SYMBOL(')
***ERROR 記号名が必要である。
 ENDPGMEXP
******** エクスポート・インターフェース識別値 : 0000000000000000000000000000000C1

    * * * * * B I N D プ ロ グ ラ ム 言 語 リ ス ト の 終 わ り * * * * *
```

図 73. エクスポートすべき記号がありません

これは重大エラーです。

サービス・プログラムからエクスポートすべき記号名が見つかりません。

サービス・プログラムは作成されません。

訂正処置

1. 以下のいずれかの訂正処置を行います。
   - エラーがある行をバインド・プログラム言語ソース・ファイルから除去します。
   - サービス・プログラムからエクスポートすべき記号名を追加します。
2. CRTSRVPGM コマンドを実行します。

この例では、バインド・プログラム言語ソース・ファイルからソース行 EXPORT SYMBOL("") を除去します。

記号がサービス・プログラム・エクスポートとして許可されない

図 74 は、このエラーを含むバインド・プログラム言語リストを示しています。

```
                                        BIND プログラム言語のリスト
 STRPGMEXP  PGMLVL(*CURRENT)
   EXPORT     SYMBOL(A)
***ERROR 記号がサービス・プログラム・エクスポートとして許可されない。
   EXPORT     SYMBOL(D)
 ENDPGMEXP
******** エクスポート・インターフェース識別値 : 00000000000000000000000000000CD4

    * * * * * B I N D プ ロ グ ラ ム 言 語 リ ス ト の 終 わ り * * * * *
```

図 74. サービス・プログラムからのエクスポートとして適切な記号名ではありません

これは重大エラーです。

サービス・プログラムからエクスポートすべき記号が、コピーによってバインドされるモジュールのいずれからもエクスポートされません。通常、サービス・プログラムからのエクスポートを指定された記号は、実際には、サービス・プログラムによるインポートが必要な記号です。

サービス・プログラムは作成されません。

訂正処置

1. 以下のいずれかの訂正処置を行います。
   - エラーである記号を、バインド・プログラム言語ソース・ファイルから除去します。
   - CRTSRVPGM コマンドの MODULE パラメーターに、エクスポートすべき記号をもっているモジュールを指定します。
   - コピーによってバインドされるモジュールのいずれかにこの記号を追加し、モジュール・オブジェクトを再作成します。
2. CRTSRVPGM コマンドを実行します。

この例では、バインド・プログラム言語ソース・ファイルからソース行 EXPORT SYMBOL(A) を除去します。

記号が定義されていない

図 75 は、このエラーを含むバインド・プログラム言語リストを示しています。

```
                                        BIND プログラム言語のリスト
 STRPGMEXP  PGMLVL(*CURRENT)
   EXPORT     SYMBOL(A)
   EXPORT     SYMBOL(Q)
***ERROR  記号が定義されていない。
 ENDPGMEXP
******** エクスポート・インターフェース識別値 : 00000000000000000000000000000CE8

