IBM i

プログラミング
ILE 概念

バージョン *7.4*

IBM i

プログラミング
ILE 概念

バージョン *7.4*

注記

本書および本書で紹介する製品をご使用になる前に、 255 ページの『特記事項』に記載されている情報をお読みください。

本書は、IBM i 7.3 (製品番号 5770-SS1)、および新しい版で明記されていない限り、以降のすべてのリリースおよびモディフィケーションに適用されます。このバージョンは、すべての RISC モデルで稼働するとは限りません。また、CISC モデルでは稼働しません。

本書にはライセンス内部コードについての参照が含まれている場合があります。ライセンス内部コードは機械コードであり、IBM 機械コードのご使用条件に基づいて使用権を許諾するものです。

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： IBM i
Programming
ILE Concepts
Version 7.4

発行： 日本アイ・ビー・エム株式会社

担当： トランスレーション・サービス・センター

目次

第 1 部 ILE 概念

本書では、IBM® i ライセンス・プログラムの統合化言語環境 (ILE) 体系に関する概念と用語について説明します。対象とするテーマには、モジュールの作成、バインディング、メッセージの処理、プログラムの実行とデバッグ、および例外の処理が含まれます。

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。

本書では、すべての ILE 言語に直接関連する IBM i の機能についても説明します。

既存の IBM i 言語から ILE 言語への移行については、本書では説明しません。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。

第 1 章 IBM i 7.3 での変更点

本書の主な変更点は次のとおりです。

- IFS ストリーム・ファイルからバインド・プログラム言語ソースを処理できるようになりました。
- 静的プロシージャー呼び出しで最大 16,383 個の引数を指定できるようになりました。

第 2 章 ILE 概念の PDF ファイル

本書の PDF ファイルを表示および印刷することができます。

本書の PDF 版を表示またはダウンロードするには、「ILE 概念」を選択します。

PDF ファイルの保管

表示または印刷するために PDF をご使用のワークステーションに保管するには、以下を実行します。

1. ご使用のブラウザーで、PDF のリンクを右クリックします。
2. PDF をローカルで保管するオプションをクリックします。
3. PDF を保管するディレクトリーにナビゲートします。
4. 「保管」をクリックします。

Adobe Reader のダウンロード

これらの PDF を表示または印刷するには、ご使用のシステムに Adobe Reader がインストールされている必要があります。Adobe Reader は、Adobe の Web サイト (www.adobe.com/products/acrobat/readstep.html) から無料でダウンロードできます。

第 3 章 統合化言語環境の概要

このトピックでは、統合化言語環境 (ILE) モデルを定義し、ILE の利点を示し、ILE が以前のプログラム・モデルからどのように発展したかを説明します。

ILE とは何か ?

ILE は、IBM i オペレーティング・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。

このモデルの機能は、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使用できます。 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、ソース・デバッガーなどがあります。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、呼び出し操作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。元のプログラマーが自分とは異なるスタイルでプログラムを作成している場合は、特に困難になります。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。この分離化により、テスト範囲を漏れなくカバーすることができます。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。

- 他のオペレーティング・システムからのコードの容易な移行

再使用可能なコンポーネント

ILE では、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できます。ILE 言語のいずれかで作成されたルーチンは、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。ご使用の環境で、任意の言語について独自の標準ルーチンのセットを開発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予定です。

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。IBM i Information Center のプログラミング・カテゴリーの中の API トピック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 153 ページの『第 12 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 (例えば、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、この制限はアプリケーションの設計担当者にとって作業上の制約になります。

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 を使用すれば、以下のいずれに対しても、リソースの有効範囲を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って、より優れた制御が可能なリソースの例です。

例として、アプリケーションのパフォーマンスを改善するため、プログラマーが顧客マスター・ファイルに対して共用 ODP を使用することに決定したとします。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であるため、いずれかのアプリケーションが原因となって、予期しない問題がもう一方のアプリケーションで発生する可能性が十分に考えられます。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・ファイルには、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、次のレコードを要求すると、D1 の次のレコードが表示されます。

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機能するアプリケーシ

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する必要があるが、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、オペレーティング・システムでプログラムがどのように実行されるかは、以下の要因の組み合わせによって異なります。

言語標準 (例えば、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 (12 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。

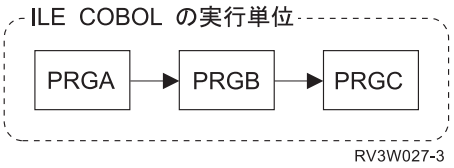

図 *1.* 実行単位における *3* つの *ILE COBOL* プログラム

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。

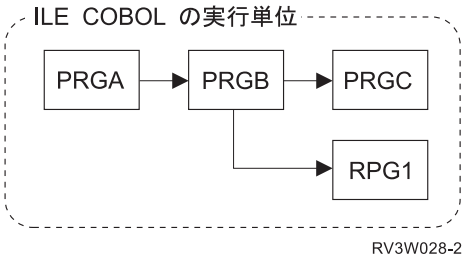

図 *2.* 実行単位における *3* つの *ILE COBOL* プログラムと *1* つの *ILE RPG* プログラム

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メモリーから消去されます。多くの場合はこれは問題ありません。しかし、RPG1 が送り状番号の発行を制御するユーティリティー・プログラムなどの場合、これは望ましい状態ではありません。

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。

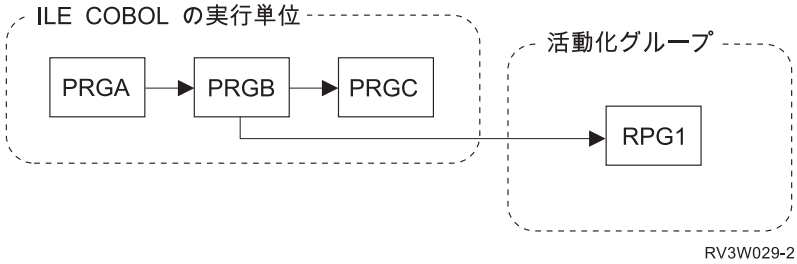

図 *3.* 別の活動化グループにおける *ILE RPG* プログラム

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

変換プログラムは、OPM プログラムに対してよりも多くの種類の最適化を ILE プログラムに対して行うことができます。

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。共通 ILE 変換プログラムの入力として使用される中間コードを使用することで、1 つの ILE 言語について変換プログラムに追加された最適化が、全 ILE 言語の役に立つ場合があります。

ILE の沿革

ILE は IBM i 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。OS/400® 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

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・オブジェクトを直接に作成および実行するために使用される機能、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)* と呼ばれています。

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、オペレーティング・システムが呼び出し要求を検出すると活動化されます。実行時での別のプログラムの呼び出しは、動的プログラム呼び出しです。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。

14 ページの図 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。リリース 6.1 では、BASIC コンパイラーは、使用できなくなりました。

OPM の境界を示す破線は、OPM が IBM i 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。例えば、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、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。

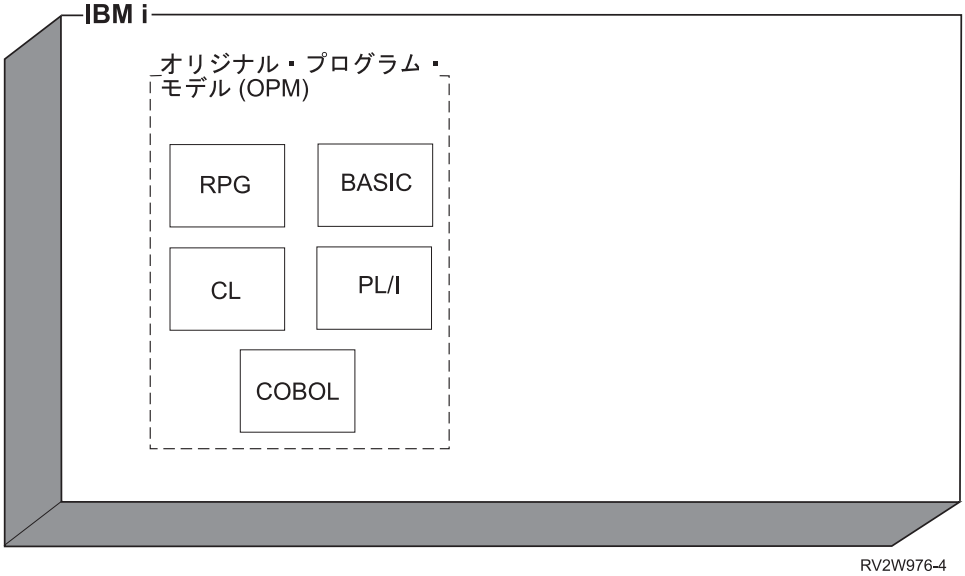

図 *4. OPM と IBM i の関係*

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。

- 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。オペレーティング・システムは、実行時にプログラム B を探し出し、プログラム B を使用す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 (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。

コンパイル単位間でプロシージャー呼び出しを定義する言語や、ローカル変数を使用してプロシージャーを定義する言語を、オペレーティング・システムで実行できるようにするために、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。

IBM i オペレーティング・システムでは、EPM コンパイラーを提供しなくなりました。

統合化言語環境の記述

図 5 に示したように、ILE は OPM と同様に、IBM i 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。

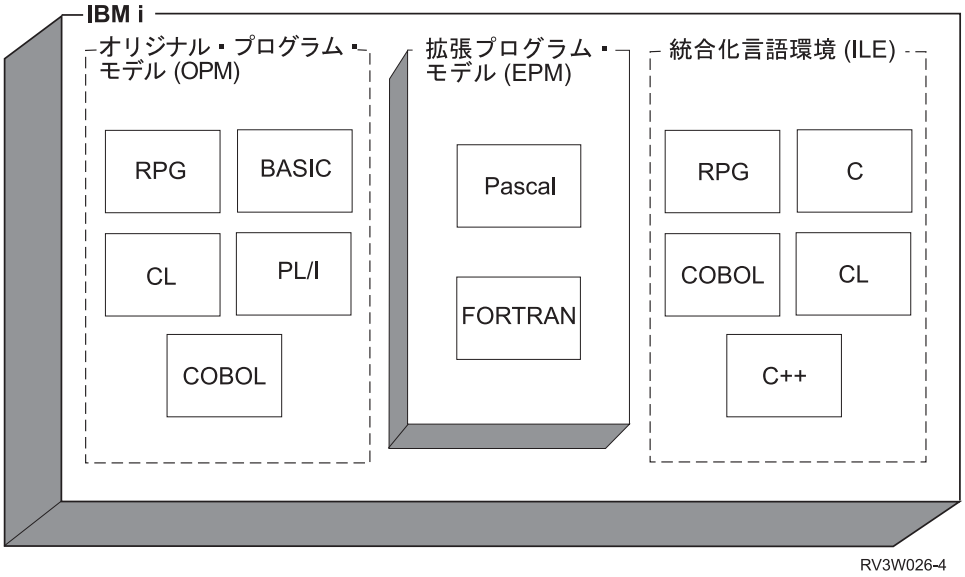

図 *5. OPM*、*EPM*、および *ILE* と *IBM i* の関係

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。例えば、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 18 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

  OPM 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。例えば、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 Microsoft Windows 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 22 ページの『サービス・プログラム』で詳しく説明します。

- 頻繁な呼び出し

  プロシージャー・ベースの言語で作成されたプログラムでは、呼び出しが頻繁に行われることがあります。

第 4 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) モデルを比較および対比させたものです。このトピックでは、以下の表に示す類似点と相違点について簡単に説明します。

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、更に、1 つ以上のプロシージャーを含みます (図 6 を参照)。

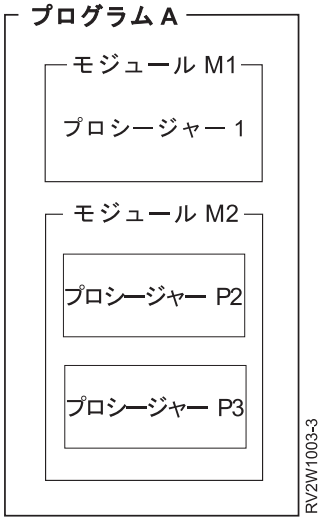

図 *6. ILE* プログラムの構造

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。例えば、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あり、実行できない 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可能なプロシージャーやデータ項目のコーディングについて詳しくは、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。

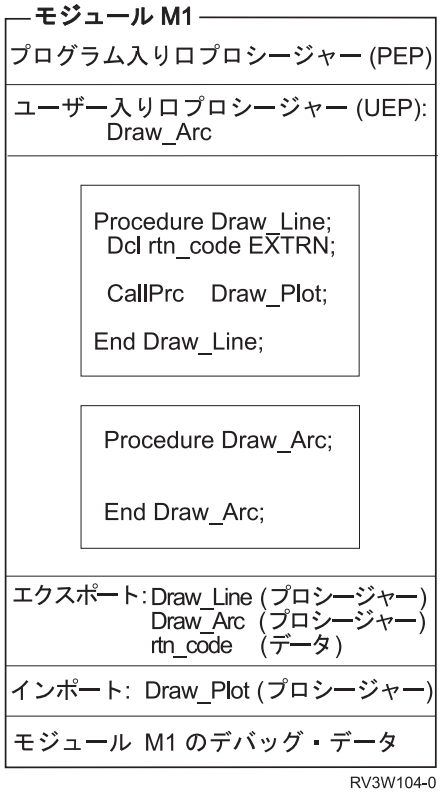

図 7. モジュールの概念図

*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 プログラムには、OPM プログラムと共通する以下のような特性があります。

- 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ときに、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。

ILE プログラムに対して動的プログラム呼び出しが行われると、プログラムの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・データを格納する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になります。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。

21 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、ILE デバッガーに必要なデータは、M1 と M3 の 2 つのモジュールだけにあります。モジュール M2 と M4 からのプロシージャーは、ILE デバッガーを使用してデバッグできません。

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。

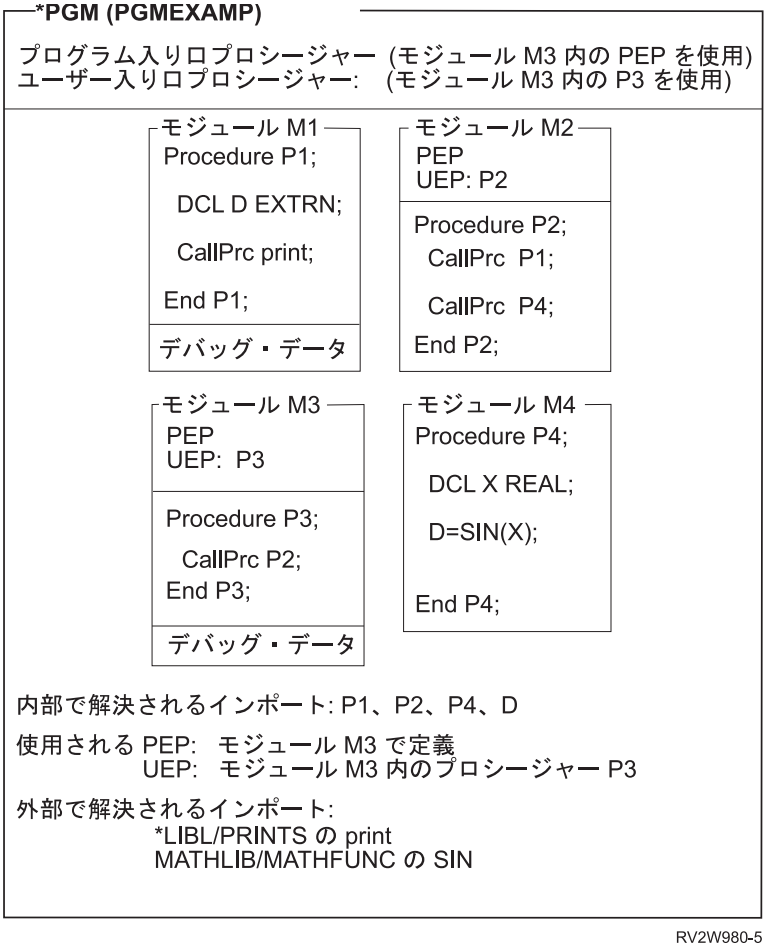

図 *8. ILE* プログラムの概念図

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が行われると、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 に関連付けられた 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、 22 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 オブジェクトで必要になる可能性がある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。オペレーティング・システムによって提供される一連のサービス・プログラムの例は、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、多くの場合、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 97 ページの『バインド・プログラム言語』を参照してください。

23 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、 21 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・プログラムは、動的プログラム呼び出しのターゲットとすることはできません。

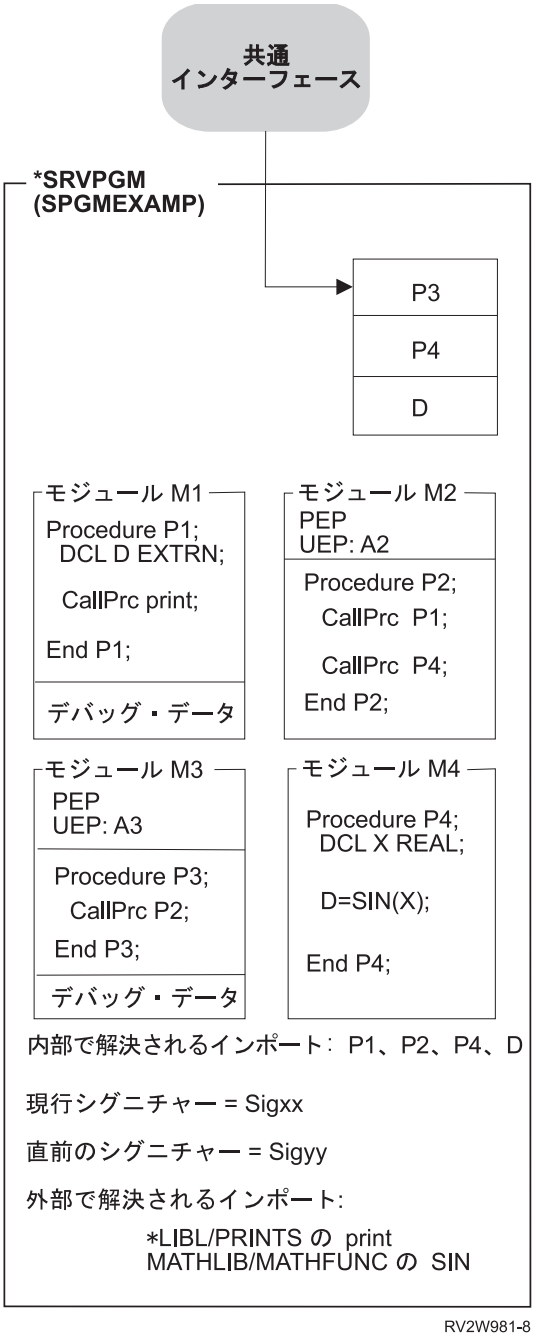

図 *9. ILE* サービス・プログラムの概念図

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・プログラムはデータ項目および実行可能なプロシージャーの集合です。
- 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 95 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リストされます。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・プログラムをリスト化する便利な方法を提供します。例えば、1 つのバインディング・ディレクトリーで、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・プログラムをリストすることができます。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・ディレクトリーにリストされ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 245 ページの『第 19 章 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。

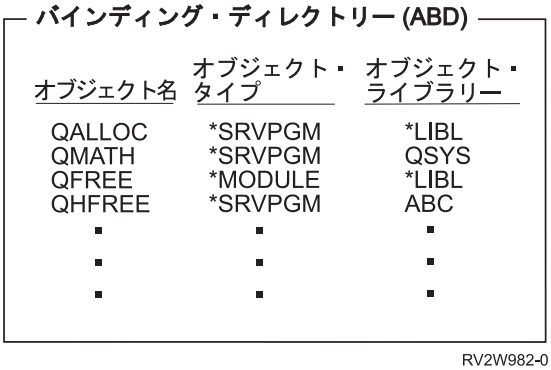

図 *10.* バインディング・ディレクトリーの概念図

*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「記号の定義が複数回指定された (Definition supplied multiple times for symbol)」がシグナル通知される場合があります。

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 の名前がリストされます。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のランタイム・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。

  例えば 23 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。

  23 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。

使用しているプロシージャーとデータ項目への物理リンクが確立されたことにより、実行時での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

28 ページの図 11 および 28 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。

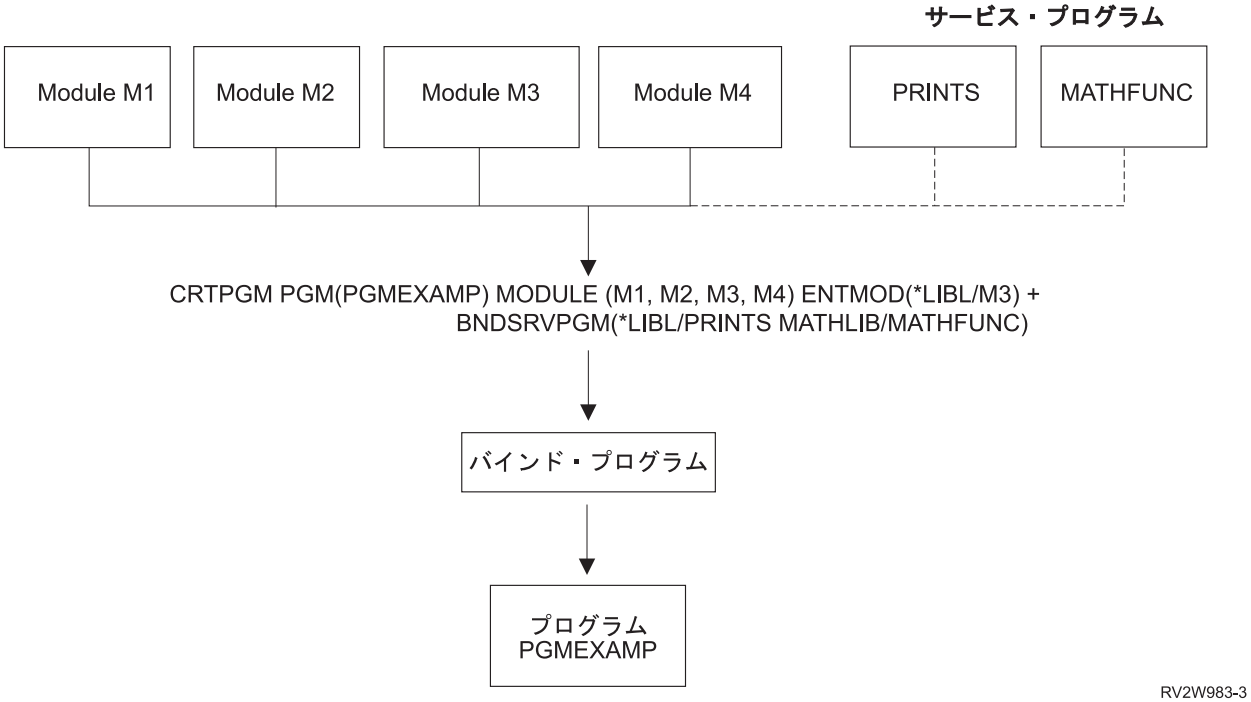

図 *11. ILE* プログラムの作成： (破線は、サービス・プログラムがコピーによってではなく、参照によってバインドされることを示しています。)

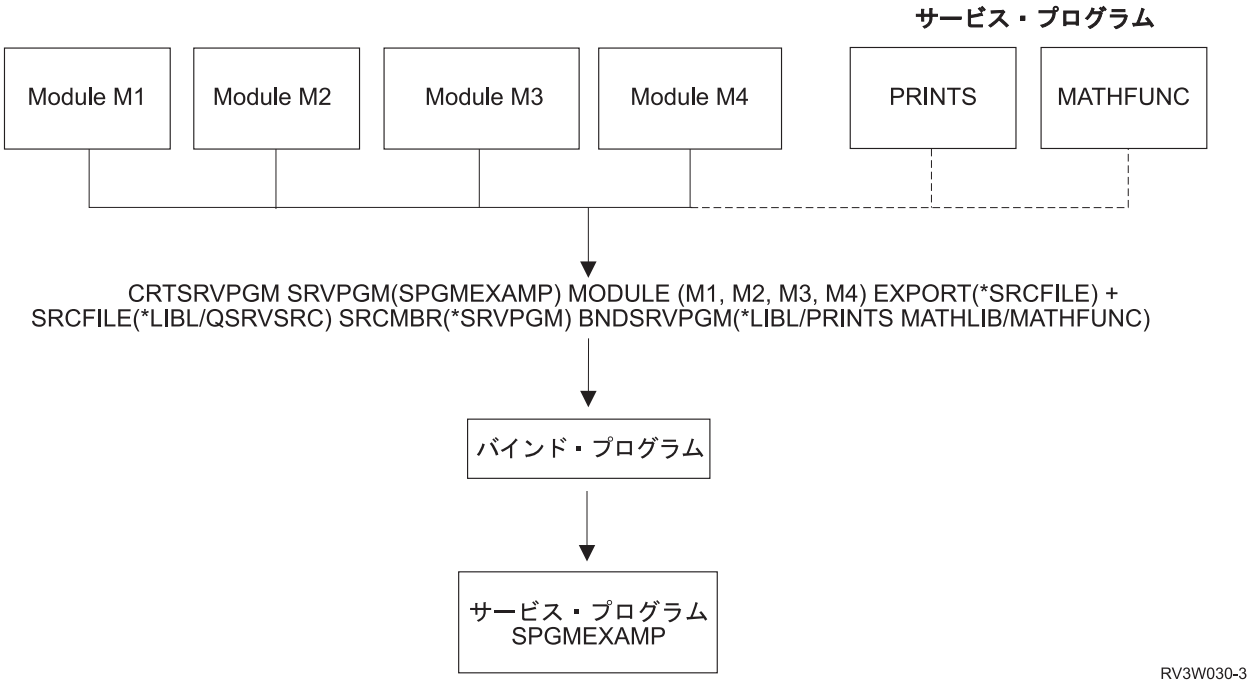

図 *12.* サービス・プログラムの作成： (破線は、サービス・プログラムがコピーによってではなく、参照によってバインドされることを示しています。)

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 79 ページの『第 7 章 プログラム作成の概念』を参照してください。

プログラムまたは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。

プロシージャー名の許可された長さについては、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が、ILE サービス・プログラムには渡しません。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、別の 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
- ILE プログラムは、OPM プログラムまたは別の ILE プログラムを呼び出すことができます
- 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ができます

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

30 ページの図 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。

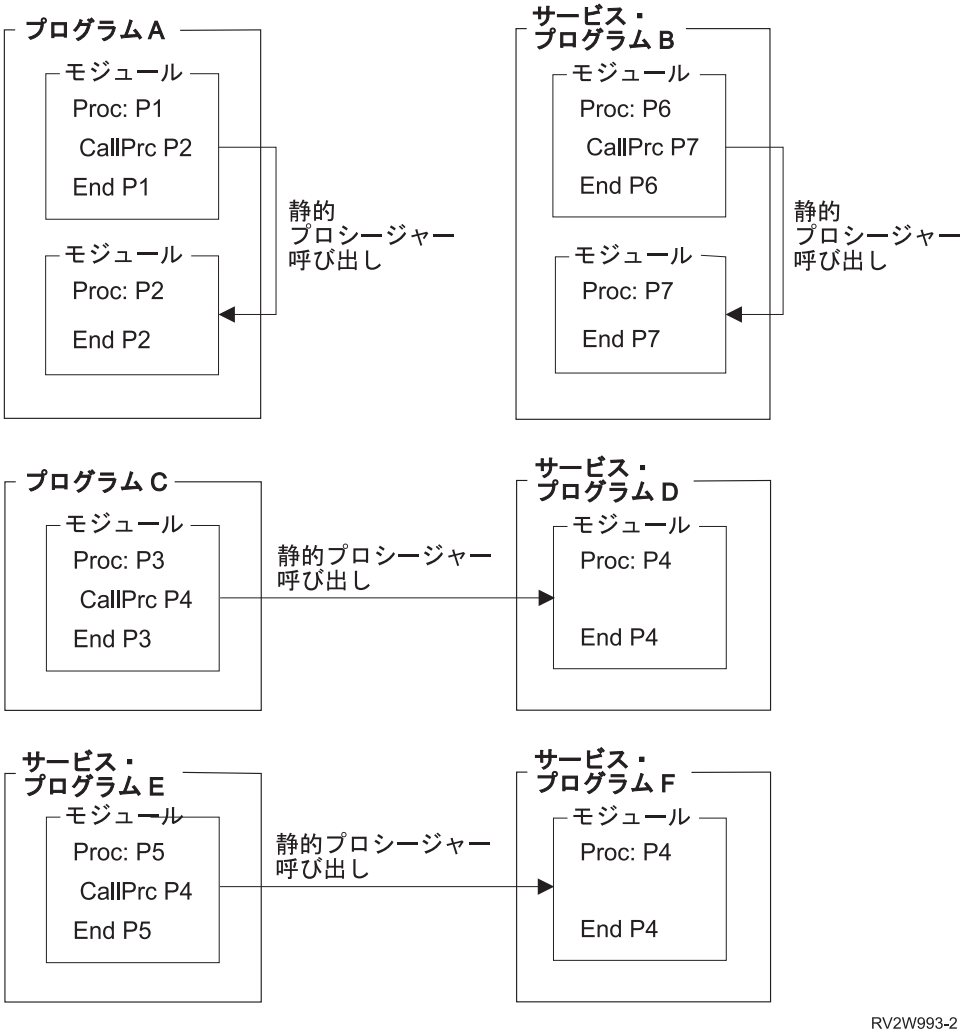

図 *13.* 静的プロシージャー呼び出し

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については、 125 ページの『第 9 章 プロシージャー呼び出しとプログラム呼び出し』を参照してください。

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。

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割り振って初期化します。
- 対応するサービス・プログラム・エクスポートの実行時アドレスへのインポートを解決します。

プログラムまたはサービス・プログラムは、同じジョブ内にあっても、複数の活動化グループ内で活動化することができます。各活動化は、特定の活動化グループに対してローカルなものであり、各活動化には独自の静的ストレージがあります。1 つのプログラムまたはサービス・プログラムが多くのジョブによって並行して使用された場合、そのオブジェクトの命令の 1 つのコピーのみがストレージに常駐しますが、静的変数は活動化ごとに分離されています。

デフォルトでは、サービス・プログラムは、直接的または間接的にそのサービスを必要とするプログラムに対する呼び出し中に、即時に活動化されます。 V6R1 以降について作成される ILE プログラムまたはサービス・プログラムをバインディングしているときは、サービス・プログラムについて、据え置き活動化を要求できます。サービス・プログラムについて据え置き活動化を要求すると、サービス・プログラムの活動化は、インポートされるプロシージャーの 1 つが呼び出されるまで据え置くことができます。プログラムの開始時とプログラム実行中の両方で活動化コストを最小化するには、プロシージャー・インポートを満たし、あまり移動しないコード・パスでのみ使用されるサービス・プログラムについて、据え置き活動化を指定することが推奨されます。

注**:**

1. データ・インポートを満たすサービス・プログラムについて据え置き活動化を要求する場合は、静的データを初期化するために、部分的に即時に活動化する必要があります。
2. プロシージャー・ポインター呼び出しで、プロシージャー・インポートを満たすサービス・プログラムについて据え置き活動化を要求する場合は、プロシージャー・ポインター呼び出しでバインディングを提供するために、部分的に即時に活動化する必要があります。

サービス・プログラムについて即時または据え置きのいずれかの活動化モードを指定するには、以下の CL コマンドの BNDSRVPGM パラメーターで、*IMMED または *DEFER を使用します。

- 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)
- サービス・プログラムの更新 (UPDSRVPGM)

バインディング・ディレクトリー追加 (ADDBNDDIRE) コマンドは、サービス・プログラムの項目に、類似の入力フィールドを提供します。バインディング・ディレクトリー項目での作業 (WRKBNDDIRE) コマンドは、バインディング・ディレクトリーにおけるサービス・プログラムの項目の活動化モードの出力を提供します。

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 36 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は、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 37 ページの『活動化グループ』を参照してください。

エラー処理の概要

33 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 50 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。例えば、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。

33 ページの図 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 50 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

最適化とは、オブジェクトのランタイム・パフォーマンスを最大限にすることです。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、最適化レベルが高くなると、オブジェクトの作成に必要な時間が長くなります。実行時では、高い最適化レベルのプログラムまたはサービス・プログラムは、低い最適化レベルで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、実行の速度が速くなります。

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)

パフォーマンス上の理由から、プログラムを稼働させるときは、高レベルの最適化の使用が望ましいでしょう。最初のテストでは、デバッグの制限のため、低い最適化レベルを使用する必要があるかもしれません。ただし、最終テストでは、プログラムがリリースされるときの最適化レベルの使用を強くお勧めします。これは、未初期化データなどの一部のバグが、高い最適化レベルでしか見つからない場合があるためです。

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 153 ページの『第 12 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 243 ページの『第 18 章 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オペレーティング・システムに存在しますが、OPM プログラムのデバッグにしか使用できません。ただし、 ILE デバッガーは、OPTION(*SRCDBG) または OPTION(*LSTDBG) のいずれかを使用してコンパイルされる OPM プログラムをデバッグできます。

最適化されたプログラムのデバッグは、難しい場合があります。 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る変数を表示または変更すると、デバッガーは、その変数のストレージ・ロケーションで、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ストレージに存在せず、デバッガーがアクセスできない場合があります。 したがって、変数に関して表示される値は現行値ではない可能性があります。このため、開発の過程でモジュールを作成するには、最適化レベル 10 (*NONE) を使用する必要があります。その後、最高のパフォーマンスを得るために、プログラムを稼働させる前に最終テストでモジュールを作成する際に、最適化レベル 30 (*FULL) または 40 を使用してください。

ILE デバッガーについての詳細は 153 ページの『第 12 章 デバッグに関する考慮事項』を参照してください。

第 5 章 ILE の拡張概念

このトピックでは、ILE モデルの拡張概念について説明します。このトピックを読む前に、 17 ページの『第 4 章 ILE の基本概念』で説明する概念をよく理解しておく必要があります。

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 44 ページの『サービス・プログラムの活動化』で記述します。

36 ページの図 15 は、永続ディスク装置に保管されているプログラム・オブジェクトを示しています。すべての IBM i オブジェクトと同様に、プログラム・オブジェクトも、異なるジョブで実行されている複数の並行ユーザーによって共用でき、プログラムの命令のコピーは 1 つだけ存在します。ただし、プログラムが活動化されるときに、プログラム変数のストレージの割り振りと初期化を行う必要があります。

36 ページの図 15 に示すように、それぞれのプログラム活動化には、これらの変数の独自のコピーがあります。

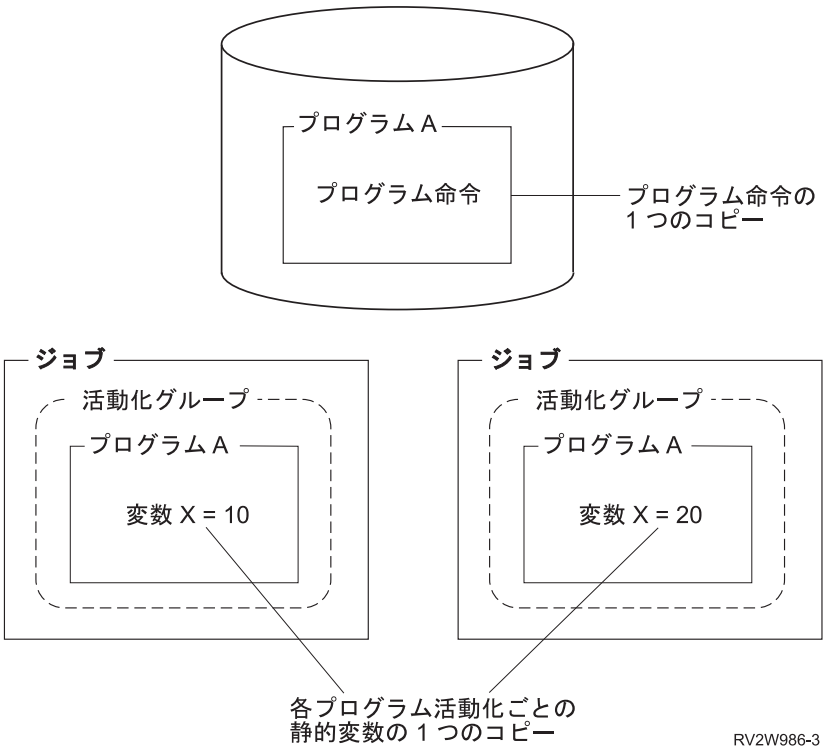

図 *15.* 各プログラム活動化ごとの静的変数の *1* つのコピー

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 37 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既に存在していれば、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、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。 37 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、 125 ページの『呼び出しスタック』を参照してください。

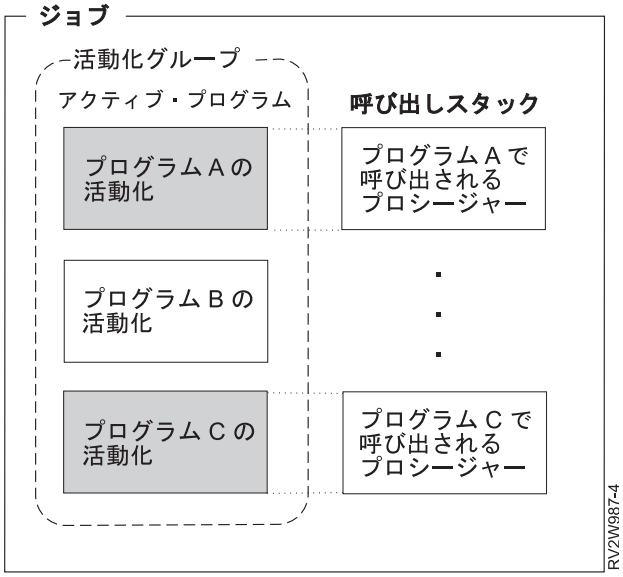

図 *16.* 呼び出しスタックにはないが活動化されている可能性があるプログラム

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・ストレージまたはテラスペースを使用します。詳細については 63 ページの『第 6 章 テラスペースおよび単一レベル・ストレージ』を参照してください。単一レベル・ストレージを使用した場合、静的プログラム変数および動的ストレージに、各活動化グループごとの個別のアドレス・スペースが割り当てられます。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) 通信

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 39 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。

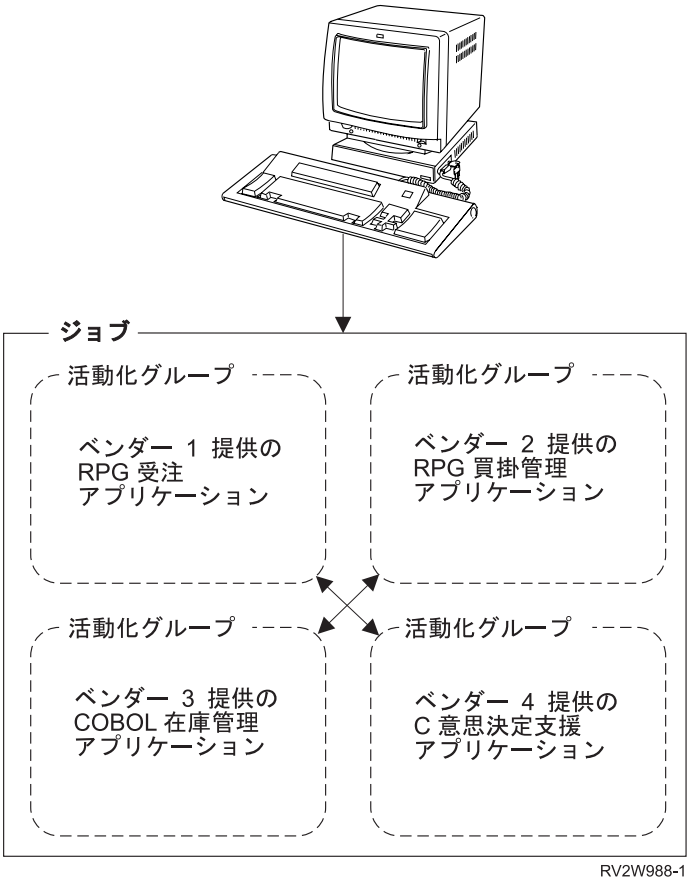

図 *17.* 活動化グループによる各ベンダーのアプリケーションの分離

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

デフォルト以外の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に、活動化グループ属性を指定することによって制御できます。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。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、CBLLE、または 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。

UPDPGM および UPDSRVPGM コマンドに ACTGRP パラメーターを使用して、プログラムまたはサービス・プログラムが活動化される活動化グループ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

ジョブが開始されるときに、システムはすべての 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。これらの活動化グループの 1 つがアプリケーション・プログラムに使用されます。それ以外は、オペレーティング・システム・プログラムに使用されます。これらの OPM の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・ストレージを使用します。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 オプションまたは DFTACTGRP(*YES) オプションを指定して作成された場合。

  注: DFTACTGRP(*YES) オプションは、CRTBNDCL (ILE CL) および CRTBNDRPG (ILE RPG) コマンドでのみ使用可能です。
- その ILE プログラムまたはサービス・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。

オペレーティング・システムは、必要であると判断すると、テラスペースのデフォルトの活動化グループも作成します。テラスペースのデフォルトの活動化グループは、静的プログラム変数にテラスペース・ストレージを使用します。テラスペースのデフォルトの活動化グループは、削除することができません。これはジョブの終了時にシステムにより削除されます。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、テラスペースの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。
- ILE プログラムまたは ILE サービス・プログラムの状態が *USER である。

ILE プログラムまたは ILE サービス・プログラムをテラスペースのデフォルトの活動化グループ内で活動化するには、さらに以下の条件のうちの 1 つを満たす必要があります。

- 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テラスペースのデフォルトの活動化グループで発生し、その ILE プログラムまたは ILE サービス・プログラムがストレージ・モデル *INHERIT またはストレージ・モデル *TERASPACE オプションのいずれかを指定して作成されている。
- ILE プログラムまたは ILE サービス・プログラムがストレージ・モデル *INHERIT オプションを指定して作成され、異なる活動化グループに関連した呼び出しスタックにアプリケーション項目がなく、以下の呼び出しのいずれかに備えて活動化が発生する。
  - SQL ストアード・プロシージャー
  - SQL 機能
  - SQL トリガー

  注: IBM i 7.1 からは、SQL のプロシージャー、関数、およびトリガーは、ストレージ・モデル *INHERIT で作成されます。前のリリースでは、SQL のプロシージャー、関数、およびトリガーはストレージ・モデル *SNGLVL で作成されていました。

- ILE プログラムまたは ILE サービス・プログラムがストレージ・モデル *TERASPACE オプションを指定して作成され、テラスペース・ストレージ・モデルの活動化グループに関連した呼び出しスタック項目がない。詳細については、 65 ページの『互換性のある活動化グループの選択』を参照してください。

デフォルトの活動化グループの 1 つで活動化された ILE プログラムによって使用される静的ストレージおよびヒープ・ストレージは、ジョブが終了するまで、システムに戻されません。同様に、いずれかのデフォルトの活動化グループに関連した一時データ管理機能リソースは通常、ジョブが有効範囲になります。例えば、通常、オープン・ファイルはジョブが終了するまではシステムによってクローズされません。詳細については、 122 ページの『ILE プログラムの場合のリソース再利用コマンド』を参照してください。

デフォルト以外の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、関連付けられた活動化グループを終了または削除せずに呼び出し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、別の活動化グループに関連した呼び出しスタック項目 (125 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

  例えば、COBOL の STOP RUN または C の exit()。
- API CEETREC の呼び出し
- 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