    * * * * * B I N D プ ロ グ ラ ム 言 語 リ ス ト の 終 わ り * * * * *
```

図 75. コピーによってバインドされるモジュールに記号が見つかりません

これは重大エラーです。

サービス・プログラムからエクスポートすべき記号が、コピーによってバインドされるモジュールにありません。

サービス・プログラムは作成されません。

訂正処置

1. 以下のいずれかの訂正処置を行います。
   - 定義されていない記号をバインド・プログラム言語ソース・ファイルから除去します。
   - CRTSRVPGM コマンドの MODULE パラメーターに、エクスポートすべき記号をもっているモジュールを指定します。
   - コピーによってバインドされるモジュールのいずれかにこの記号を追加し、モジュール・オブジェクトを再作成します。
2. CRTSRVPGM コマンドを実行します。

この例では、バインド・プログラム言語ソース・ファイルからソース行 EXPORT SYMBOL(Q) を除去します。

構文が正しくない

これは重大エラーです。

ソース・メンバーのステートメントが無効なバインド・プログラム言語ステートメントです。

サービス・プログラムは作成されません。

訂正処置

1. 有効なバインド・プログラム言語ステートメントを含むソース・メンバーに訂正してください。

2. CRTSRVPGM コマンドを実行します。

付録 B. 最適化プログラムにおける例外

場合によっては、最適化レベル 30 (*FULL) または 40 を指定してコンパイルしたプログラムで、MCH3601 例外メッセージが出ることがまれにあります。この付録では、このメッセージが出される 1 つの例について説明します。同じプログラムを最適化レベル 10 (*NONE) または 20 (*BASIC) を指定してコンパイルした場合には、MCH3601 例外メッセージは出されません。この例でメッセージが出されるか否かは、ILE HLL コンパイラーが配列にストレージを割り振る方法によって決まります。この例は、言語によっては発生しないことがあります。

最適化レベル 30 (*FULL) または 40 を要求すると、ILE はループの外側で配列の指標参照を計算することによってパフォーマンスの改善を試みます。ループで配列を参照する場合、配列のすべての要素を順番にアクセスすることは珍しいことではありません。前のループの反復からの最後の配列要素アドレスを保管することによって、パフォーマンスを改善することができます。このパフォーマンスの改善を行うために、ILE は最初の配列要素アドレスをループ外で計算し、その値を保管してループ内で使用します。

以下に例を示します。

```
DCL ARR[1000] INTEGER;
DCL I INTEGER;

I = init_expression;   /* init_expression が -1 と評価され、
                          それが I に割り当てられると想定する */

/* この間にいくつかのステートメントがある */

WHILE ( I < limit_expression )

  I = I + 1;

  /* WHILE ループの中の一部のステートメント */

  ARR[I] = some_expression;

  /* WHILE ループの中のその他のステートメント */

END;
```

ARR[init_expression] への参照が正しくない配列の指標を作ると、この例では、MCH3601 例外が起きる可能性があります。これは、ILE が、WHILE ループに入る前に最初の配列要素アドレスの計算を試みたことが原因です。

最適化レベル 30 (*FULL) または 40 で MCH3601 例外が出された場合には、以下の状態になっていないか調べてください。

1. 配列要素の指標としての変数を使用する前に、その変数を増分するループがある。
2. ループに入った時点で、指標変数の初期値が負である。
3. 変数の初期値を使用した配列の参照が無効である。

上記の条件に該当する場合には以下を行うことにより、最適化レベル 30 (*FULL) または 40 が使用できるようになることがあります。

1. 該当の変数を増分するプログラムの部分を、ループの最下部に移す。
2. 必要に応じてその変数に対する参照を変更する。

前の例は次のように変更します。

```
I = init_expression + 1;

WHILE ( I < limit_expression + 1 )

  ARR[I] = some_expression;

  I = I + 1;

END;
```

このような変更が不能な場合には、最適化レベルを 30 (*FULL) または 40 から 20 (*BASIC) または 10 (*NONE) に変更します。

付録 C. ILE オブジェクトに使用される CL コマンド

以下のテーブルは、ILE の各オブジェクトに使用される CL コマンドを示しています。

モジュールに使用される CL コマンド

表 13. モジュールに使用される CL コマンド

コマンド	記述名
CHGMOD	モジュールの変更
CRTCBLMOD	COBOL モジュール作成
CRTCLMOD	CL モジュールの作成
CRTCMOD	C モジュールの作成
CRTCPPMOD	C++ モジュールの作成
CRTRPGMOD	RPG モジュールの作成
DLTMOD	モジュールの削除
DSPMOD	モジュールの表示
RTVBNDSRC	バインダー・ソース検索
WRKMOD	モジュールの処理

プログラム・オブジェクトに使用される CL コマンド

表 14. プログラム・オブジェクトに使用される CL コマンド

コマンド	記述名
CHGPGM	プログラム変更
CRTBNDC	バインド C プログラム作成
CRTBNDCBL	バインド COBOL プログラムの作成
CRTBNDCL	バインド CL プログラムの作成
CRTBNDCPP	バインド C++ プログラムの作成
CRTBNDRPG	バインド RPG プログラムの作成
CRTPGM	プログラムの作成
DLTPGM	プログラム削除
DSPPGM	プログラム表示
DSPPGMREF	プログラム参照表示
UPDPGM	プログラムの更新
WRKPGM	プログラムの処理

サービス・プログラムに使用される CL コマンド

表 15. サービス・プログラムに使用される CL コマンド

コマンド	記述名
CHGSRVPGM	サービス・プログラムの変更
CRTSRVPGM	サービス・プログラムの作成
DLTSRVPGM	サービス・プログラムの削除
DSPSRVPGM	サービス・プログラムの表示
RTVBNDSRC	バインダー・ソース検索
UPDSRVPGM	サービス・プログラムの更新
WRKSRVPGM	サービス・プログラムの処理

バインディング・ディレクトリーに使用される CL コマンド

表 16. バインディング・ディレクトリーに使用される CL コマンド

コマンド	記述名
ADDBNDDIRE	バインディング・ディレクトリー項目の追加
CRTBNDDIR	バインディング・ディレクトリーの作成
DLTBNDDIR	バインディング・ディレクトリーの削除
DSPBNDDIR	バインディング・ディレクトリーの表示
RMVBNDDIRE	バインディング・ディレクトリー項目の除去
WRKBNDDIR	バインディング・ディレクトリーの処理
WRKBNDDIRE	バインディング・ディレクトリー項目の処理

構造化照会言語に使用される CL コマンド

表 17. 構造化照会言語に使用される CL コマンド

コマンド	記述名
CRTSQLCI	SQL ILE C オブジェクトの作成
CRTSQLCBLI	SQL ILE COBOL オブジェクトの作成
CRTSQLRPGI	SQL ILE RPG オブジェクトの作成

CICS®に使用される CL コマンド

表 18. CICSに使用される CL コマンド

コマンド	記述名
CRTCICSC	CICS ILE C オブジェクトの作成
CRTCICSCBL	CICS COBOL プログラムの作成

ソース・デバッガーに使用される CL コマンド

表 19. ソース・デバッガーに使用される CL コマンド

コマンド	記述名
DSPMODSRC	モジュール・ソースの表示
ENDDBG	デバッグ・モード終了
STRDBG	デバッグ開始

バインド・プログラム言語ソース・ファイルの編集に使用される CL コマンド

表 20. バインド・プログラム言語ソースの編集に使用される CL コマンド

コマンド	記述名
EDTF	ファイルの編集
STRPDM	PDM 開始
STRSEU	SEU 開始
注: 以下の実行不能コマンドは、バインド・プログラム言語ソース・ファイルに入力することができます。	
ENDPGMEXP	プログラム・エクスポート・リストの終了
EXPORT	エクスポート

付録 D. 特記事項

本書は米国 IBM が提供する製品およびサービスについて作成したものであり、本書に記載の製品、サービス、または機能が日本においては提供されていない場合があります。

本書に記載の製品、サービス、または機能が日本においては提供されていない場合があります。日本で利用可能な製品、サービス、および機能については、日本 IBM の営業担当員にお尋ねください。本書で IBM 製品、プログラム、またはサービスに言及していても、その IBM 製品、プログラム、またはサービスのみが使用可能であることを意味するものではありません。これらに代えて、IBM の知的所有権を侵害することのない、機能的に同等の製品、プログラム、またはサービスを使用することができます。ただし、IBM 以外の製品とプログラムの操作またはサービスの評価および検証は、お客様の責任で行っていただきます。