  例えば、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

  例えば、例外メッセージの送信や、ご使用の活動化グループに関連しない呼び出しスタック項目への分岐。

HLL 終了 verb を使用するか、API CEETREC を呼び出すことによって、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に関連した最も古い呼び出しスタック項目である場合 (ハード制御境界 とも呼ばれる)、上記の操作を行うと活動化グループは必ず削除されます。最も近い制御境界が最も古い呼び出しスタック項目でない場合 (ソフト制御境界 とも呼ばれる)、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 47 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。対照的に、システム指定活動化グループ内でこうした同じ操作を使用すると、活動化グループが削除されます。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、削除されます。活動化グループに関連付けられた最も古い呼び出しスタック項目からの通常の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。

図 18 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関連の活動化グループが削除されます。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関連の活動化グループが削除されることはありません。

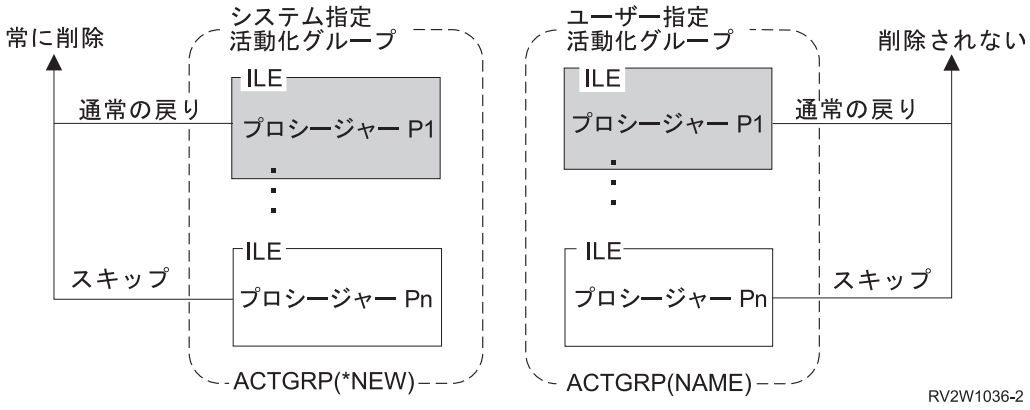

図 *18.* ユーザー指定活動化グループとシステム指定活動化グループの存続

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。

44 ページの図 19 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たプログラムに関連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。

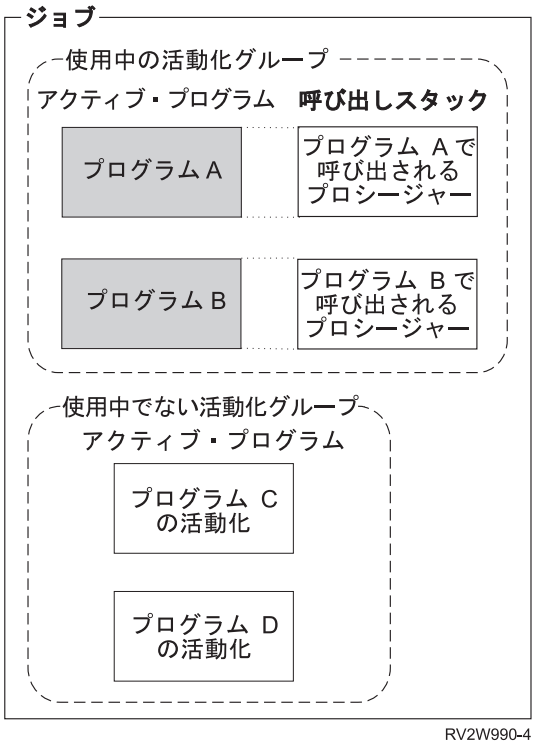

図 *19.* 呼び出しスタックに項目が存在する使用中の活動化グループ

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 57 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

システムは、サービス・プログラムを活動化するために、固有のステップに従います。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 35 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、即時活動化についてバインドされるサービス・プログラムに特有のものです。

- 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。

このような活動化処理は、インポートされたプロシージャーの 1 つが実行されるときに、据え置き活動化についてバインドされるサービス・プログラムについて実行されます。

活動化グループ内で最初に活動化されるプログラムは、サービス・プログラムへのバインディングについて検査されます。サービス・プログラムが、即時活動化につ

いて、活動化されるプログラムにバインドされる場合、サービス・プログラムは同じ動的呼び出し処理の一部としても活動化されます。サービス・プログラムが、据え置き活動化について、活動化されるプログラムにバインドされる場合、プロシージャー・インポートを満たすサービス・プログラムは、インポートされるプロシージャーの 1 つが呼び出されるまで、活動化されない場合があります。データ・インポートを満たすサービス・プログラムは、少なくとも部分的に、静的データを初期化するために活動化されます。 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 20 は、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。サービス・プログラム B および C は、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。

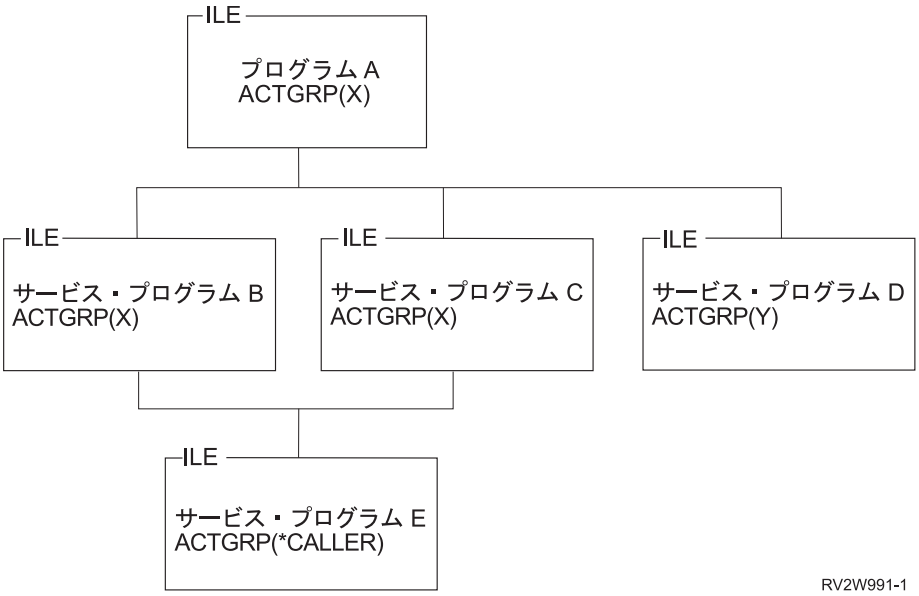

図 *20.* サービス・プログラムの活動化

これらのサービス・プログラムがすべて即時活動化についてバインドされるケースについて考えます。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 20 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・プログラムの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で、バインディングの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バインドされたサービス・プログラム・オブジェクトに対する実行権限が必要です。 45 ページの図 20 で、プログラム A およびすべてのバインドされたサービス・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、すべてのバインドされたサービス・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。

据え置き活動化についてプログラム A がサービス・プログラム B および D にバインドされ、即時活動化について C にバインドされるケースについて、 45 ページの図 20 を再度検討します。D は A についてのデータ・インポートを満たします。B は、静的プロシージャー呼び出しの場合のみ、A のプロシージャー・インポートを満たし、プロシージャー・ポインター呼び出しの場合は満たしません。次に、B は据え置き活動化についてサービス・プログラム E にバインドされ、C は即時活動化について E にバインドされます。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。

- サービス・プログラム C、D、および E が、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・プログラムを作成するときに指定できます。D が検索されるのは、データ・インポートを満たし、少なくとも静的データが初期化される時点までに活動化される必要があるためです。
- E は C の代わりに活動化されます。B が実行され、E 内のプロシージャーを呼び出すとき、 E は再度活動化されません。サービス・プログラムの活動化は、1 つの活動化グループで一度しか行われないためです。
- 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。
- シグニチャー・チェックは、即時活動化についてバインドされるか、または A が活動化されるときに部分的または完全な即時活動化を必要とする、全プログラムおよびサービス・プログラム間で行われます。この例では、C、D、および E についてシグニチャー・チェックが行われます。

サービス・プログラム B の検索および B についてのシグニチャー・チェックは、インポートされるプロシージャーの 1 つが呼び出されるまで行われません。

プログラム A の呼び出し元の現行の権限は、プログラム A およびサービス・プログラム C、D、および E に対する権限の検査に使用されます。サービス・プログラム B の呼び出し元の現行の権限は、B に対する権限の検査に使用されます。B の権限検査の結果は、B が即時活動化についてバインドされるケースの結果と異なることがあります。

制御境界

ILE は、未処理の機能チェックが発生するか、HLL 終了 verb が使用されるか、API CEETREC が呼び出されると、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。『活動化グループの制御境界』および 48 ページの『OPM と ILE の呼び出しスタック項目間の制御境界』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活動化グループに関連している 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 プログラムに対するものである ILE 呼び出しスタック項目。

活動化グループの制御境界

この例は、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。

48 ページの図 21 は、2 つの活動化グループ、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 の呼び出しは、制御境界になる可能性があります。例えば、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。

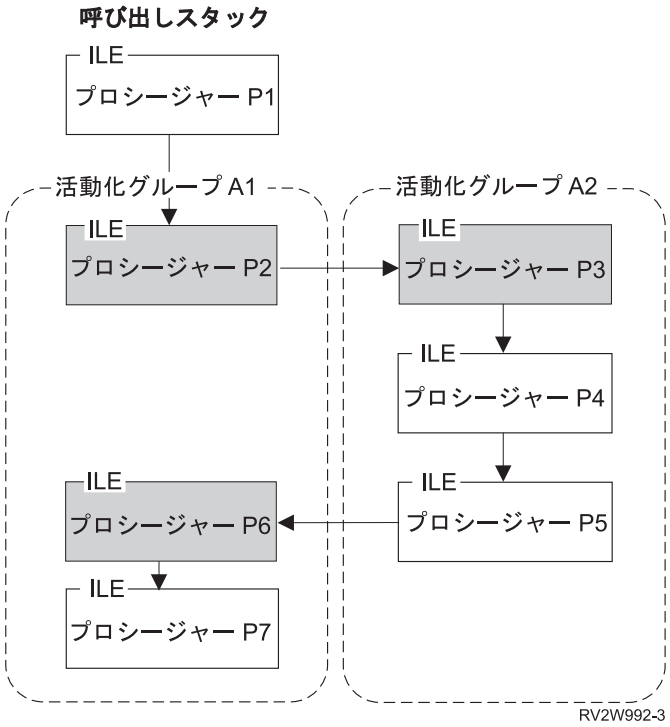

図 *21.* 制御境界： (陰影の付いたプロシージャーが制御境界です。)

注**:** 制御境界は、デフォルトの活動化グループを含むすべての活動化グループの間で定義されます。例えば、テラスペースのデフォルトの活動化グループに関連した呼び出しスタック項目が OPM デフォルト活動化グループに関連した呼び出しスタック項目の直前にある場合、2 つの呼び出しスタック項目間で制御境界ができます。

OPM と ILE の呼び出しスタック項目間の制御境界

この例では、ILE 呼び出しスタック項目と OPM 呼び出しスタック項目間の制御境界を定義する方法を示しています。

49 ページの図 22 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 の呼び出しは、前の呼び出しスタック項目が OPM プログラム A および B に関連しているので、制御境界になる可能性があります。

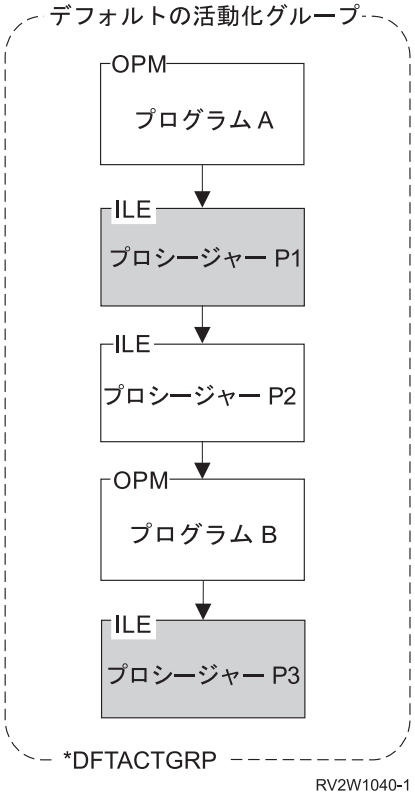

図 *22. ILE* 呼び出しスタック項目と *OPM* 呼び出しスタック項目間の制御境界： (陰影の付いたプロシージャーが制御境界です。)

制御境界の使用

ILE HLL 終了 verb を使用するか、API CEETREC を呼び出すと、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 50 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デフォルト以外の活動化グループに関連した最も古い呼び出しスタック項目である場合、HLL 終了 verb、CEETREC API の呼び出し、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デフォルト以外の活動化グループに関連した最も古い呼び出しスタック項目でない場合、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。

48 ページの図 21 は、各活動化グループ内に関連した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 (ただし P6 でも P7 でもない) で HLL 終了 verb を使用するか、API CEETREC を呼び出すと、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 23 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 143 ページの『第 11 章 例外および条件管理』を参照してください。図 23 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 *23. 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、各ジョブ内の呼び出しスタック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、各ジョブには 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。

API を使用した例外メッセージの送受信については、IBM i Information Center のプログラミング・カテゴリーの中の API トピック・コレクション内の『Message Handling APIs』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。

これらのメッセージ・タイプとその他のメッセージ・タイプについては、IBM i Information Center のプログラミング・カテゴリーの中の API トピック・コレ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

  オペレーティング・システム (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、追加機能が定義されています。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。 54 ページの『例外ハンドラーのタイプ』のトピックでは、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。 例外メッセージを変更する方法については、IBM i Information Center のプログラミング・カテゴリーの中の API トピック・コレクション内の『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 143 ページの『第 11 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。

50 ページの図 23 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。

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。未処理例外が制御境界にパーコレートするときに取られる処理ステップは、例外のタイプによります。

1. 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。
2. 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。
3. エスケープ例外の場合には、再開点に特殊な機能チェック・メッセージが送られます。この機能チェックは、処理するか、制御境界にパーコレートすることができます (以下参照)。
4. 機能チェックの場合には、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みなします。制御境界までのすべての呼び出しスタック項目が取り消されます。制御境界がデフォルト以外の活動化グループに関連した最も古い呼び出しである場合、活動化グループが終了します。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。このメッセージのメッセージ ID は CEE9901 であり、ILE によって制御境界の直前の呼び出しスタック項目に送られます。

54 ページの図 24 は、ILE 内での未処理のエスケープ例外を示しています。この例では，プロシージャー P1 の呼び出しは制御境界であり、関連した活動化グループの最も古い呼び出し項目でもあります。プロシージャー P4 には、エスケープ例外が発生します。エスケープ例外は P1 制御境界にパーコレートし、結果として、特殊な機能チェック・メッセージがプロシージャー P3 の再開点に送られます。機能チェックは処理されず、P1 制御境界にパーコレートします。アプリケーションは、終了したものとみなされ、活動化グループは破棄されます。最終的に CEE9901 エスケープ例外が制御境界 (プログラム A) の直前の呼び出しスタック項目に送られます。

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。

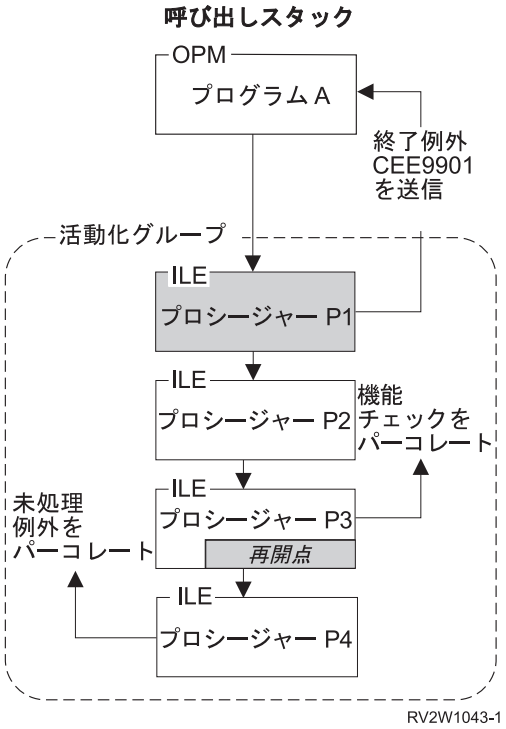

図 *24.* 未処理例外のデフォルト・アクション

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。 50 ページの図 23 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
```

上記のようなステートメントは、HLL 固有のハンドラーであるにもかかわらず、直接モニター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 条件ハンドラーは、実行時に例外ハンドラーを登録することを可能にします。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 を使用すると、例外が発生したときに制御が渡される実行時のプロシージャーを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。例えば、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。

57 ページの図 25 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、RPG (E)、MONITOR、INFSR、およびエラー標識、また 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。例えば、 57 ページの図 25 のプロシージャー P2 が HLL 固有ハンドラーを持ち、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、P2 のハンドラーが P1 の直接モニター・ハンドラーの前に考慮されます。

   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。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。例外は、直接モニター・ハンドラーの説明で言及した HLL 固有ハンドラーです。

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。例えば、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。

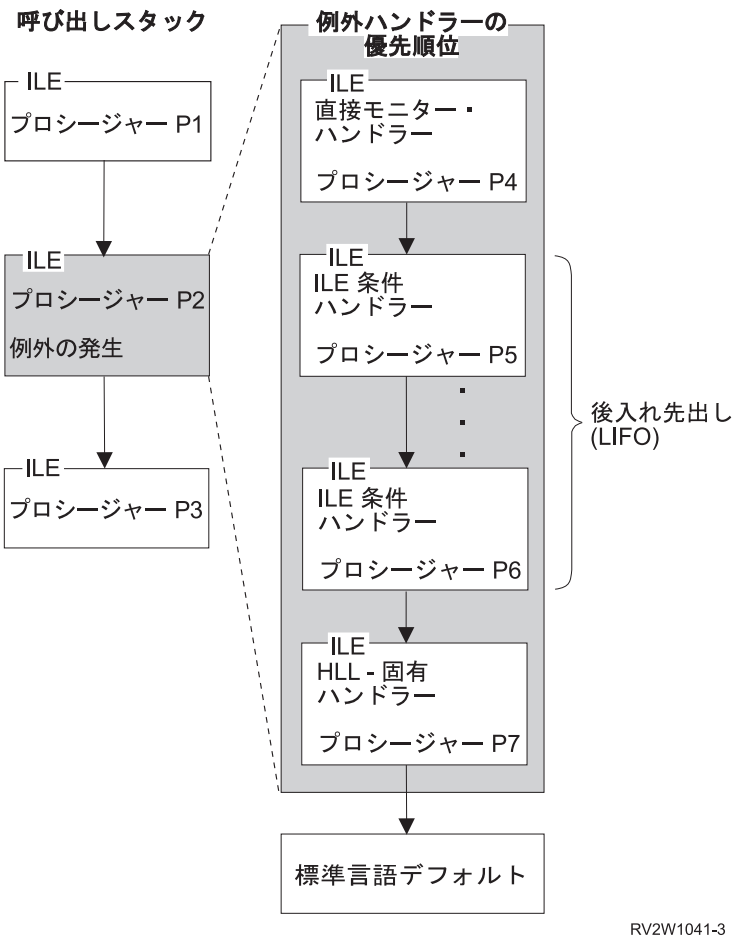

図 *25.* 例外ハンドラーの優先順位

ILE 条件

システム間の整合性の向上のために、ILE では、エラー条件の処理に使用できる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。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、基礎になる例外メッセージと条件を関連付けることができる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 143 ページの『第 11 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。例えば、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。例えば、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 159 ページの『第 13 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。 59 ページの図 26 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。

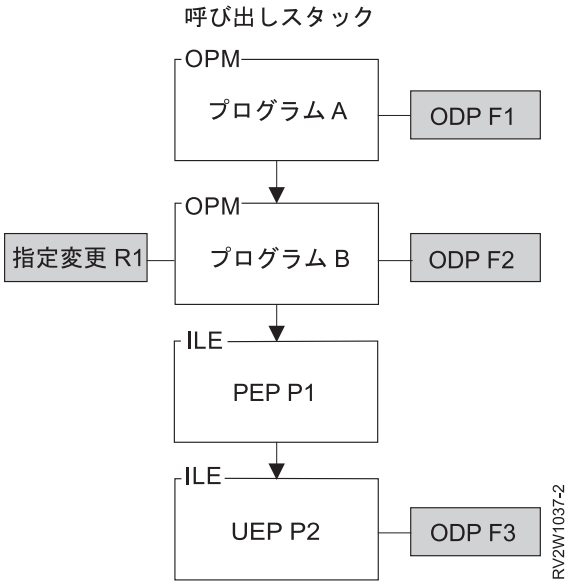

図 *26.* 呼び出しレベルの有効範囲指定： (ODP および指定変更の有効範囲は、呼び出しレベルにすることができます。)

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 プログラムまたはプロシージャー呼び出しに関連した活動化グループに接続された時点で行われます。活動化グループが削除されると、データ管理は、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。 60 ページの図 27 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。例えば、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。

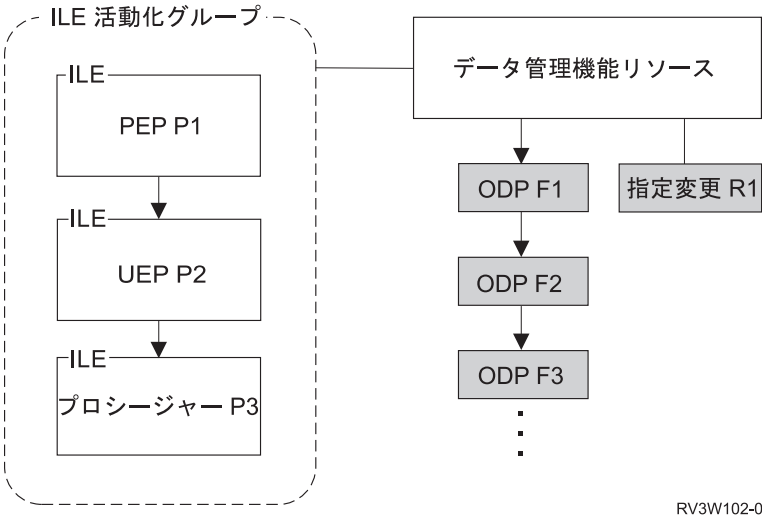

図 27. 活動化グループ・レベルの有効範囲指定： (ODP および指定変更の有効範囲は活動化グループにすることができます。)

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。例えば、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。

61 ページの図 28 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。

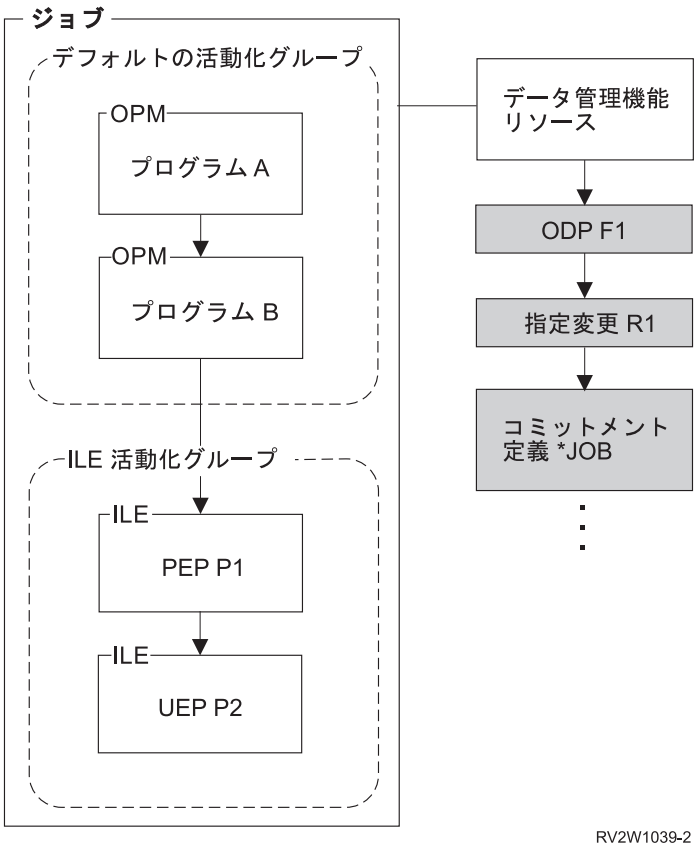

図 *28.* ジョブ・レベルの有効範囲指定： (ODP、指定変更、およびコミットメント定義の有効範囲はジョブ・レベルにすることができます。)

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 159 ページの『第 13 章 データ管理機能の有効範囲指定』にあります。

第 6 章 テラスペースおよび単一レベル・ストレージ

ILE プログラムを作成する際に、一部のコンパイラーで以下のストレージ・モデルの 1 つを選択できます。

- 単一レベル・ストレージ
- テラスペース
- 継承

継承ストレージ・モデルは、プログラムが活動化される活動化グループのストレージ・モデル (テラスペースまたは単一レベル・ストレージのいずれか) を、そのプログラムが採用することを示します。ILE プログラムは、デフォルトでは単一レベル・ストレージを使用します。

このトピックでは、テラスペースのオプションに焦点を合わせます。

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、同じジョブ内でスペース・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・ストレージの比較を示しています。

表 2. テラスペースと単一レベル・ストレージの比較

属性	テラスペース	単一レベル・ストレージ
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	合計約 100 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 (例えば、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。

ストレージでのテラスペースの使用

プログラムのストレージ・モデル は、自動、静的、および固定のストレージに使用されるストレージのタイプを決定します。

デフォルトでは、コンパイラーは、ストレージ・モデルと一致するヒープ・ストレージ・インターフェースも使用します。ただし、一部のコンパイラーでは、プログラムのストレージ・モデルとは無関係にヒープ・ストレージのタイプを選択できます。

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、ヒープ・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。例えば、`malloc()` は `_C_TS_malloc()` にマップされます。

ILE RPG コンパイラーでは、制御仕様書で ALLOC キーワードを使用すると、ストレージ・モデルとは無関係にヒープ・ストレージのタイプを明示的に設定することができます。

これらのコンパイラーのオプションについて詳しくは、「ILE C/C++ プログラマーの手引き」 または「ILE RPG プログラマーの手引き」 を参照してください。

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 ILE プログラムをオプションで作成できます。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・モデルについての詳細は 71 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、プログラム、およびサービス・プログラムについて、一部のコンパイラーで以下のストレージ・モデルの 1 つを指定するオプションがあります。

- 単一レベル・ストレージ (*SNGLVL)
- テラスペース (*TERASPACE)
- 継承 (*INHERIT)

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

RPG、COBOL、C または C++ プログラムのテラスペース・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. C および C++ の場合、モジュールの作成時に、TERASPACE パラメーターに *YES を指定する。

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. 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 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 67 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。

   モジュールを 1 つだけ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、バインド C プログラム作成 (CRTBNDC)、バインド C++ プログラム作成 (CRTBNDCPP)、バインド RPG プログラム作成 (CRTBNDRPG) およびバインド COBOL プログラム作成 (CRTBNDCBL) 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。

   CRTPGM および 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、プログラムまたはサービス・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・ストレージまたはテラスペースを使用できるようにする方法で、プログラムまたはサービス・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、プログラムまたはサービス・プログラムは単一レベル・ストレージ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。例えば、すべての静的変数の合計サイズは、単一レベル・ストレージ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	可、ただし ACTGRP(*CALLER) を指定した場合のみ	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・ストレージ・モデルのプログラムは、単一レベルの自動、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・ストレージも使用します。

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、ルート・プログラムが単一レベル・ストレージ・モデルを使用する活動化グループ (これには OPM デフォルト活動化グループが含まれます) には活動化することはできません。単一レベル・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	許可されない。	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。

- 67 ページの『モジュールのバインディングに関する規則』
- 67 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』
- 68 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』
- 68 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	単一レベル・ストレージ
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・ストレージ
	単一レベル・ストレージ	エラー	エラー	単一レベル・ストレージ

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・ストレージ
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[1]
	継承[1]	可[2]	可	可[2]
	単一レベル・ストレージ	可[1]	可	可

注:

1. 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。例えば、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。
2. 呼び出し側プログラムまたはサービス・プログラムが継承ストレージ・モデルを使用し、ターゲット・サービス・プログラムが単一レベル・ストレージ・モデルまたはテラスペース・ストレージ・モデルを使用する場合は、ターゲット・サービス・プログラムが活動化される活動化グループが、ターゲット・サービス・プログラムと同じストレージ・モデルを必ず持つようにしてください。例えばこのようになります。サービス・プログラム A が継承ストレージ・モデルで作成されます。サービス・プログラム B はテラスペース・ストレージ・モデルで作成され、*CALLER 活動化グループ属性を持ちます。サービス・プログラム A がサービス・プログラム B にバインドされます。このケースでは、サービス・プログラム A は必ず、テラスペース・ストレージ・モデルを持つ活動化グループに活動化される必要があります。

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・モデルを指定するサービス・プログラムは、単一レベル・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。それらは、このトピックで説明する規則や制約事項に準拠する限り、バインドしてデータを共用することができます。

ストレージ・モデルを継承するためのプログラムまたはサービス・プログラムの変換

ストレージ・モデルを継承するようにプログラムまたはサービス・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。プログラムのコードが、テラスペースと単一レベル・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、 71 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。

既存のプログラムまたはサービス・プログラムをテラスペース・ストレージ・モデルに使用可能にするには、2 つの方法があります。つまりそれらを再作成する方法と、既存のプログラムのストレージ・モデルを変更する方法 (場合による) です。プログラムを作成する場合、まず *INHERIT 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します。次に *INHERIT ストレージ・モデルを使用して ILE プログラムまたは ILE サービス・プログラムを作成します。

代わりに、以下の条件が当てはまると、CHGPGM または CHGSRVPGM コマンドを使用して、既存のプログラムまたはサービス・プログラムのストレージ・モデルを単一レベル・ストレージ (*SNGLVL) から *INHERIT に変更することができます。

1. オブジェクトが ILE プログラムまたは ILE サービス・プログラムである。
2. オブジェクトが単一レベル・ストレージ・モデルを使用する。
3. オブジェクトがその呼び出し元の活動化グループを使用する。
4. オブジェクトが V6R1M0 以降のターゲット・リリースのプログラムである。
5. オブジェクトが V5R1M0 以降のターゲット・リリースのサービス・プログラムである。
6. オブジェクト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 V5R1M0 以降である。

プログラムの更新: テラスペースに関する考慮事項

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	不可	可
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
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

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
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。ただし、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。例えば、関数に対するポインター引数が、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。

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

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

- 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。そうでない場合は、MCH0609 例外がシグナル通知されます。
- 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、例えば、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、例えば、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。

このトピックでは、理想的なテラスペース環境に移行するために実行できる各種方法について説明します。また、このトピックでは、単一レベルのストレージを使用するプログラムと、テラスペースを使用するプログラムが混在する環境での潜在的な問題を最小化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・ストレージ・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 (例えば、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) 74 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・ストレージ・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 74 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。

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・ストレージが使用されるので、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・ストレージ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』

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・ストレージ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・ストレージ・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。

IBM i のインターフェースおよびテラスペース

ポインター・パラメーターを持つ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 (例えば、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 (例えば、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 は、テラスペースの割り振りおよび廃棄用にバインド可能 API を提供しています。[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_malloc64() では、8 バイト・サイズ値を使用して、テラスペース・ストレージ割り振りを指定できます。
- _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の詳細は、IBM i Information Center の『*UNIX-type APIs*』のトピック (プログラミング・カテゴリーおよび API トピックの下) を参照してください。

3. STGMDL(*SNGLVL) を指定すると、malloc()、free()、calloc()、および realloc() をそのテラスペース・バージョンに自動的にマップするために、テラスペース・コンパイラー・オプション TERASPACE(*YES *TSIFC) が ILE C および C++ コンパイラーから使用可能です。

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。

- *TGTRLS(V5R4M0)* パラメーターを使用してプログラムを作成する場合、他のすべてのプログラムがテラスペース・アドレスを処理できる場合以外は、これらの他のプログラムへのテラスペース・アドレスの引き渡しに対する依存関係を作成しないようにしてください。 IBM i 6.1 以降で実行されるプログラムはすべて、テラスペース・アドレスを使用できます。
- 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。
  - MATBPGM
  - MATPG
  - SCANX (一部のオプションのみが制限される)
- 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、POWER6® 以前のプロセッサー用に生成された一定のコードで発生する場合があります。 198 ページの『適応コード生成』を参照してください。特に結果が開始アドレスより低く 16 MB 境界に満たないテラスペース・アドレス値である場合、16 バイト・ポインターでのアドレス計算で EAO が発生することがあります。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。別の 16 MB エリアでより小さい値を計算する、頻繁なテラスペースのアドレス計算は避けてください。あるいは、 190 ページの『ライセンス内部コードのオプション』で説明するように、MinimizeTeraspaceFalseEAOs LICOPT を使用してプログラムを作成してください。

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

  ヒープ・ストレージにテラスペースが使用されるように、`_C_TS_malloc` を使用するか、プログラムを作成します (64 ページの『ストレージでのテラスペースの使用』を参照)。
- 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、さらに言語でサポートされていれば 8 バイト・ポインターのデータ・モデルを使用します。

- 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。

- 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・モデルを使用し、さらに 8 バイト・ポインターのデータ・モデルをサポートする言語を使用すると、メモリー・フットプリントが削減し、ポインター操作の速度が上がります。

- 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・モデルを使用し、さらに 8 バイト・ポインターのデータ・モデルをサポートする言語を使用します。

- 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・ストレージを使用する以外に選択肢がない場合があります。例えば、プロセス間通信用のユーザー・データを保管するために、単一レベル・ストレージが必要な場合があります。単一レベル・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。

  – QUSCRTUS API または 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
  – プログラミング言語での単一レベル・ストレージ・ヒープ・インターフェース
  – プログラムに渡された単一レベル・ストレージ参照
  – ALCHS MI 命令から獲得された単一レベル・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 69 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・ストレージ・モデル・モジュールの組み込み

  単一レベル・ストレージ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 71 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・ストレージ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・ストレージ・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。

- シナリオ *11:* 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、変換は行われません。例えば、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 iconv(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 = iconv(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 *12:* 関数の再宣言

  IBM 提供のヘッダー・ファイルで既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。通常、ローカル宣言では、ポインターの長さが正しく指定されません。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 IBM i 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 *13:* 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。

第 7 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。このプロセスには、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。

このトピックでは、バインド・プログラム、および 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する概念について説明します。このトピックを読む前に、 17 ページの『第 4 章 ILE の基本概念』で説明するバインディングの概念をよく理解しておく必要があります。

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) MODULE(*PGM)	SRVPGM(*CURLIB/UTILITY) 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) BNDDIR(*NONE)	BNDSRVPGM(*NONE) BNDDIR(*NONE)
実行時	ACTGRP(*ENTMOD)	ACTGRP(*CALLER)
最適化	IPA(*NO) IPACTLFILE(*NONE) ARGOPT(*NO)	IPA(*NO) IPACTLFILE(*NONE) ARGOPT(*NO)

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) DETAIL(*NONE) ALWUPD(*YES) ALWLIBUPD(*NO) ALWRINZ(*NO) REPLACE(*YES) AUT(*LIBCRTAUT) TEXT(*ENTMODTXT) TGTRLS(*CURRENT) USRPRF(*USER) STGMDL(*SNGLVL)	OPTION(*GEN *NODUPPROC *NODUPVAR *WARN *RSLVREF) DETAIL(*NONE) ALWUPD(*YES) ALWLIBUPD(*NO) ALWRINZ(*NO) REPLACE(*YES) AUT(*LIBCRTAUT) TEXT(*BLANK) TGTRLS(*CURRENT) USRPRF(*USER) 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 (91 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る必要があるのは、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だけです。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、借用権限の使用 (USEADPAUT(*YES)) 属性でプログラムを作成できるユーザーを定義します。 システム値 QUSEADPAUT によって許可されているすべてのユーザーは、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。必要な権限を確認するには、「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。指定した権限リストに対して

*USE 以上の権限を持っているユーザーは、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	説明
authorizationlist name	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。 • プログラムまたはサービス・プログラムが作成されているときに、他のエラーがない。 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。

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。プログラムを作成または変更する際に使用する最適化パラメーターについての詳細は、 171 ページの『第 15 章 拡張最適化技法』を参照してください。

モジュールおよびプログラムに保管されるデータ

モジュールが作成されると、コンパイルされたコードのほかにも有益なデータがモジュール内に格納されます。使用されたコンパイラーに応じて、モジュールには以下のタイプのデータが含まれることがあります。

- 作成データ (*CRTDTA)

  モジュールの再作成または変換に必要です (例えば、最適化レベルを変更するため)。新規に作成されたモジュールには、常に作成データが含まれます。
- デバッグ・データ (*DBGDTA)

  モジュールがバインドされるプログラムのデバッグに必要です。デバッグについての詳細は、 153 ページの『第 12 章 デバッグに関する考慮事項』および 180 ページの『プロシージャー間分析 (IPA)』を参照してください。
- 中間言語データ (*ILDTA)

プロシージャー間分析 (IPA) の拡張最適化手法に必要です。詳しくは、 171 ページの『第 15 章 拡張最適化技法』を参照してください。

モジュールの表示 (DSPMOD) コマンドを使用して、モジュールに保管されているデータの種類を調べます。バインディングのプロセスで、モジュール・データは、作成されるプログラムまたはサービス・プログラムにコピーされます。中間言語データ (*ILDTA) はプログラムにもサービス・プログラムにもコピーされません。

プログラム表示 (DSPPGM) またはサービス・プログラムの表示 (DSPSRVPGM) コマンドで DETAIL(*MODULE) パラメーターを指定して、詳細を表示します。プログラムには、モジュール・データに加えて、以下の種類のデータが含まれている可能性があります。

- 作成データ (*CRTDTA)

  プログラムまたはサービス・プログラムの再作成に必要です。新規に作成されたプログラムには、常にこの作成データが含まれます。
- ブロック順プロファイル作成データ (*BLKORD)

  アプリケーション・プロファイルについて生成されます。詳しくは、 171 ページの『第 15 章 拡張最適化技法』を参照してください。
- プロシージャー順プロファイル作成データ (*PRCORD)

  アプリケーション・プロファイルについて生成されます。詳しくは、 171 ページの『第 15 章 拡張最適化技法』を参照してください。

作成データ (*CRTDTA) は、プログラム自体およびバインドされた各モジュールについて存在できます。データが作成されて、モジュールまたはプログラムに保管されると、このデータはプログラム識別情報になります。オペレーティング・システムは、このデータを、モジュール変更 (CHGMOD) やプログラム変更 (CHGPGM) コマンドの実行、プログラムのデバッグ、IPA の使用などの操作に使用できます。

プログラム識別情報は、モジュール変更 (CHGMOD)、プログラム変更 (CHGPGM)、およびサービス・プログラムの変更 (CHGSRVPGM) コマンドで、プログラム識別情報除去 (RMVOBS) パラメーターを指定することにより、除去できます。プログラム識別情報を除去すると、MI プログラムはデータにアクセスできなくなります。

注: プログラム識別情報を除去すると、取り消しで復元することができません。

ほとんどの種類のデータでは、プログラム識別情報を除去すると、そのデータはオブジェクトからも除去されます。対応する機能が必要なくなり、オブジェクトを小さくしたい場合は、データを除去できます。ただし、作成データ (*CRTDTA) はオブジェクトから除去されません。作成データは、プログラム識別情報ではない形式に変換されます。オペレーティング・システムではこのデータを使用できませんが、マシンでは、プログラム識別情報ではない作成データを使用して、オブジェクトを変換できます。

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

解決されたインポートおよび未解決インポート

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、インポートのタイプと名前がエクスポートのタイプと名前にまだ一致しないものです。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、実行時に未解決のインポートを使用しようとすると、エラー・メッセージ MCH4439 が発行されます。「解決されていないインポートを使用しようとしています (Attempt to use an import that was not resolved.)」というメッセージが表示されます。

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。例えば、次のようになります。

```
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: プログラムへのバインド対象をより効率的に制御するには、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。*LIBL とともに *ALL を指定すると、プログラムの実行時に予測不能な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。例えば、`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。例えば、`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。

重複記号

変数名およびプロシージャー名の記号は、ウイーク・エクスポートまたはストロング・エクスポートとしてモジュールからエクスポートすることができます。

ウイーク・エクスポートは、 95 ページの『インポートおよびエクスポートの概念』で説明されているようにさまざまなプログラミング言語によってサポートされています。モジュールが弱い記号を定義および参照すると、その記号のエクスポートとインポートは、共に弱く行われます。いくつかのモジュールは、同じ記号をウイーク・エクスポートとして定義することができます。記号の解決時には、すべての一致するインポートを満たすために 1 つのモジュールからのエクスポートが選択されるので、すべてのモジュールは同じ項目を参照します。

ストロング・エクスポートは、一般的に単一モジュールによって定義されます。複数のモジュールが記号をストロング・エクスポートとして定義すると、エラー・メッセージが出され、プログラム作成は失敗します。プログラム作成コマンド (CRTPGM、CRTSRVPGM、UPDPGM、UPDSRVPGM) では、OPTION(*DUPPROC *DUPVAR *WARN) を使用して、重複記号エクスポートの警告メッセージがあったとしてもコマンドを続行できるようにすることができます。こうしたエラー・メッセージは重要です。モジュールがそのエクスポートする記号を参照するときには、その参照はインポート参照ではなく、ローカル参照として行われます。2 つのモジュールが同じ記号をストロング・エクスポートとして定義し、各モジュールがその記号を参照する場合、それらは別個の項目を参照していることになります。こうしたメッセージを慎重に調べて、アプリケーションが正しく機能するようにしてください。ストロング・エクスポートをウイーク・エクスポートに変更すること、または、単一のモジュールのみが記号をエクスポートし、他のすべてのモジュールがその記号をインポートするようにコードを変更することが必要な場合があります。

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。 87 ページの図 29 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、88 ページの図 30 および 90 ページの図 31 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))
```

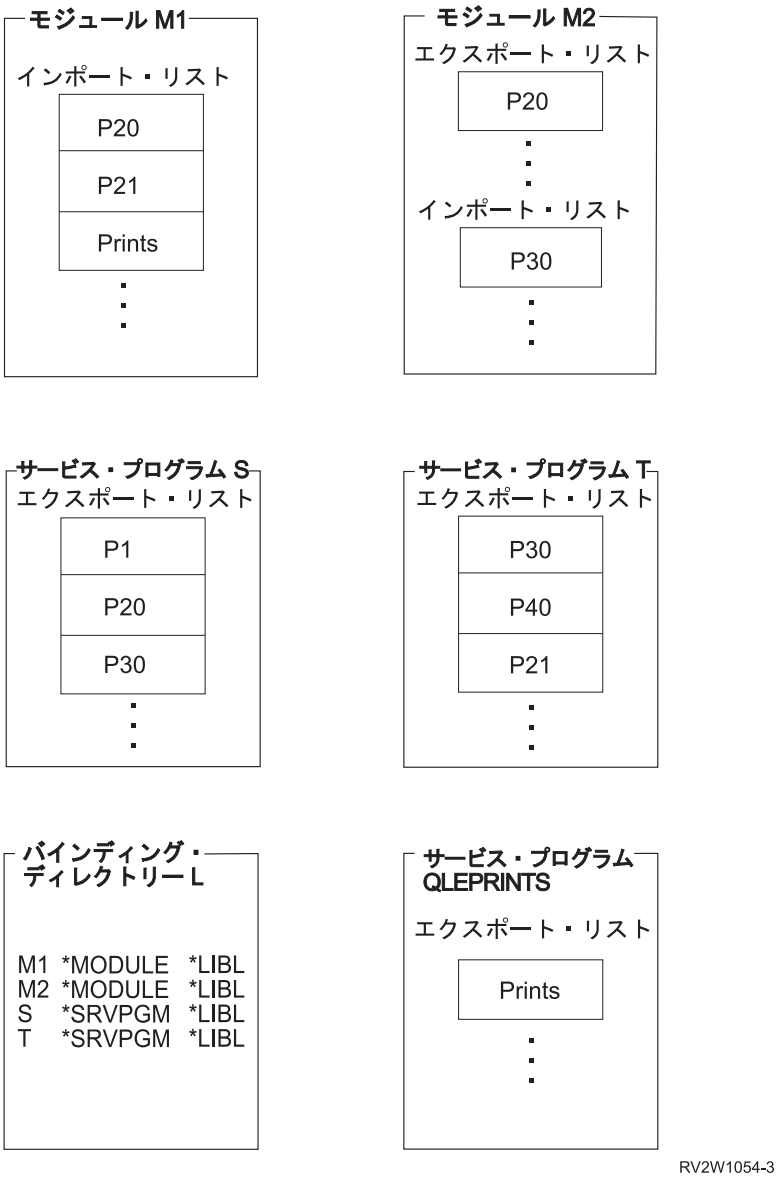

図 29. モジュール、サービス・プログラム、およびバインディング・ディレクトリー

プログラムの作成例 1

88 ページの図 30 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)
```