IBM は、本書に記載されている内容に関して特許権 (特許出願中のものを含む) を保有している場合があります。本書の提供は、お客様にこれらの特許権について実施権を許諾することを意味するものではありません。実施権については、書面にて下記宛先にお送りください。

〒106-0032
東京都港区六本木 3-2-31
IBM World Trade Asia Corporation
Licensing

以下の保証は、国または地域の法律に沿わない場合は、適用されません。IBM およびその直接または間接の子会社は、本書を特定物として現存するままの状態で提供し、商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示もしくは黙示の保証責任を負わないものとします。国または地域によっては、法律の強行規定により、保証責任の制限が禁じられる場合、強行規定の制限を受けるものとします。

この情報には、技術的に不適切な記述や誤植を含む場合があります。本書は定期的に見直され、必要な変更は本書の次版に組み込まれます。 IBM は予告なしに、随時、この文書に記載されている製品またはプログラムに対して、改良または変更を行うことがあります。

本書において IBM 以外の Web サイトに言及している場合がありますが、便宜のため記載しただけであり、決してそれらの Web サイトを推奨するものではありません。それらの Web サイトにある資料は、この IBM 製品の資料の一部ではありません。それらの Web サイトは、お客様の責任でご使用ください。

IBM は、お客様が提供するいかなる情報も、お客様に対してなんら義務も負うことのない、自ら適切と信ずる方法で、使用もしくは配布することができるものとします。

本プログラムのライセンス保持者で、(i) 独自に作成したプログラムとその他のプログラム（本プログラムを含む）との間での情報交換、および (ii) 交換された情報の相互利用を可能にすることを目的として、本プログラムに関する情報を必要とする方は、下記に連絡してください。

IBM Corporation
Software Interoperability Coordinator, Department 49XA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

本プログラムに関する上記の情報は、適切な使用条件の下で使用することができますが、有償の場合もあります。

本書で説明されているライセンス・プログラムまたはその他のライセンス資料は、IBM 所定のプログラム契約の契約条項、IBM プログラムのご使用条件、またはそれと同等の条項に基づいて、IBM より提供されます。

本書には、日常の業務処理で用いられるデータや報告書の例が含まれています。より具体性を与えるために、それらの例には、個人、企業、ブランド、あるいは製品などの名前が含まれている場合があります。これらの名称はすべて架空のものであり、名称や住所が類似する企業が実在しているとしても、それは偶然にすぎません。

著作権使用許諾:

本書には、様々なオペレーティング・プラットフォームでのプログラミング手法を例示するサンプル・アプリケーション・プログラムがソース言語で掲載されています。お客様は、サンプル・プログラムが書かれているオペレーティング・プラットフォームのアプリケーション・プログラミング・インターフェースに準拠したアプリケーション・プログラムの開発、使用、販売、配布を目的として、いかなる形式においても、IBM に対価を支払うことなくこれを複製し、改変し、配布することができます。このサンプル・プログラムは、あらゆる条件下における完全なテストを経ていません。従って IBM は、これらのサンプル・プログラムについて信頼性、利便性もしくは機能性があることをほのめかしたり、保証することはできません。お客様は、IBM のアプリケーション・プログラミング・インターフェースに準拠したアプリケーション・プログラムの開発、使用、販売、配布を目的として、いかなる形式においても、IBM に対価を支払うことなくこれを複製し、改変し、配布することができます。

それぞれの複製物、サンプル・プログラムのいかなる部分、またはすべての派生的創作物にも、次のように、著作権表示を入れていただく必要があります。

© (お客様の会社名) (西暦年). このコードの一部は、IBM Corp. のサンプル・プログラムから取られています。© Copyright IBM Corp. _年を入れる_. All rights reserved.