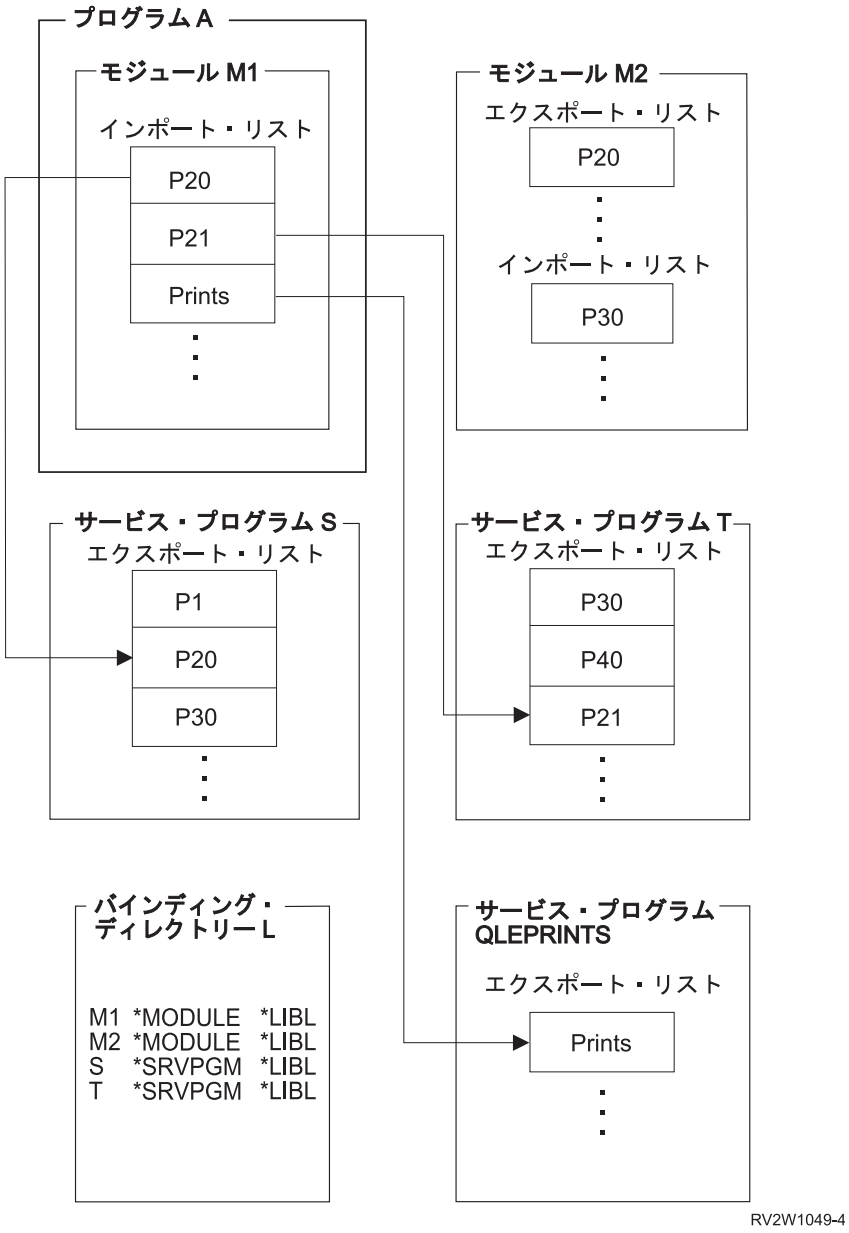

図 *30.* 記号の解決とプログラムの作成*:* 例 *1*

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 (88 ページ) のステップで既に処理されており、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の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。これにより、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の参照を探す必要があることを認識します。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、通常、バインドは失敗します。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

90 ページの図 31 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)
```

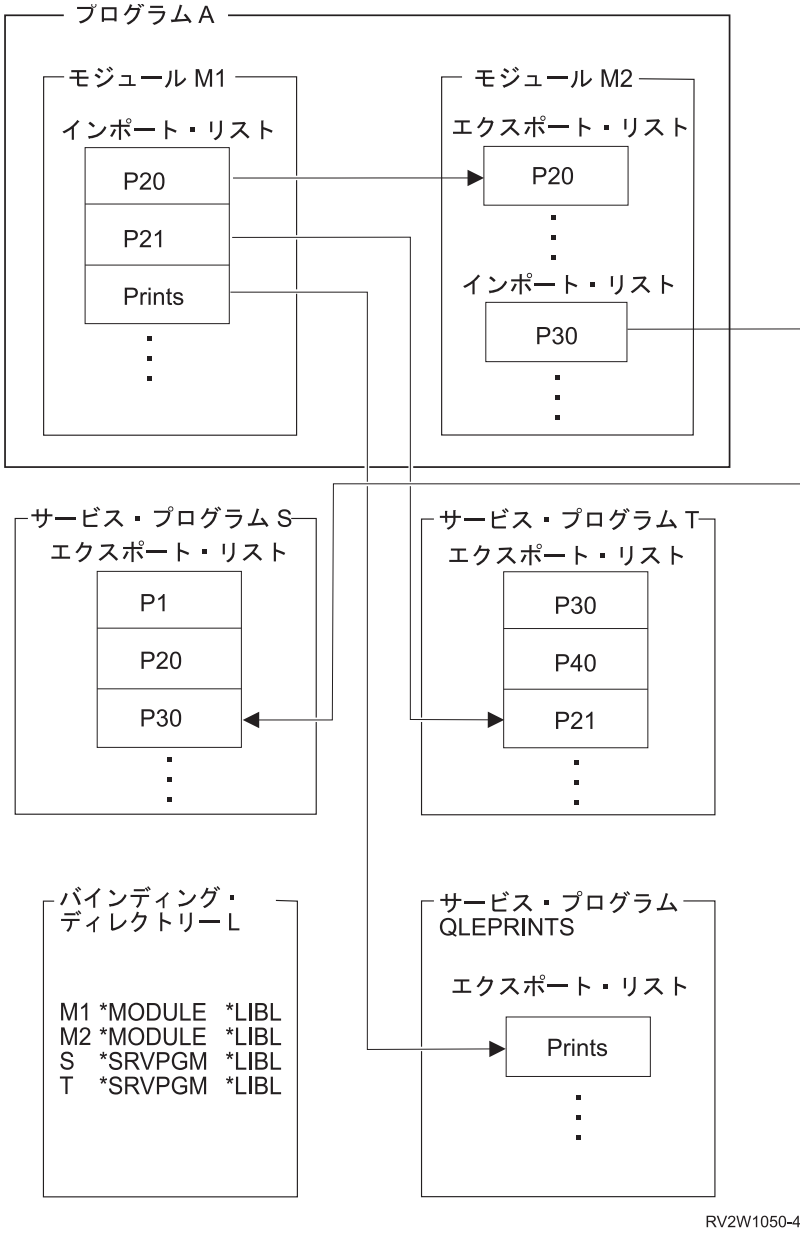

図 *31.* 記号の解決とプログラムの作成*:* 例 *2*

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。88 ページの図 30 と 図 31 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既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 95 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 (79 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 (SRCFILE)、ソース・メンバー (SRCMBR)、およびソース・ストリーム・ファイル (SRCSTMF) 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指定します。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 パラメーターまたは SRCSTMF 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 (97 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 コマンドまたは UPDSRVPGM コマンドの SRCFILE パラメーターまたは SRCSTMF 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。この値を指定すると、バインド・プログラム言語ソースを生成する必要がありません。しかし、
EXPORT(*ALL) が指定されたサービス・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に、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が変更される可能性があります。したがって、そのサービス・プログラムのシグニチャーが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 95 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。
- 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 パラメーターおよび SRCMBR パラメーター、または SRCSTMF パラメーターのいずれか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。

IBM i 7.3 より、ストリーム・ファイル (SRCSTMF パラメーター) からバインド・プログラム・ソースを処理できます。定義でのストリーム・ファイルがバイトのス

トリームであるためにストリーム・ファイルにレコード長がない場合でも、バインド・プログラム言語コンパイラーが処理できる行の最大長 (改行文字間の文字数) は 240 文字です。

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。 (92 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。

- ILE RPG プログラマーの手引き
- ILE COBOL プログラマーの手引き
- ILE C/C++ プログラマーの手引き

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. 記号のエクスポート (EXPORT) コマンド。これらの各コマンドは、 サービス・プログラムからエクスポートできる記号名を識別します。
3. プログラム・エクスポート終了 (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 32 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 *32.* 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できます。

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。IBM i 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、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、IBM i Information Center のセキュリティー・カテゴリーの情報を参照してください。

明示的シグニチャーの指定を選択する場合、バインド・プログラム言語ソースで必要となるのは 1 つのエクスポート・ブロックのみです。新規のエクスポートは、エクスポートのリストの最後に追加できます。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。新規のエクスポートをサービス・プログラムに追加するたびに、新規のエクスポート・ブロックをバインド・プログラム・ソースに追加する必要があります。

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。この規則は、明示的シグニチャーを指定する場合にも、バインド・プログラムに新規シグニチャーの生成を許可する場合にも適用されます。

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 コマンドを使用して確実に再作成される場合にのみ行うことができます。

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。特定のプロシージャーのインターフェースに

対して互換性のない変更を行うには、プロシージャーを呼び出すすべてのモジュールが再コンパイルされ、これらのモジュールを含むすべてのプログラムおよびサービス・プログラムが CRTPGM または CRTSRVPGM を使用して再作成される必要があります。

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 (例えば、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 (例えば、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。

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・プログラムは実行時にシグニチャーを検査します。システムは、値がサービス・プログラムのクライアントに認識されてい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、サービス・プログラムに再バインドせずに共通インターフェースを使用できます。

LVLCHK(*NO) 値は注意して使用してください。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説明については 224 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。

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 97 ページの図 32 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。

記号名は、ワイルドカード文字 (<<< または >>>) を使用してエクスポートすることもできます。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。例えば、『r』>>> および

『ra』>>> の 2 つのワイルドカードを指定すると、いずれも記号「rate」をエクスポートしますが、それらのワイルドカードが作成する 2 つのシグニチャーは異なります。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 215 ページの『第 17 章 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。

103 ページの図 33 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。

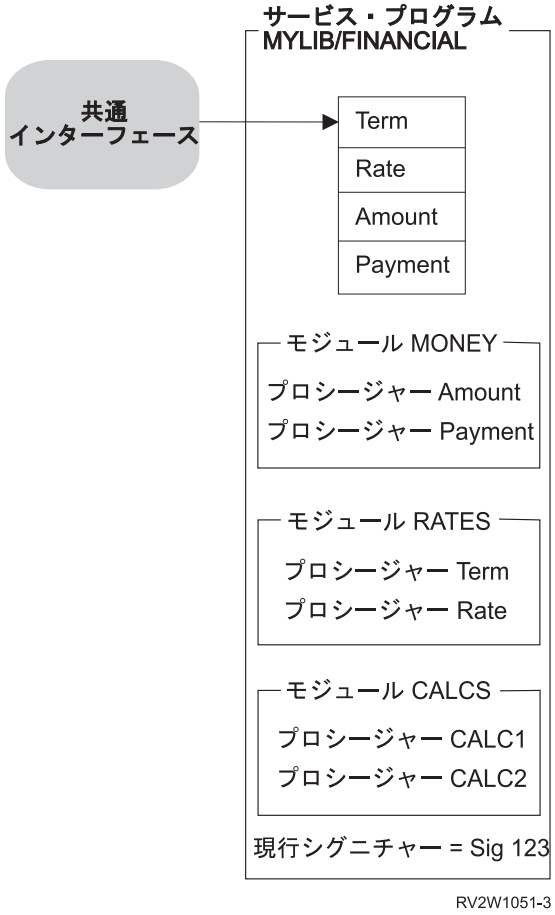

図 *33.* バインド・プログラム言語の使用によるサービス・プログラムの作成

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 104 ページの図 34 に示しています。

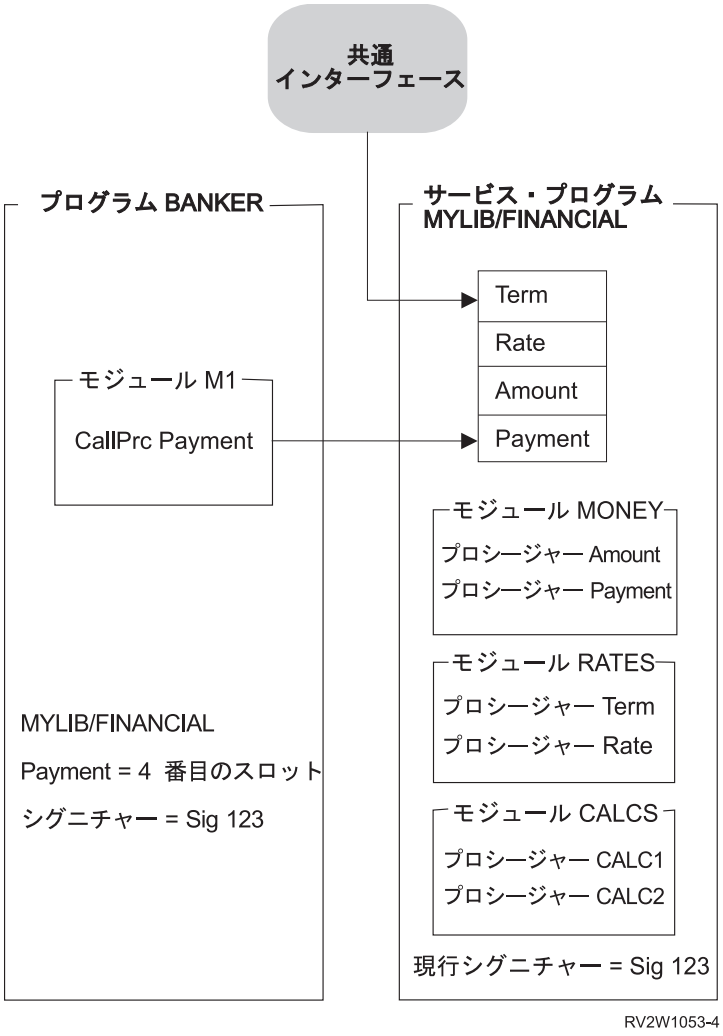

図 *34.* サービス・プログラム *FINANCIAL* の使用法

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では、BANKER の作成時に BANKER が使用した共通インターフェースが、実行時にまだ有効かどうかが確認されます。

図 34 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
```

以下の CRTSRVPGM コマンドでは、 107 ページの図 35 に示す拡張されたサービス・プログラムを作成するために、バインド・プログラム言語の例 3 に示した更新されたバインド・プログラム言語が使用されています。

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

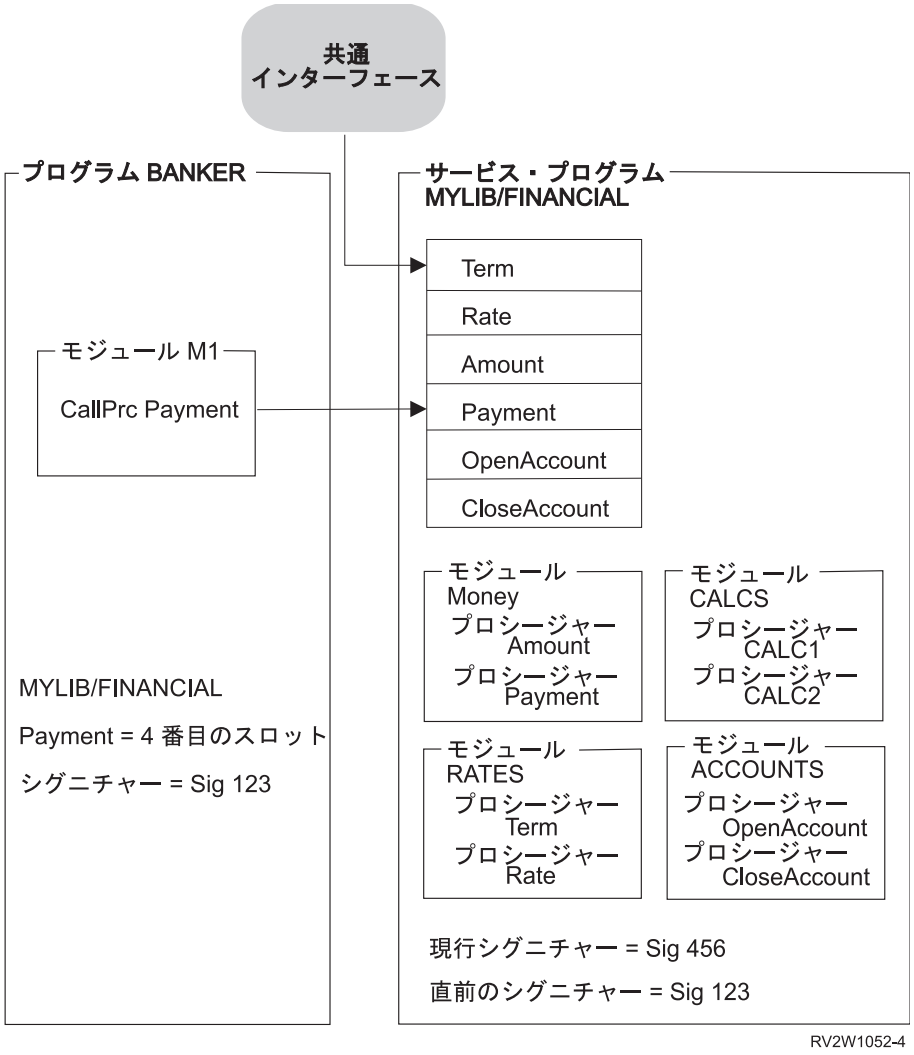

図 *35.* バインド・プログラム言語の使用によるサービス・プログラムの更新

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')
```

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。

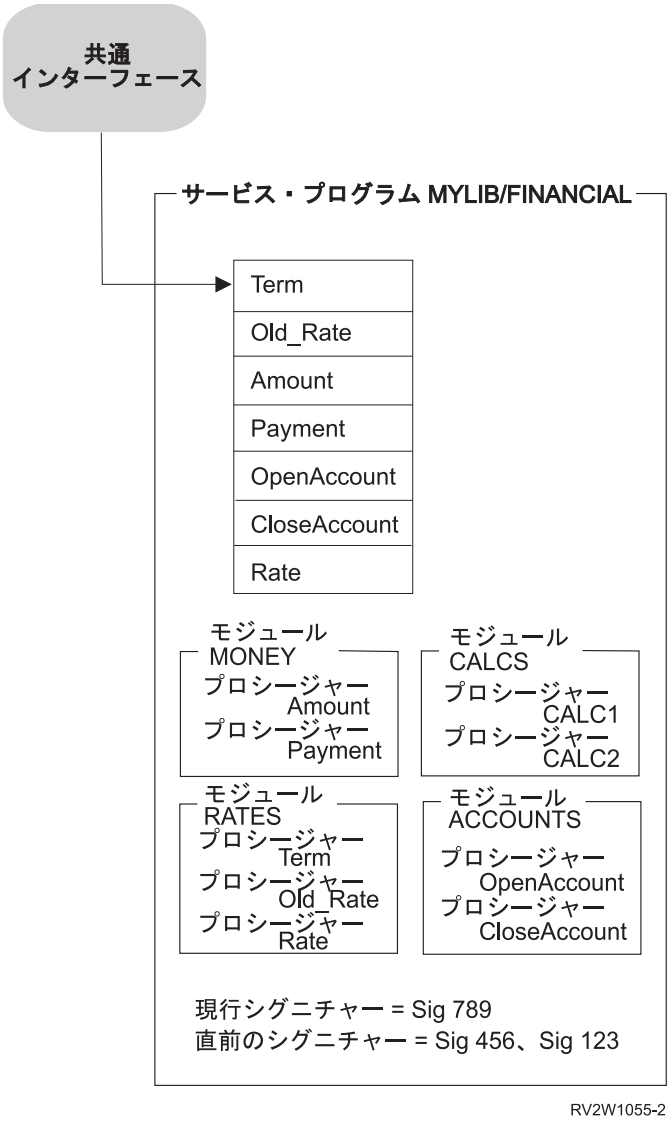

図 *36.* バインド・プログラム言語の使用によるサービス・プログラムの更新

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) およびサービス・プログラム変更 (CHGSRVPGM) コマンドは、プログラムおよびサービス・プログラムの属性を変更します。その場合、再コンパイルは必要ありません。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。

- 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。
- ストレージ・モデル (*SNGLVL から *INHERIT のみ)。

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)
- プロファイリング・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)
- ストレージ・モデル・プロンプト (STGMDL 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。

スレッド数 (NBRTHD) パラメーターを使用して、特にマルチプロセッサー・システムについて、使用可能な処理装置サイクルを利用できます。

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 37 を参照してください。

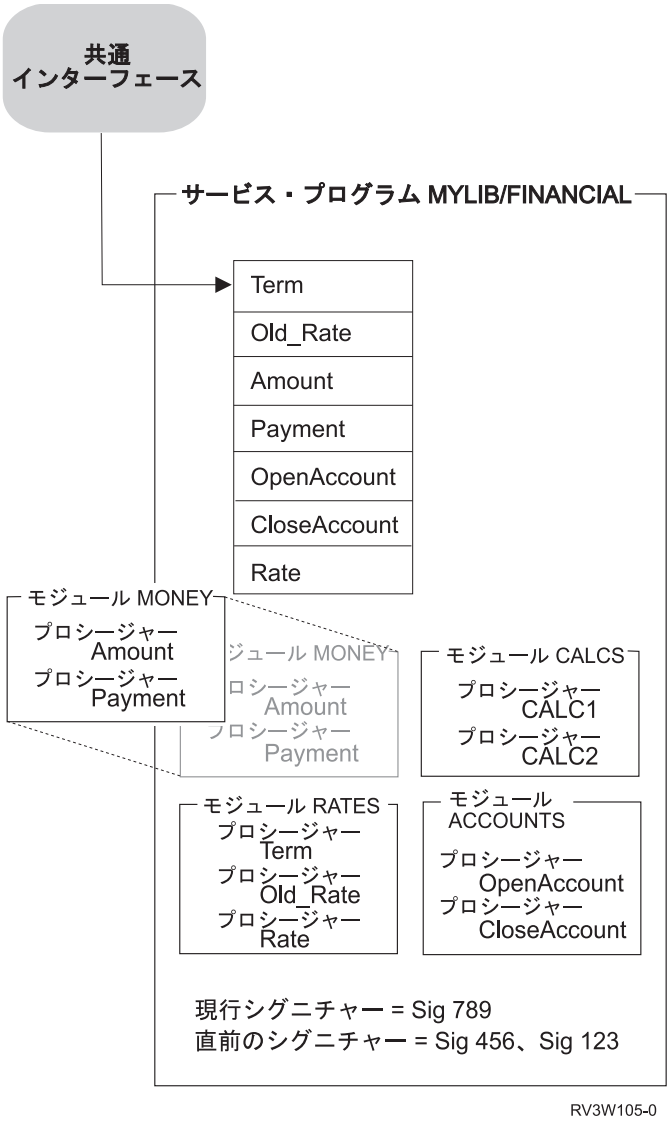

図 37. サービス・プログラムでのモジュールの置き換え

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) と ALWLIBUPD(*YES) は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、実稼働環境では、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。
- 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ることになるサービス・プログラムの数については、単純な答はありません。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。

第 8 章 活動化グループの管理

このトピックでは、活動化グループを使用したアプリケーションの構成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。

120 ページの図 38 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。

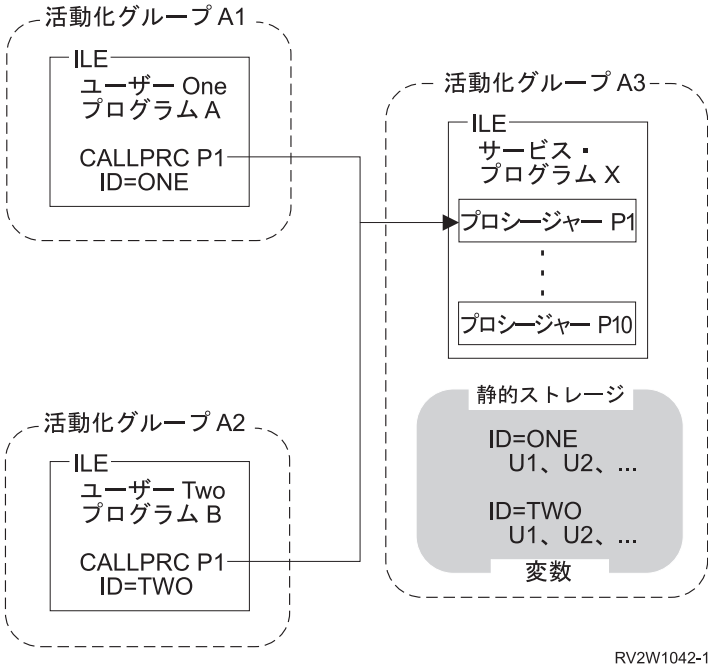

図 *38.* 同じジョブで実行される複数のアプリケーション

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できると同時に、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 39 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。

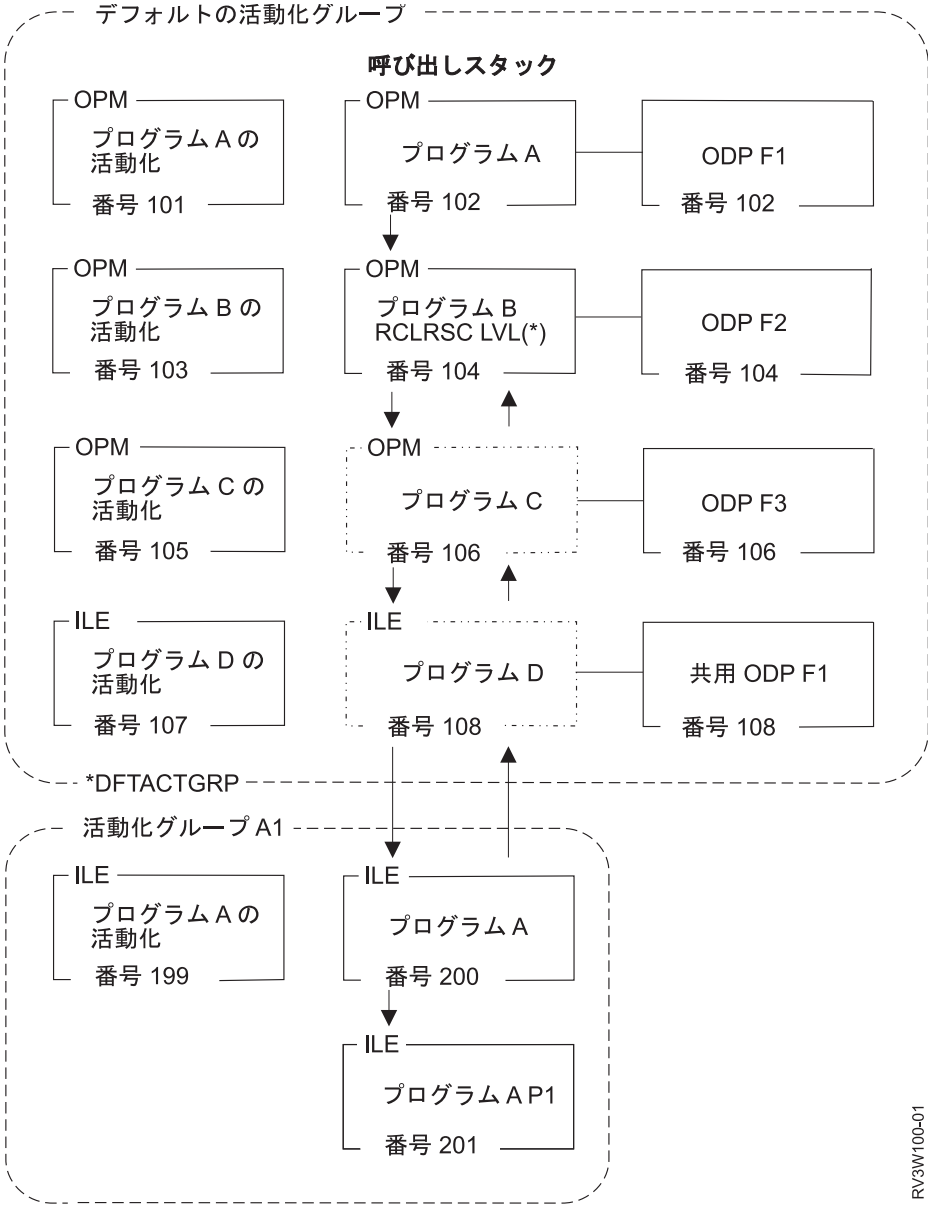

図 *39.* リソースの再利用

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 (例えば、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 CRTBNDRPG コマンドおよび CRTBNDCL 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 CRTBNDRPG コマンドまたは CRTBNDCL 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 を使用すると、 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。 121 ページの図 39 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、IBM i 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 (単一レベル・ストレージ・モデルを使用する場合)。

  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 9 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。このトピックでは、 29 ページの『プログラムまたはプロシージャーの呼び出し』で紹介する、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のさまざまな例を示します。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。

また、このトピックでは、新しい ILE 機能または OPM から ILE への変換を使用して、OPM アプリケーション・プログラミング・インターフェース (API) をサポートする方法についても説明します。

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 (例えば、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スレッドごとに 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。呼び出されたプロシージャーまたはプログラムが終了するとスタック項目が除去され、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。詳しくは、「IBM i におけるスレッド」を参照してください。

呼び出しスタックの例

126 ページの図 40 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 18 ページの『モジュール・オブジェクト』で定義されています。以下のステップで呼び出し処理が実行されます。

1. プログラム A に対して動的プログラム呼び出しを行う。
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。

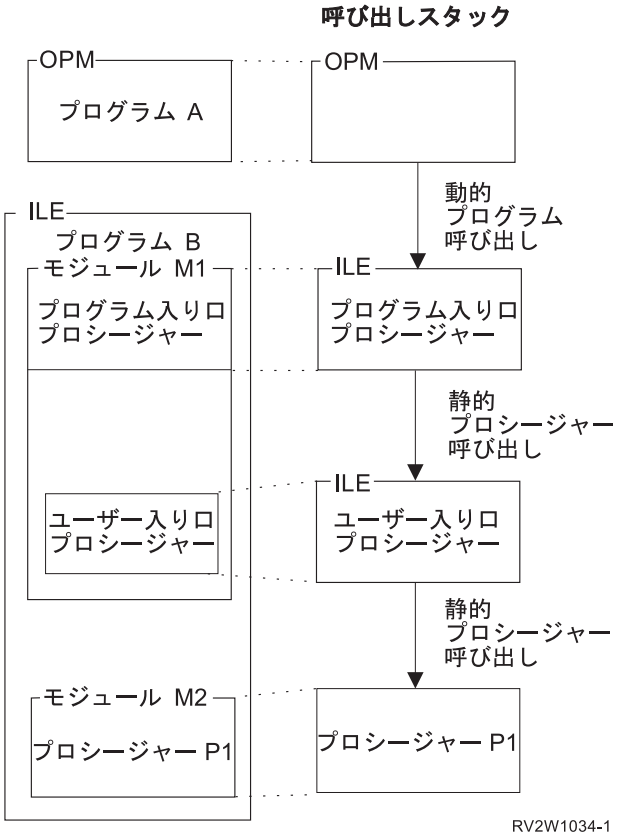

図 *40.* 呼び出しスタックにおける動的プログラム呼び出しと静的プロシージャー呼び出し

図 40 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。これは、現在処理中のプログラムまたはプロシージャーの項目です。現行プログラムまたはプロシージャー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、その呼び出しスタック項目がスタックから除去されます。

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。この場合、プロシージャー P1 は UEP に制御を戻し、P1 の項目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 項目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 項目はスタックから除去されます。プログラム A の項目だけが、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 の実行時には、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の 3 つのタイプの呼び出しを実行できます。

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。活動化されたプログラムがまだ存在しない場合にプログラムが動的に呼び出されると、プログラムの活動化が実行されます。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、活動化の据え置きが適用されないすべてのサービス・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。 この呼び出しの例としては、以下のようなものがあります。

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、 3 の場合、活動化グループ境界を超えて静的プロシージャー呼び出しが実行されることはありません。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、呼び出し側プロシージャーにバインドされています。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。対照的に、プロシージャー・ポインターのターゲットは、呼び出しごとに異なる場合があります。

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。例えば、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。

ILE プロシージャーへの引数の引き渡し

ILE プロシージャー呼び出しでは、引数は、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 41 は、これらの引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。

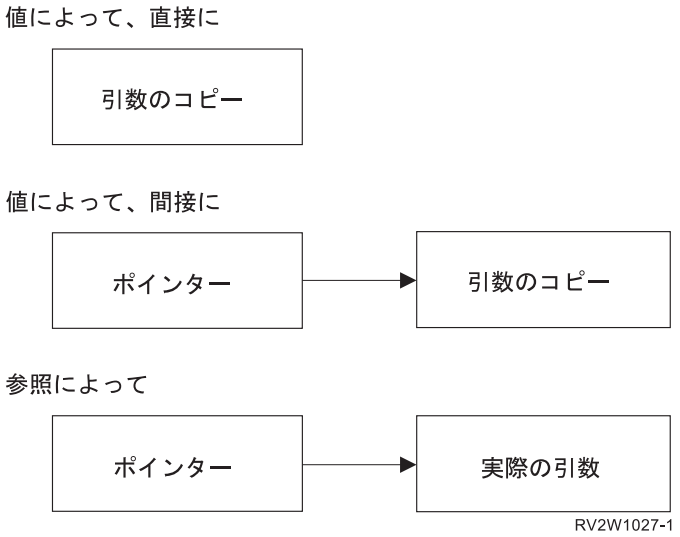

図 *41. ILE* プロシージャーへの引数の引き渡し方式

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。例えば、ILE C は引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。

IBM i 7.3 より前、静的プロシージャー呼び出しで指定できる引数は最大 400 個でした。 IBM i 7.3 以降では、静的プロシージャー呼び出しで最大 16,383 個の引数

を指定できます。各 ILE 言語は、引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、以下の引数引き渡し形式をサポートします。

- ILE C は、引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ます。呼び出される関数の #pragma 引数ディレクティブに適切な値を指定すると、引数を拡大しないで、または値によって間接的に受け渡すことができます。
- ILE C++ は、引数を値によって直接受け渡します。C++ は、デフォルトでパラメーターおよび浮動小数点値を拡大しません。呼び出される関数の宣言に関する外部リンケージ指定子に適切な値を指定すると、引数を拡大するか、または値によって間接的に受け渡すことができます。
- ILE COBOL は、値、参照、または間接的な値により、引数を受け渡します。値によって受け渡されるパラメーターは拡大されません。
- ILE RPG は、値または参照により、引数を受け渡します。 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数を受け渡しします。値によって受け渡されるパラメーターは拡大されません。

関数の結果

関数 (結果の引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 42 に示すように特殊な関数の結果の引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。

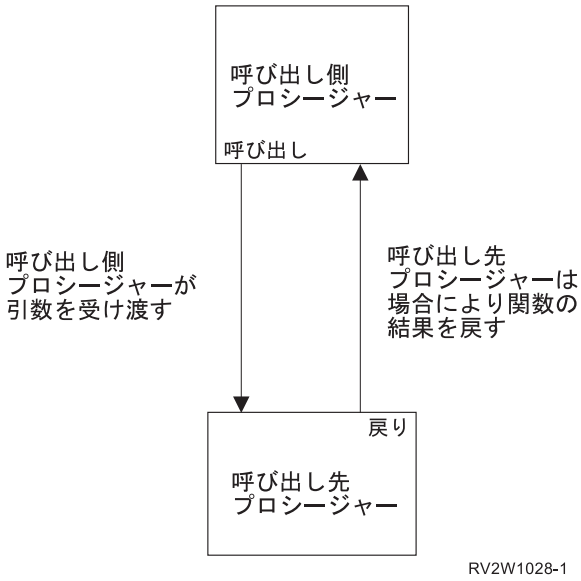

図 42. プログラム呼び出し引数の用語

省略された引数

すべての ILE 言語は、省略された引数をシミュレートできます。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できます。例えば、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数を想定している場合、引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数を省略できることがあります。省略された引数を特定の ILE 言語で指定する方法については、その言語の「プログラマーの手引き」を参照してください。IBM i Information Center のプログラミング・カテゴリーの中の API トピックには、各 API ごとに省略できる引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。例えば、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。

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)、参照によって引数を渡します。これは、呼び出されたプログラムがそれぞれの引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 個の引数を指定できます。

各 ILE 言語は、引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。例えば、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。例えば、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数を渡すための構文が用意されています。例えば、ILE C には、値の引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数の形式 (例えば、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数を正確に解釈することができます。

引数は値を提供します。操作記述子はその引数のサイズおよびタイプに関する情報を提供します。例えば、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。例えば、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

操作記述子を使用する必要があるのは、異なる ILE 言語で作成された呼び出し先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数に記述子を必要としています。IBM i Information Center のプログラミング・カテゴリーの中の API トピックのバインド可能 API に関する情報に、指定されたバインド可能 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。必要な記述子が指定されないと、プロシージャーが呼び出されても実行時に失敗します。

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**:** ただし、不必要な記述子を生成すると、パフォーマンスを損なう恐れがあります。記述子を使用すると、結果的に呼び出しパスの長さが増すためです。

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 (CEEGSI) 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・プログラム内のプロシージャーは OPM プログラムからは直接呼び出せないという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 43 は、この手法の例を示しています。

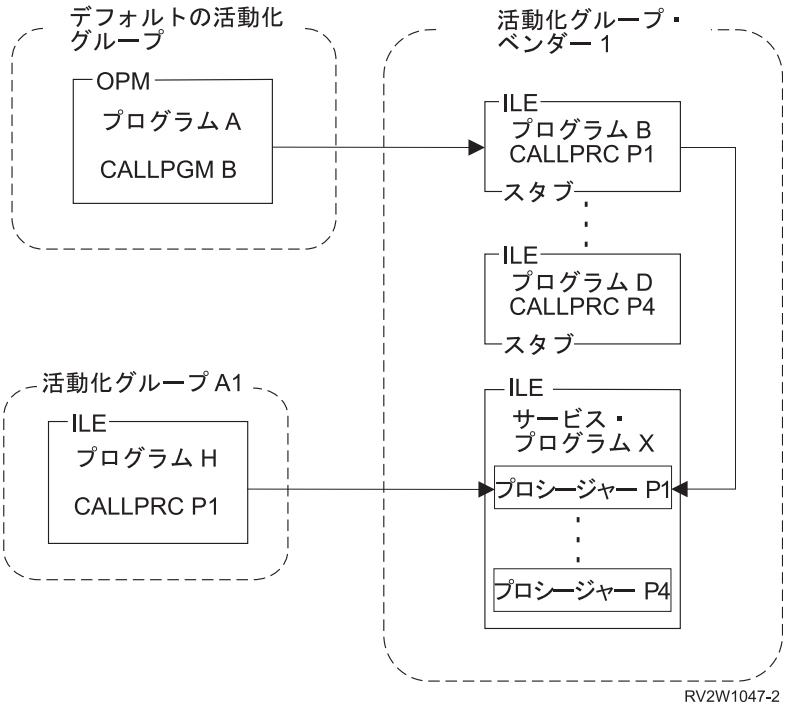

図 *43. OPM* および *ILE API* のサポート

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。

ご使用の API のインプリメンテーションを、ILE プロシージャー (この例ではプロシージャー P1) に指定してください。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呼び出すことも、プログラム・スタブの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。例外メッセージの送信など、呼び出しスタッ

ク構造に依存する処置を取る際は、注意してください。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、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 10 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や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。このトピックの残りの部分では、動的ストレージとその制御方法に焦点を合わせます。

単一レベル・ストレージ・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・ストレージ・ヒープを使用できます。ALCHSS 命令は常に単一レベル・ストレージを使用します。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・ストレージ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。

例えば、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。例えば、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 138 ページの『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。

他の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。 ストレージ管理 API については、IBM i Information Center のプログラミング・カテゴリーの中の API トピック・コレ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・ストレージ・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。すべての _CEE4ALC 割り振りストラテジー属性については、IBM i Information Center のプログラミング・カテゴリーの中の API トピック・コレクションを参照してください。

単一レベル・ストレージ・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。

ストレージ管理 API については、IBM i Information Center のプログラミング・カテゴリーの中の API トピック・コレ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のバインド可能 API を明示的に使用できます。

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 63 ページの『第 6 章 テラスペースおよび単一レベル・ストレージ』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、ヒープ・ストレージ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 モジュールは、単一レベル・ヒープ・ストレージまたはテラスペース・ヒープ・ストレージを使用することができます。テラスペース・ストレージ・モデルを使用するモジュールの場合、ヒープ・ストレージのデフォルト・タイプはテラスペースです。継承ストレージ・モデルまたは単一レベル・ストレージ・モデルを使用するモジュールの場合、ヒープ・ストレージのデフォルト・タイプは単一レベルです。ただし、制御仕様書で ALLOC キーワードを使用すると、ヒープ・ストレージのタイプを明示的に設定することができます。RPG サポートでは、単一レベルのヒープ・ストレージ操作に CEEGTST、CEECZST 、および CEEFRST のバインド可能 API が使用され、テラスペースのヒープ・ストレージ操作に `_C_TS_malloc()`、`_C_TS_realloc()`、および `_C_TS_free()` 関数が使用されます。

スレッド・ローカル・ストレージ

ILE C、ILE C++、および ILE RPG コンパイラーはすべて、スレッド・ローカル・ストレージ (TLS) をサポートします。各プログラムまたはサービス・プログラムの TLS 変数は、TLS フレームに編成されます。*TLS* フレーム には、プログラムまたはサービス・プログラムに関連した各 TLS 変数の、初期化されたコピーが含まれます。プログラムまたはサービス・プログラムを実行する各スレッドについて、TLS フレームのコピーが 1 つ作成されます。 特定のコンパイラーで使用可能なサポートについては、特定の高水準言語 (HLL) の資料を参照してください。

TLS 変数は、各スレッドごとに TLS 変数の固有のコピーが存在する点を除いて、静的変数に似ています。TLS 変数と静的変数の違いについて、次の表に示します。

表 *9. TLS* 変数と静的変数の違い

	静的変数	**TLS** 変数
変数のストレージが割り振られる時期	プログラムまたはサービス・プログラムの活動化が行われるとき。	変数を含む TLS フレームにスレッドが最初にタッチするとき。
変数が初期化される時期	プログラムまたはサービス・プログラムの活動化が行われるとき。[2]	変数を含む TLS フレームにスレッドが最初にタッチするとき。
変数のストレージが解放される時期	プログラムまたはサービス・プログラムの非活動化が行われるとき。	スレッドが破棄されるとき。
各スレッドにそれぞれ変数のコピーがあるか。	ない。全スレッドで単一コピーを共有します。	ある。
変数は、単一レベル・ストレージとテラスペース・ストレージのどちらに保管されるか。	プログラムまたはサービス・プログラムの活動化グループによって異なる。[1]	TLS 変数は常にテラスペース・ストレージに保管される。[1]

[1] 詳細については、 63 ページの『第 6 章 テラスペースおよび単一レベル・ストレージ』を参照してください。

[2] これは、変数がシステムによって直接初期化される時期を表します。変数は、あとで HLL によって間接的に初期化される場合があります。

スレッド内の TLS 変数への参照が行われるとき、その参照は、そのスレッドに関連する変数のコピーにアクセスします。他のスレッドに関連する変数のコピーにアクセスしたり、更新することはありません。

各 TLS 変数は 1 つのスレッドと関連しているので、通常は同期 (207 ページの『第 16 章 共用ストレージの同期』を参照) を気にする必要はありません。ただし、TLS 変数のアドレスが別のスレッドに渡される場合は、同期が必要になることがあります。

第 11 章 例外および条件管理

このトピックでは、例外処理と条件処理の詳細について記載します。このトピックを読む前に、 50 ページの『エラー処理』で説明する拡張概念を読んでおいてください。

オペレーティング・システム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。例えば、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。これらの相互作用については、このトピックで説明します。例外ハンドラー という用語は、オペレーティング・システムの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。

図 44 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。

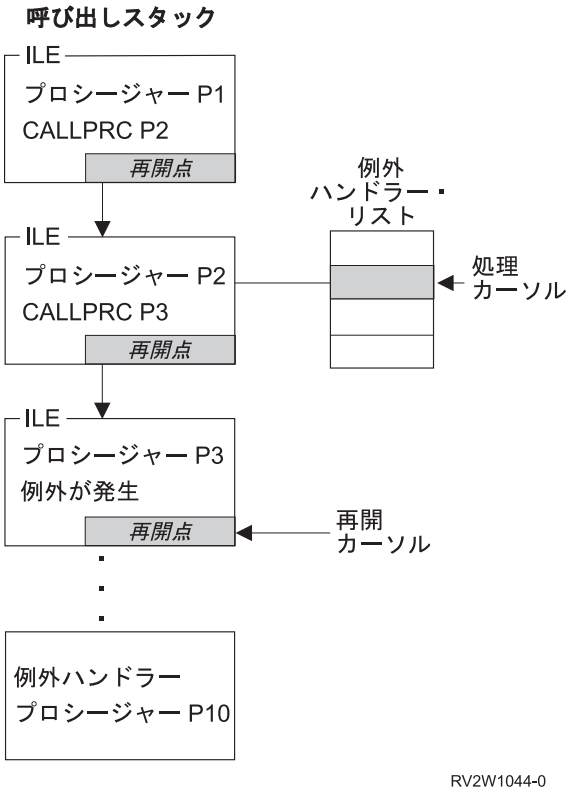

図 *44.* 処理カーソルおよび再開カーソルの例

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 44 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。例えば、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。ユーザー作成条件ハンドラー登録 (CEEHDLR) バインド可能 API の実際の戻りコード値については、IBM i Information Center のプログラミング・カテゴリーの中の API トピック・コレ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。ユーザー作成条件ハンドラー登録 (CEEHDLR) バインド可能 API の実際の戻りコード値については、IBM i Information Center のプログラミング・カテゴリーの中の API トピック・コレ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、例外処理の続行に使用される処理カーソルの位置をある程度制御できます。IBM i Information Center のプログラミング・カテゴリーの中の API トピック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。ユーザー作成条件ハンドラー登録 (CEEHDLR) バインド可能 API の実際の戻りコード値については、IBM i Information Center のプログラミング・カテゴリーの中の API トピック・コレ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 10 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 *10.* 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、システムから独立した方法で表されるオペレーティング・システムの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に使用できる 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。例えば、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 45 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。

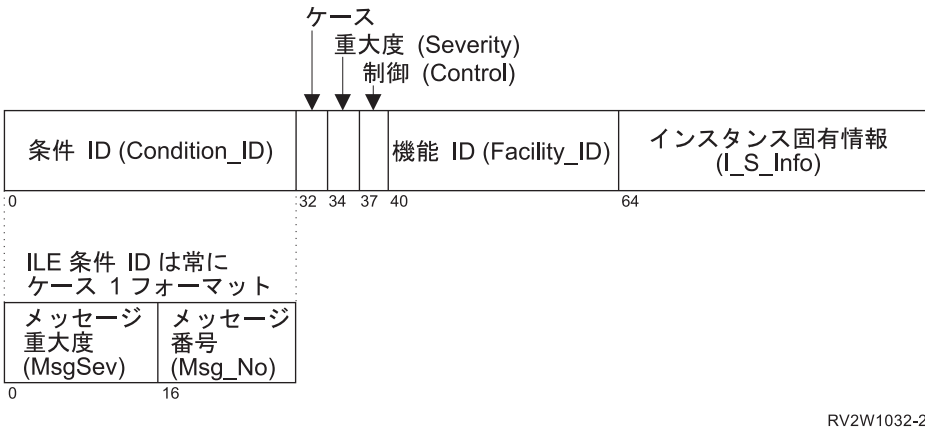

図 *45. ILE* 条件トークンのレイアウト

各条件トークンには 図 45 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 147 ページの表 10 を参照してください。対応するオペレーティ

ング・システムのメッセージ重大度については、 151 ページの表 12および 151 ページの表 13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、メッセージを生成したのはシステムなのか、または HLL 実行時なのかを示します。表 11 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 147 ページの表 10 を参照してください。対応するオペレーティング・システムのメッセージ重大度については、 151 ページの表 12および 151 ページの表 13 を参照してください。

メッセージ番号 **(MsgNo)**
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。MsgNo は、Facility_ID と組み合わせることで、条件を一意的に識別します。

表 11 は、ILE 条件トークンおよび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 *11.* 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	IBM i メッセージ
MCH	IBM i 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 条件とオペレーティング・システム・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。

実行時メッセージの形式は、すべて **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 150 ページの表 11 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 12 および表 13 は、ILE 条件重大度をメッセージ重大度にマップする方法を示しています。

表 *12. ILE* 条件の重大度への **ESCAPE* メッセージの重大度のマッピング

IBM i のメッセージの重大度から	**ILE** 条件の重大度へ	**IBM i** メッセージの重大度へ
0 から 29	2	20
30 から 39	3	30
40 から 99	4	40

表 *13. ILE* 条件の重大度への **STATUS* と **NOTIFY* メッセージの重大度のマッピング

IBM i のメッセージの重大度から	**ILE** 条件の重大度へ	**IBM i** メッセージの重大度へ
0	0	0
1 から 99	1	10

IBM i Messages およびバインド可能 API 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 12 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。

このトピックでは、ソース・デバッガーに関するいくつかの考慮事項について説明します。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 (例えば、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。
  - STRDBG コマンドの OPMSRC (OPM ソース・レベルのデバッグ) 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

最適化レベルと、バインドされるモジュールのデバッグ・データのプログラム識別情報は、プログラムをデバッグする能力に影響します。

最適化レベル

高いレベルの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・データを既に持っている、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。例えば、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。これについては、CL プログラミングの資料のテストに関するトピック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、IBM i Information Center のプログラミング・カテゴリーの中の API トピック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。 例えば、