プログラミング・インターフェース情報

本書の目的は、統合化言語環境プログラムをお客様が使用する手助けをすることです。本書は、OS/400 が提供している汎用プログラミング・インターフェースとそれに関連する情報を記述しています。

汎用プログラミング・インターフェースにより、お客さまが OS/400 の機能を使用するプログラムを書くことができます。

商標

本書で使用する以下の用語は、IBM Corporation の米国およびその他の国における商標です。

400
AS/400
CICS
IBM
iSeries
OS/2
OS/400
POWER4
WebSphere

Windows および Windows ロゴは、Microsoft® Corporation の米国およびその他の国における商標です。

Java およびすべての Java 関連の商標およびロゴは、Sun Microsystems, Inc. の米国およびその他の国における商標または登録商標です。

UNIX は、The Open Group がライセンスしている米国およびその他の国における登録商標です。

他の会社名、製品名およびサービス名などはそれぞれ各社の商標または登録商標です。

参考文献

iSeries サーバーの ILE 環境のトピックについての関連情報は、以下の資料を参照してください。

- Information Center のトピック『バックアップおよび回復』では、バックアップと回復のストラテジーの立案に関する情報、およびシステム・データの保管と復元に使用できるさまざまなタイプのメディアに関する情報を提供し、ジャーナルを使用してデータベース・ファイルに対して行われた変更を記録する方法、およびシステム回復のためにその情報を使用する方法について説明しています。この資料は、ユーザーの補助ストレージ・プール (ASP)、ミラー保護、およびチェックサムを計画し設定する方法とともに、他の使用可能な回復に関するトピックについて説明しています。また、バックアップをもとにシステムを再びインストールする方法についても説明しています。

- 「CL プログラミング」  は、オブジェクトとライブラリー、CL プログラミング、プログラム間の流れとやりとりの制御、CL プログラムのオブジェクトの処理、および CL プログラムの作成に関する一般的な説明を含む、プログラミング・トピックに関する広範な説明を提供します。他のトピックには、事前定義メッセージと即時メッセージおよびメッセージ処理、ユーザー定義コマンドとメニューの定義と作成、アプリケーションのテスト (デバッグ・モード、停止点、トレース機能、および表示機能を含む) があります。

- 「Communications Management」  は、通信環境における実行管理、通信状況、通信上の問題のトレースと診断、エラー処理と回復、パフォーマンス、および特定の回線速度とサブシステム・ストレージに関する情報を提供します。

- 「ICF Programming」  は、通信機能および OS/400 システム間通信機能 (OS/400-ICF) を使用するアプリケーション・プログラムを作成するのに必要な情報について説明しています。この資料には、データ記述仕様 (DDS) キーワード、システム提供形式、戻りコード、ファイル転送サポートおよびプログラム例に関する情報も含まれています。

- 「WebSphere Development Studio ILE C/C++ Programmer's Guide」  では、ILE C および ILE C++ プログラムを作成、コンパイル、デバッグ、および実行するための方法を説明しています。このガイドには、ILE および OS/400 のプログラミング機能、iSeries のファイル・システムと装置と機能、入出力操作、ローカライズ、プログラムのパフォーマンス、および C++ のランタイム・タイプ情報 (RTTI) に関する情報が記載されています。

- 「WebSphere Development Studio C/C++ Language Reference」  では、「*プログラム言語 - C*」(ISO/IEC 9899:1990) 標準および「*プログラム言語 - C++*」(ISO/IEC 14882:1998) 標準に準拠した言語が説明されています。

- 「WebSphere Development Studio ILE C/C++ Compiler Reference」  には、iSeries および QShell 環境で使用されるプリプロセッサー・ステートメント、マクロ、pragma、およびコマンド行、そして入出力の考慮事項など、ILE C/C++ コンパイラーに関する参照情報が記載されています。

- 「ILE C/C++ Run-time Library Functions」  には、ILE C/C++ ランタイム・ライブラリー関数、インクルード・ファイル、および実行時の考慮事項に関する参照情報が記載されています。