```
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 (例えば、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 13 章 データ管理機能の有効範囲指定

このトピックでは、ILE プログラムまたはサービス・プログラムで使用可能なデータ管理リソースについての情報を記載します。このトピックを読む前に、 57 ページの『データ管理機能の有効範囲指定の規則』で説明するデータ管理機能の有効範囲指定の概念を理解しておく必要があります。

各リソース・タイプの詳細については、ILE HLL のそれぞれの「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・レベルです。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・レベルとジョブ・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) コマンドで有効範囲制御 (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、IBM i

Information Center のバックアップおよびリカバリーのトピック、または iSeries バックアップおよび回復の手引き SD88-5008 を参照してください。

ローカル **SQL** カーソル
SQL カーソルは、ユーザー作成アプリケーション内か、CREATE PROCEDURE、CREATE FUNCTION、または CREATE TRIGGER LANGUAGE SQL ステートメントの代わりに作成されるデータベース構成 ILE C プログラムまたは ILE C サービス・プログラム内で宣言および使用することができます。

あらゆるタイプの SQL アプリケーションには、SQL カーソル有効範囲制御機能があります。カーソル有効範囲制御機能には、活動化グループにカーソルの有効範囲を設定するオプションのほか、他のいくつかの有効範囲選択項目が含まれます。

詳しくは、「IBM i DB2® for i SQL Reference」を参照してください。

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、IBM i Information Center のプログラミング・カテゴリーの中の API のトピック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 をデフォルト以外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、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、IBM i Information Center のプログラミング・カテゴリーの中の API のトピック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)** 通信の会話
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。共通プログラミング・インターフェース (CPI) 通信の会話の詳細については、IBM i Information Center のプログラミング・カテゴリーの中の API トピック・コレ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、IBM i Information Center のプログラミング・カテゴリーの中の API のトピック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 におけるコミットメント制御では、従来の制御と比較して次の 2 点が変更されています。

- 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は、 データベース・ファイルまたはデータベース表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および処理することを可能にします。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。こうしたリソースについては、同じジョブに対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、リソースのコミットまたはロールバックを実行することができます。

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、単一レベル・ストレージの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す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。ただし、デフォルトの活動化グループは 1 つのコミットメント定義を共用します。有効範囲が活動化グループのコミットメント定義は、活動化グループ・レベルのコミットメント定義と呼ばれます。デフォルトの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) のコミットメント定義と呼ばれます。ジョブのさまざまな活動化グループ内で実行されるプログラムは、多数の活動化グループ・レベルのコミットメント定義を開始および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。活動化グループ内で実行されるプログラムのうち、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持たないものは、ジョブ・レベルの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。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・レベルのいずれかの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が終了すると、その活動化グループ内で実行されているプログラムは、コミットメント制御下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、または既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非コミットの変更が存在する
- 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、コミット定義が終了する前に、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のクローズ操作は、活動化グループ・レベルのコミットメント定義に対する暗黙のコミット操作の前に実行されます。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る必要があります。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。

以下の条件が存在すると、通知オブジェクトが更新されます。

- 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

第 14 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には、特定の高水準言語が提供する機能以外の追加機能が用意されている場合があります。例えば、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。例えば、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フローの管理
- 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、IBM i Information Center のプログラミング・カテゴリーの中の API トピック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取得 (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。

DSM バインド可能 API については、IBM i Information Center のプログラミング・カテゴリーの中の API トピック・コレクションを参照してください。

第 15 章 拡張最適化技法

このトピックでは、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。

- 198 ページの『適応コード生成』
- 187 ページの『引数の拡張最適化』
- 180 ページの『プロシージャー間分析 (IPA)』
- 190 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

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 178 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: プログラムが既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 176 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。両方のコマンドとも、*EXCLUDE という PUBLIC 権限を伴って出荷されます。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 176 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、増加するたびに DASD に書き込むとプログラムの実行時のパフォーマンスが大幅に低下するため、増加するたびには書き込まれません。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 172 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。既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。

既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。

例えば、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。

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。例えば、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。例えば、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。

プログラムがプロファイル作成データを収集することが必要でなくなった場合は、以下のいずれかのアクションを実行できます。

- 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。

いずれかのアクションによって、プログラムはプロファイル作成データの収集が可能にされる前の状態に戻ります。 CHGMOD コマンドまたはモジュールを再コンパイルして、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順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、既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 186 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) 引数の値引数への変換。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) 引数を値引数へ変換します。
- 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 を使用して最適化されたコードの実行時間は、コンパイル時のみ最適化され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 (例えば、`/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 (例えば、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-threshold=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 (例えば、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。 182 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。例えば、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 は、C++ モジュールの作成 (CRTCPPMOD) コマンドまたはバインド C++ プログラムの作成 (CRTBNDCPP) コマンドのいずれかに RTBND(*LLP64) オプションを指定してコンパイルされたモジュールについては、正しく最適化できない場合があります。モジュールに仮想関数が使用されていない場合、IPA はそのモジュールを最適化できます。仮想関数が使用されている場合は、MODCRTOPT(*NOKEEPILDTA) オプションを指定する必要があります。
- IPA は、10 進浮動小数点のデータまたは変数が含まれているモジュールについては、正しく最適化できない場合があります。
- IPA は、スレッド・ローカル・ストレージ変数が含まれているモジュールについては、正しく最適化できません。

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、 222 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 (例えば、QD0068xxxx。この xxxx の範囲は 0000 から 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。

引数の拡張最適化

引数の拡張最適化は、頻繁に実行されるプロシージャー呼び出しを含んだプログラム (主に非仮想メソッド呼び出しを行う C++ アプリケーションなど) のパフォーマンスを向上させるために使用されるモジュール間の最適化です。実行時パフォーマンスの向上は、変換プログラムおよびバインド・プログラムが、プログラムまたは

サービス・プログラム内で呼び出されるプロシージャー間でのパラメーターの受け渡しや結果のリターンに、最も効率的なメカニズムを使用できるようにすることで実現されます。

引数の拡張最適化を使用する方法

引数の最適化 (ARGOPT) パラメーターは、引数の拡張最適化をサポートするために、CRTPGM コマンドおよび CRTSRVPGM コマンドで使用することができます。指定可能な値は、*YES および *NO です。 ARGOPT(*YES) を指定すると、プログラムまたはサービス・プログラムが、引数の拡張最適化を使用して作成されます。デフォルトは *NO です。

引数の拡張最適化を使用する際の考慮事項および制約事項

プログラムの作成時に ARGOPT(*YES) を指定すると、引数の拡張最適化が適用されます。通常はこれにより、プログラム内のほとんどのプロシージャー呼び出しのパフォーマンスが向上します。しかし、引数の拡張最適化の使用を決定する前に、以下の項目について考慮しておく必要があります。

- プラグマ・ベースの引数の最適化との相互関係

  ARGOPT(*YES) によって有効にした引数の最適化と、C および C++ のコンパイラーでサポートされている #pragma argopt ディレクティブによって有効にした引数の最適化は、重複した部分を持つ相補的なものです。

  コード内に既に #pragma argopt がある場合、それはそのままにして、ARGOPT(*YES) を併用します。重複する #pragma argopt は、後で除去しても、そのままにしても構いません。

  コード内に #pragma argopt がない場合は、ARGOPT(*YES) を使用すると、多くの場合役に立ちます。引数の拡張最適化は関数ポインターによる呼び出しを最適化しないため、関数ポインターによってプロシージャーを呼び出す場合には、#pragma argopt を使用する必要があるかもしれません。関数ポインター呼び出しの例としては、C++ における仮想関数呼び出しがあります。

  #pragma argopt ディレクティブについて詳しくは、「 ILE C/C++ コンパイラー参照 」を参照してください。

  しかし、#pragma ディレクティブのソース・コードへの手動挿入が必要になるプラグマ・ベースの引数の最適化と異なり、引数の拡張最適化は、ソース・コードを変更する必要もなく、自動的に適用されます。また、引数の拡張最適化はどの言語で作成されたプログラムにも適用可能であるのに対し、プラグマ・ベースの方法は C および C++ 専用です。

  #pragma argopt ディレクティブは関数ポインターに適用可能ですが、引数の拡張最適化は仮想関数呼び出しおよび関数ポインターを使用した呼び出しを自動的には最適化しません。したがって、間接呼び出しを最適化する場合、argopt プラグマは、このように引数の拡張最適化と相補的な方法で使用すると便利です。
- 16 バイト・ポインター

16 バイトのスペース・ポインター・パラメーターが最も役に立つのは、引数の最適化においてです。スペース・ポインターは、文字、数値、クラス、およびデータ構造体などのデータ・オブジェクト・タイプを指します。 C および C++ におけるスペース・ポインターの例としては、char* や int* があります。しかし、システム・オブジェクトを指すポインターなど、IBM i に固有の他のタイプの 16 バイト・ポインターによって宣言されたパラメーターは、引数の最適化によって最適化されません。不完全型を基にした C および C++ の void* ポインターなど、オープン・ポインター・パラメーターも最適化されません。

- DTAMDL(*LLP64)

  DTAMDL(*LLP64) を使用して作成されたモジュールから構成される C および C++ のアプリケーションは、デフォルトの DTAMDL(*P128) を使用して作成されたものよりも、引数の最適化の利点が少なくなります。前者の場合、データへのポインターは 8 バイト長で、これらは常に最も効率的なメカニズムを使用してプロシージャー間で受け渡されます。後者の場合、データへのポインターは 16 バイト長で、これらは引数の最適化の最有力候補です。
- ターゲットのリリース

  ARGOPT(*YES) を使用して作成されるプログラムは、V6R1M0 以降のターゲット・リリースを使用して作成する必要もあります。

  引数の拡張最適化を最大限活用するには、ARGOPT(*YES) を使用して作成されたプログラムにバインドされるモジュールを、V6R1M0 以降のターゲット・リリースを使用して作成しなければなりません。これは、引数の拡張最適化は、V6R1M0 より前に作成されたモジュールに定義された関数との間で行われる呼び出しを無視するためです。
- プログラム作成時間の増加

  プログラム作成時に ARGOPT(*YES) を指定すると、プログラム内のすべてのモジュールに対して追加の分析が実行されます。数百または数千のモジュールから構成されるプログラムの場合、作成時間が大幅に増加する可能性があります。

  同様に、ARGOPT(*YES) を使用して作成したプログラムを、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) コマンドを使用して更新する場合に、更新が完了するまでの時間が増加する可能性があります。これは、すべてのモジュール間呼び出しが更新されるようにするための追加の分析が必要になることがあるためです。プロシージャー・インターフェースに変更がない場合、通常はこの追加の時間が短くなります。
- 特殊な呼び出し規約との相互関係

  引数の拡張最適化は、動的プログラム呼び出しには適用されません。また、_System キーワードを使用して定義された C および C++ の関数は、引数の拡張最適化の候補にはなりません。
- プログラム・プロファイル作成との相互関係

  引数の拡張最適化とプログラム・プロファイル作成は、同時に使用することができます。
- プロシージャー間分析 (IPA) との相互関係

IPA によって実行されるモジュール間分析および最適化は、引数の拡張最適化と重複します。したがって、IPA を使用する場合、引数の拡張最適化を使用する必要はありません。

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・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制御するために、ライセンス内部コード (LIC) に渡されます。これらのオプションは、モジュール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。デバッグを補助するオプションもあります。このセクションでは、ライセンス内部コードのオプションについて説明します。

現在定義されているオプション

現在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

最適化変換プログラムが、最適化レベル 40 でコンパイルする際に、実行時呼び出しスタックで保持されるスタック・フレーム数の削減を試みる、「call foliation (呼び出しの薄層化)」と呼ばれる最適化をより積極的に試行するように指示します。この利点は、必要なスタック・フレームが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て、まれに、実行時にスタック・オーバーフローが発生する可能性を低減できるということです。欠点は、プログラム障害の場合に、デバッグ時に呼び出しスタック内に残る手掛かりが減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]CallTracingAtHighOpt

このオプションを使用すると、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できます。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。 6.1 より前のリリースではこのオプションがデフォルトでオフになっており、最適化レベルが 40 の場合に、呼び出しと戻りのトラップはプロシージャーに挿入されません。 6.1 以降ではこのオプションが無視され、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に呼び出しと戻りのトラップが無条件に挿入されます。

[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。

CodeGenTarget=

CodeGenTarget オプションは、プログラムまたはモジュール・オブジェクトの作成ターゲット・モデルを指定します。作成ターゲット・モデルとは、そのオブジェクト用に生成されたコードが使用することのできるハードウェア機能を示します。この LICOPT に指定可能な値については、以下の表を参照してください。

値	意味
CodeGenTarget=Current	最適化変換プログラムは、現行マシン上で使用可能なすべての機能を使用できます。
CodeGenTarget=Common	最適化変換プログラムは、ターゲット・リリースがサポートしているすべてのシステムで使用可能なすべての機能を使用することができます。
CodeGenTarget=Legacy	最適化変換プログラムは、PowerPC® AS アーキテクチャーの POWER6 レベル以降でのみ使用可能な機能をどれも使用できません。
CodeGenTarget=Power6	最適化変換プログラムは、PowerPC AS アーキテクチャーの POWER6 レベルで使用可能なすべての機能を使用できます。
CodeGenTarget=Power7	最適化変換プログラムは、PowerPC AS アーキテクチャーの POWER7® レベルで使用可能なすべての機能を使用できます。
CodeGenTarget=Power8	最適化変換プログラムは、PowerPC AS アーキテクチャーの POWER8® レベルで使用可能なすべての機能を使用できます。
CodeGenTarget=Power9	最適化変換プログラムは、PowerPC AS アーキテクチャーの POWER9™ レベルで使用可能なすべての機能を使用できます。

このオプションについて詳しくは、 202 ページの『CodeGenTarget LICOPT』を参照してください。

[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

6.1 以降で稼働中のシステムでは、このオプションは無視されます。16 バイト・ポインターにテラスペース・アドレスおよび NULL ポインター値が含まれていない場合、 16 バイト・ポインターから 8 バイト・ポインターへの変換が行われるたびに、MCH0609 例外がシグナル通知されます。