- 「WebSphere Development Studio: ILE COBOL プログラマーの手引き」  は、AS/400 システムにおける ILE COBOL プログラムの作成、コンパイル、バインド、実行、デバッグ、および保守の方法について説明しています。この資料は、他の ILE COBOL プログラムや ILE COBOL 以外のプログラムを呼び出す方法、他のプログラムとデータを共用する方法、ポインターを使用する方法、および例外を処理する方

法に関するプログラミング情報を提供します。この資料は、また、外部接続装置、データベース・ファイル、表示装置ファイル、および ICF ファイルで入出力操作を行う方法を説明しています。

- 「WebSphere Development Studio: ILE COBOL 解説書」  は、ILE COBOL プログラム言語について説明しています。この資料は、ILE COBOL プログラム言語の構造および ILE COBOL ソース・プログラムの構造に関する情報を提供します。 この資料は、また、すべての Identification Division 段落、Environment Division 文節、Data Division 段落、Procedure Division ステートメント、およびコンパイラー指示ステートメントについて説明しています。
- 「WebSphere Development Studio: ILE RPG プログラマーの手引き」  は、iSeries サーバーで統合化言語環境 (ILE) に ILE RPG をインプリメントしたものである、RPG IV プログラム言語を使用するための手引書です。 この資料は、プロシージャー呼び出しとプログラム間プログラミングを考慮したプログラムの作成と実行に関する情報を含みます。また、デバッグと例外処理および、RPG プログラムでの AS/400 のファイルと装置の使用方法を説明しています。付録には、RPG IV への移行に関する情報およびコンパイラー・リスト例が含まれています。この資料は、読者が、データ処理概念と RPG 言語についての基礎的理解を持っていることを前提にしています。
- 「WebSphere Development Studio: ILE RPG 言語解説書」  は、RPG IV プログラム言語を使用して、iSeries サーバーのプログラムを作成するために必要な情報について説明しています。 この資料は、すべての RPG 仕様への有効な入力を、桁ごとに、またキーワードごとに説明します。また、すべての命令コードと組み込み関数を詳細に説明しています。また、この資料は、RPG の論理サイクル、配列とテーブル、編集機能、および標識に関する情報を含みます。
- 「Intrasystem Communications Programming」  は、同じ iSeries サーバーにおける 2 つのアプリケーション・プログラム間の対話式通信について説明しています。この資料は、他のプログラムと通信するためにシステム内通信サポートを使用するプログラムにコーディングできる通信操作について説明しています。また、OS/400 システム間通信機能 (OS/400-ICF) を使用するシステム間通信アプリケーション・プログラムの開発に関する情報も提供します。
- 「iSeries 機密保護解説書」  は、正当な許可を受けていないユーザーにシステムとデータを使用されないようにするため、データを故意または偶発的な損傷または消滅から保護するため、セキュリティー情報を常に最新にするため、またはシステムにセキュリティーを設定するために、システム・セキュリティー・サポートを使用する方法について説明しています。
- iSeries Information Center の**システム管理**カテゴリーの中の実行管理のトピックは、実行管理環境の作成および変更の方法について説明しています。他のトピックとして、システムの調整、パフォーマンス・データ (レコード形式に関する情報および収集中のデータの内容を含む) の収集、システムの全体的な操作を制御または変更するためのシステム値の使用、および誰がシステムを使用し、どのリソースが使用されているか判定するためのデータを収集する方法に関する説明が含まれます。

索引

日本語, 数字, 英字, 特殊文字の順に配列されています。なお, 濁音と半濁音は清音と同等に扱われています。

[ア行]

[カ行]

[サ行]

[タ行]

[ナ行]

[ハ行]

[マ行]

[ヤ行]

[ラ行]

A

C

D

E

H

I

L

M

O

P

Q

R

S

U

[特殊文字]

Printed in Japan

SD88-5033-07

日本アイ・ビー・エム株式会社
〒106-8711　東京都港区六本木3-2-12