[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
16 バイト・ポインターのアドレス算術演算の一部として、有効アドレス・オーバーフロー (EAO) 検査が実行されます。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・ストレージ (SLS) アドレスの両方を処理する必要があるため、コードが POWER6 より前のプロセッサー用に生成されていた場合、有効なテラスペースの使用によって、偽の EAO が発生する可能性があります。詳しくは、 198 ページの『適応コード生成』を参照してください。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。通常のケースよりも若干速度の遅い別のアドレス算術命令シーケンスが生成されますが、ほとんどの EAO が発生しなくなります。この LICOPT の使用が推奨される例としては、モジュール内で実行されるほとんどのアドレス演算で、テラスペース・アドレスの計算結果が低い値の 16 MB 領域になる場合です。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、入力ベースの別名の積極的な微調整を可能にします。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、実行時

パフォーマンスが向上する場合があります。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

[No]ReassocForIndexedLdSt

このオプションは、ロード命令または保管命令を出すアドレス指定式を、指標付けされたロード命令または保管命令への変換の要件を満たすように関連付けし直すよう、最適化変換プログラムに指示します。ほとんどの場合、これは、変位がゼロであった場合に、指標付けされたロード命令または保管命令に変形される可能性のある、ゼロ以外の変位を持つロード命令または保管命令を検索することを意味します。この場合、関連付けし直すことによって、アドレス式への変位の明示的な追加が行われ、ロード命令または保管命令の変位がゼロになります。

このライセンス内部コードのデフォルトは、ReassocForIndexedLdSt です。

[No]TailCallOptimizations

このオプションは、最適化変換プログラムが最適化レベル 40 でコンパイルする際に、実行時呼び出しスタックで管理されているスタック・フレームの数を減らすために、テール呼び出し最適化を実行するように指示します。この利点は、必要なスタック・フレームが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て、まれに、実行時にスタック・オーバーフローが発生する可能性を低減できるということです。欠点は、プログラムに障害が発生した場合、呼び出しスタック内にデバッグに使用できる手掛かりがあまり残らないということです。この LICOPT は有効のままにしておくことをお勧めします。

テール呼び出し とは、リターンの直前に実行されるプロシージャー内の呼び出しのことです。これらの場合、最適化変換プログラムは、呼び出し元がスタック・フレームを割り振る必要性をなくし、スタックが無変更のままになるようにする、テール呼び出し最適化を実行しようとします。通常、スタック・フレームは、呼び出し先がリターン・アドレスを保存および復元できるように、呼び出し元によって作成されます。これらの最適化を実行すると、テール呼び出しが、新規のリターン・アドレスを計算しない単純な分岐操作に変更されます。元のリターン・アドレスはそのままにされるため、再び呼び出し元のリターン・ロケーションを指すことになります。最適化により、呼び出し元と呼び出し先の間の中間スタック・フレームがなくなるため、呼び出し先は呼び出し元の呼び出し元に戻ります。

例えば、関数 A が関数 B を呼び出し、関数 B がテール呼び出し最適化の有効になった関数 C に対してテール呼び出しを実行した場合、関数 B はスタック・フレームを割り振るのではなく、関数 C に分岐します。関数 C は、完了すると、関数 B への呼び出しに続いて直接関数 A に戻ります。テール呼び出し最適化を行わない場合、関数 C は関数 B に戻り、関数 B が即時に関数 A に戻ります。

呼び出し頻度の高いプロシージャーの場合は、テール呼び出し最適化を実行することで、不要なスタック・フレームを作成して廃棄する操作が必要なくなるた

め、パフォーマンスを向上させることができます。 LICOPT 値 NoTailCallOptimizations を指定した場合、これらの最適化は試行されません。

この LICOPT のデフォルトは、TailCallOptimizations です。

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・モデルでは概してより高速に実行されます。このオプションの TargetProcessorModel 値、関連付けられたプロセッサー・モデル、ターゲット・リリース固有のデフォルトについては、以下の表を参照してください。

プログラムのターゲット・リリース	**TargetProcessorModel** 値	指定されたプロセッサー・モデル
IBM i 7.3 および IBM i 7.2	6	POWER8
IBM i 7.1	5	POWER7
IBM i 6.1	4	POWER6
V5R4 以前	3	POWER5

TargetProcessorModel LICOPT は、モジュールの作成、モジュールまたはバインド済みモジュールの変更、またはモジュールまたはバインド済みモジュールの再作成の際にターゲットとする必要があるプロセッサー・モデルを決定するいくつかの要因の 1 つです。モジュールとバインド済みモジュールの両方に対しては、以下の規則が適用されます。

- モジュールが作成または変更される際に、TargetProcessorModel LICOPT が指定されていると、そのモジュールに生成されるコードは、CodeGenTarget LICOPT も指定されているかどうかに関わらず、指定されたプロセッサー・モデルで最適なパフォーマンスが得られるように調整されます。
- モジュールが作成または変更される際に、TargetProcessorModel LICOPT が指定されていないが、CodeGenTarget LICOPT が POWER6、POWER7、POWER8、または POWER9 として指定されている場合、そのモジュールに生成されるコードは、指定されたプロセッサー・モデルで最適なパフォーマンスが得られるように調整されます。
- モジュールが作成または変更される際に、TargetProcessorModel LICOPT が指定されていないが、CodeGenTarget LICOPT が Current として指定されている場合、そのモジュールに生成されるコードは、モジュールが常駐する区画で使用中のプロセッサーで最適なパフォーマンスが得られるように調整されます。
- モジュールが作成または変更される際に、TargetProcessorModel LICOPT が指定されていないが、CodeGenTarget LICOPT が Common として指定されている場合、そのモジュールは、該当リリースのデフォルトのプロセッサー・モデルで最適なパフォーマンスが得られるように調整されます。デフォルトのプロセッサー・モデルは、IBM i 7.3 と IBM i 7.2 の場合は POWER8、IBM i 7.1 の場合は POWER7、IBM i 6.1 の場合は POWER6、それより前のサポートされるリリースの場合は POWER5 です。

- モジュールが作成または変更される際に、TargetProcessorModel LICOPT が指定されていないが、CodeGenTarget LICOPT が Legacy として指定されている場合、そのモジュールは、POWER5 プロセッサー・モデルで最適なパフォーマンスが得られるように調整されます。
- モジュールが再作成される場合、そのモジュールが作成された時点または直前に変更された時点で指定された TargetProcessorModel LICOPT および CodeGenTarget LICOPT の値に関わらず、そのモジュールは、モジュールが常駐する区画で使用中のプロセッサーで最適なパフォーマンスが得られるように調整されます。

これらのオプションのほとんどには、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

ライセンス内部コード・オプション (LICOPT) は、モジュールの作成時に指定することができます。また、既存オブジェクトのオプションは、モジュールの変更 (CHGMOD) コマンド、プログラム変更 (CHGPGM) コマンド、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変更することができます。これらのコマンドでは、LICOPT ストリング全体を置き換えることも、既存のストリングに LICOPT を追加することもできます。変更対象のオブジェクトに別の LICOPT があり、その既存の LICOPT はそのまま残したいという場合には、既存のストリングに LICOPT を追加すると便利です。

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')
```

既存のオブジェクトに LICOPT を追加するには、LICOPT パラメーターに *ADD キーワードを使用します。例えば、LICOPT ('maf', *ADD) のように指定すると、オブジェクトの既存の LICOPT はそのまま保持され、新しい LICOPT が追加されます。

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。例えば、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。例えば、以下のようなコマンドは指定できません。

```
> CHGMOD MODULE(TEST) LICOPT('Maf,NoMaf')   <- NOT ALLOWED!
```

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。例えば、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、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、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できません。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。

IBM i では、新規のリリース (または所定のリリース内における PTF の適用) で、新規のライセンス内部コードのオプションが定義されることがあります。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、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。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、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変換対象オブジェクトに適用する

ことはありません。 CHGMOD、CHGPGM、または CHGSRVPGM コマンドに LICOPT(*SAME) を使用して再作成を行う場合、サポートされない LICOPT 値は無視されます。これらはまた、システムがオブジェクトを自動変換する際の再作成でも無視されます。対照的に、CHGMOD、CHGPGM、または CHGSRVPGM コマンドの LICOPT パラメーターにサポートされないオプションを指定しようとした場合、それらはすべて失敗します。

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。例えば、

```
    Licensed Internal Code options  . . . . . . . . . . . :   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。例えば、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。

```
    Licensed Internal Code options  . . . . . . . . . . . :   +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 196 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。例えば、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。

```
    Licensed Internal Code options  . . . . . . . . . . . :   *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。

適応コード生成

通常、基本となるハードウェア・アーキテクチャーを時間とともにスムーズに変更するために、その詳細について理解する必要はありません。アーキテクチャー変更には、単一のプロセッサー命令の追加から、プロセッサーの命令セット全体の変更までさまざまなものがあります。オペレーティング・システムが、さまざまなレベルの基本ハードウェアを持つプラットフォーム間で移行される際に、プログラムが引き続き正常に実行されるようにするには、ハードウェアから独立した形式でプログラムを表現する抽象マシン・インターフェース (MI) を使用します。

最適化変換プログラムは、MI 表現からのハードウェア命令の生成を担当します。最適化変換プログラムはオペレーティング・システムのコンポーネントであるため、それぞれのリリースごとに、それぞれのバージョンの最適化変換プログラムが 1 つ存在します。しかし、あるリリースが、互いにわずかに異なるプロセッサー・ハードウェアを持った複数のシステム・モデルでサポートされることがあります。

6.1 より前のリリースの場合、ある 1 つのリリースの最適化変換プログラムは、そのリリースでサポートされるすべてのシステム・モデルで実行される命令のみを生成するように設計されていました。このポリシーの利点は、特定のリリース用にコンパイルされたプログラムは、同じリリースを実行するすべてのシステム上で、変更なしに実行できるということです。これにより、リリースごとにソフトウェアを作成し、配布することが容易になります。しかし、大きなパフォーマンス上の利点が見込まれる重要な新規プロセッサー機能の場合は、現行のリリースがサポートするすべてのシステムにそれらの機能が備わるまで、それらの機能を使用することはできません。プロセッサー機能が使用できるようになった時期と、それがプログラム内で使用されるようになった時期には、数年の開きがあることがあります。

6.1 の時点では、システム上のすべてのプロセッサー機能を利用することができます。同じリリースでサポートされている他のシステム・モデルにこれらの機能が存在するかどうかは関係ありません。また、元のマシンで利用できたすべてのプロセッサー機能が新規のマシンに存在しなくても、あるシステム・モデルから別のシステム・モデルにプログラムを移し、正常に実行することができます。これを可能にするテクノロジーを、適応コード生成と呼びます。適応コード生成 (ACG) は、ほとんどのシナリオにおいてユーザー介入なしで動作可能です。ただし、さまざまなシステム・モデルで実行されるソフトウェアを作成して配布する場合には、適応コード生成でどのプロセッサー機能を使用するかを、ある程度制御することが必要になる場合があります。

ACG の概念

適応コード生成 (ACG) の動作の仕組みを理解するには、以下の概念を理解することが役立ちます。

ハードウェア機能 は、IBM i がサポートするプロセッサーのファミリーに追加された機能です。例えば、6.1 がサポートする一部のプロセッサーで利用できる 1 つの新規機能として、新規ハードウェアの 10 進浮動小数点装置があります。ACG は、この装置を持つプロセッサーには 10 進浮動小数点機能があると見なす一方、この装置を持たないプロセッサーには 10 進浮動小数点機能がないと見なします。 1 つのプロセッサーにあるすべての追加機能の集合体を、そのプロセッサーの機能セットと呼びます。

ターゲット・モデル とは、同じ機能セットを持つすべてのプロセッサーを表す抽象概念です。ターゲット・モデルの一例は、PowerPC AS アーキテクチャーの POWER6 レベルに準拠したすべてのプロセッサーです。

モジュールまたはプログラム・オブジェクトにもオブジェクトが必要とする機能を特定する機能セットがあるため、変更なしでもオブジェクトが正常に実行されます。オブジェクトの機能セット内のすべての機能が、ターゲット・モデルの機能セットにも存在するとき、そのオブジェクトとそのターゲット・モデルには互換性があるということになります。

コード生成 とは、モジュールまたはプログラム・オブジェクト用のハードウェア命令を作成するプロセスのことです。コード生成は、最適化変換プログラムによって実行されます。

モジュールまたはプログラム・オブジェクトは、あるシステムから別のシステムへ移すことができます。オブジェクトの存在するシステムを、現行マシン と呼びます。

通常操作

C モジュールの作成 (CRTCMOD) などのコマンドを使用してモジュール・オブジェクトをコンパイルする際、最適化変換プログラムは、システム上でどのプロセッサー機能が使用可能なのかを自動検出します。モジュール・オブジェクト用に生成されたハードウェア命令では、有用な任意のオプション・プロセッサー機能を利用することができます。最適化変換プログラムは、モジュール・オブジェクトの中で使用される機能セットを、そのオブジェクトの一部として格納します。

プログラムの作成 (CRTPGM) やサービス・プログラムの作成 (CRTSRVPGM) などのコマンドを使用してプログラム・オブジェクトを作成する際、バインド・プログラムは、そのプログラム・オブジェクト用の機能セットを判別し、それをそのプログラム・オブジェクトの一部として格納します。ある機能が、プログラムが持っている各モジュールのいずれかの機能セットに含まれている場合、その機能はそのプログラムの機能セットに含まれていることになります。

システムで最初にプログラム・オブジェクトが活動化される際、システムは、そのプログラム・オブジェクトに、システムと関連付けられたターゲット・モデルとの互換性があるか検査します。つまり、システムは、システム上で使用できない機能をプログラムが使用しないようにします。このシステムでコンパイルしたプログラム・オブジェクトは常にこの互換性検査を通過し、そのプログラムは正常に実行されます。

このプログラム・オブジェクトを、同じリリースを使用するが、ターゲット・モデルが異なる別のシステムにマイグレーションするとします。移行先のシステムで最初にそのプログラムを活動化する際に、システムは、このシステムのターゲット・モデルに対して互換性検査を実行します。プログラムにシステムとの互換性があれば、プログラムは正常に実行されます。しかし、プログラムが、移行先のシステムがサポートしていないプロセッサー機能を必要としている場合、システムは自動的に最適化変換プログラムを呼び出して、そのプログラムを互換性のあるものに変換します。変換プログラムは、新規のシステムでどの機能が使用可能なのかを検出し

て、元のモジュール・オブジェクトが作成されたときと同様に、適用可能なすべての機能を利用します。その後、要求に応じて、変換されたプログラムが活動化されます。

復元オプション

システムにモジュールおよびプログラム・オブジェクトを復元するときの適応コード生成の動作を変更するには、復元時の強制変換 (QFRCCVNRST) システム値およびオブジェクトの強制変換 (FRCOBJCVN) コマンド・パラメーターを設定します。FRCOBJCVN パラメーターは、復元 (RST)、オブジェクト復元 (RSTOBJ)、およびライブラリー復元 (RSTLIB) コマンドで使用します。

QFRCCVNRST システム値

復元時の強制変換 (QFRCCVNRST) システム値で使用可能な値は、以下のとおりです。

値	意味
0	何も変換しません。
1	妥当性検査エラーのあるオブジェクトを変換します。 (これがデフォルトです。)
2	現行バージョンのオペレーティング・システム上または現行マシン上での変換を必要とするオブジェクト、および妥当性検査エラーのあるオブジェクトを変換します。
3	損傷の疑いがあるオブジェクト、妥当性検査エラーのあるオブジェクト、および現行バージョンのオペレーティング・システム上または現行マシン上での変換を必要とするオブジェクトを変換します。
4	変換対象の十分な作成データがあり、有効なデジタル署名がないオブジェクトを変換します。
5	十分な作成データがあるオブジェクトを変換します。
6	有効なデジタル署名のないすべてのオブジェクトを変換します。
7	すべてのオブジェクトを変換します。

QFRCCVNRST を 2 または 3 に設定することによって、互換性のないプログラムおよびモジュール・オブジェクトを、最初の活動化時ではなく、復元コマンド (RST、RSTOBJ、RSTLIB) の処理時に即時に変換するようにすることができます。プログラムの変換は長時間におよぶことがあるため (大きなプログラムや高い最適化レベルでコンパイルされるプログラムの場合は特に)、場合によってはこのように設定する方がよい場合もあります。非互換の可能性があるシステムにプログラムを復元することが多い場合は、このシステム値を変更することを検討してください。

FRCOBJCVN パラメーター

復元 (RST)、オブジェクト復元 (RSTOBJ)、およびライブラリー復元 (RSTLIB) コマンドでオブジェクトの強制変換 (FRCOBJCVN) パラメーターを使用することによって、適応コード生成を制御することができます。以下の表は、これらのコマンドで使用可能な値を示したものです。例として、RSTOBJ コマンドを使用します。

値	意味
RSTOBJ FRCOBJCVN(*SYSVAL)	復元時の強制変換 (QFRCCVNRST) システム値の値に基づいてオブジェクトを変換します。 (これがデフォルトです。)
RSTOBJ FRCOBJCVN(*NO)	復元操作時にオブジェクトは変換されません。
RSTOBJ FRCOBJCVN(*YES *RQD)	現行オペレーティング・システムで使用する変換、または現行マシンと互換性のある変換が必要なオブジェクトのみを、復元操作時に変換します。
RSTOBJ FRCOBJCVN(*YES *ALL)	現行の形式やマシン互換性とは無関係に、すべてのオブジェクトを変換します (現行形式で互換性のあるオブジェクトも変換されます)。

オプション RSTOBJ FRCOBJCVN(*YES *RQD) を指定すると、復元される非互換のモジュールおよびプログラム・オブジェクトが、後で最初に活動化するときに再変換されるのではなく、即時に変換されるようになります。 QFRCCVNRST システム値は変更したくないが、このコマンドによって復元されるすべての非互換オブジェクトは即時に変換されるようにしたいという場合には、このオプションの使用を検討してください。

作成オプション

モジュールおよびプログラム・オブジェクトを作成する際、現行マシン上で使用可能なすべての機能を使用するというデフォルトの動作は、多くの場合、適切な動作となります。しかし、さまざまなシステムに配布するソフトウェアを作成している場合は、プログラムでどの機能を使用すべきかを、より慎重に指定することが必要になる場合があります。例えば、6.1 上で動作するソフトウェア・パッケージを販売する場合に、そのプログラムおよびサービス・プログラムが顧客のマシン上で変換されることは望ましくないという場合などです。こうしたことは、作成に使用したシステムが、一部の顧客のシステムよりも多くの機能を持っている場合に発生することがあります。顧客のマシンで変換を行うことが望ましくない理由としては、以下のことが挙げられます。

- 顧客が、(復元や初回活動化時に) オブジェクトの変換に時間を費やすことを望んでいない。
- 変換されたプログラムに、元のプログラムのものとは異なるハードウェア命令が含まれている。このことにより、プログラムの機能が同じままであっても、カスタマー・サポートのプロセスに影響が出る場合があります。

変換が行われないように、ターゲット・リリースがサポートしているすべてのシステムに共通の機能のみを使用してプログラムを作成するように指定することができます。これは、以下のいずれかの方法によって行うことができます。

- モジュールおよびプログラム・オブジェクトの作成時に、CodeGenTarget ライセンス内部コード・オプション (LICOPT) を指定する。
- プログラム変更 (CHGPGM) またはサービス・プログラムの変更 (CHGSRVPGM) コマンドに CodeGenTarget LICOPT を指定する。
- 環境変数を設定して、どの機能を使用するかを制御する。

CodeGenTarget LICOPT

C モジュールの作成 (CRTCMOD) やバインド C プログラムの作成 (CRTBNDC) などのモジュールやプログラムの作成コマンドの多くで、ライセンス内部コード・オプション (LICOPT) パラメーターを指定することができます。 LICOPT パラメーターは、最適化変換プログラムに、オブジェクトのハードウェア命令を作成する際に、特定のオプションを使用するように、または使用しないように指示を出します。 LICOPT CodeGenTarget オプションを使用すると、適応コード生成によって選択される機能を制御することができます。このオプションに指定可能なすべての値については、CodeGenTarget を参照してください。

モジュールおよびプログラム・オブジェクトの作成時に CodeGenTarget=Common オプションを選択することによって、ソフトウェア・プロダクトが顧客のマシン上に復元される際、または顧客が初めてソフトウェア・プロダクトを使用する際に、ソフトウェア・プロダクトが変換を必要としないようにすることができます。しかし、より多くのハードウェア機能を持つマシンであれば得られたかもしれないパフォーマンスの向上が得られないこともあります。

プログラム・オブジェクトの作成後、オブジェクトでどの機能が使用できるのかを指定すべきであったことに気付く場合があります。プログラムを再作成する代わりに、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、そのコマンドに CodeGenTarget LICOPT パラメーターを指定することができます。プログラムが変換され、最適化変換プログラムは指定された機能のみを使用します。

新機能は、PowerPC AS アーキテクチャーの POWER6、POWER7、および POWER8 のレベルで使用することができます。適応コード生成で選択される機能を制御するために、以下の `CodeGenTarget` LICOPT 値を使用できます。

POWER8 機能 (POWER8 には、POWER7 と POWER6 の全機能が含まれます) を選択するには、以下のいずれかを行います。

- CodeGenTarget=POWER8 を指定します。
- POWER8 ハードウェアで実行中の場合、CodeGenTarget=Current を指定します。

POWER7 機能 (POWER7 には、POWER6 の全機能が含まれますが、POWER8 の機能は何も含まれません) を選択するには、以下のいずれかを行います。

- CodeGenTarget=POWER7 を指定します。
- POWER7 ハードウェアで実行中の場合、CodeGenTarget=Current を指定します。
- IBM i 7.3 用に作成する場合、CodeGenTarget=Common を指定します。

POWER6 機能を選択し、POWER7 機能および POWER8 機能を選択しない場合は、以下のいずれかを行います。

- CodeGenTarget=POWER6 を指定します。
- POWER6 ハードウェアで実行中の場合、CodeGenTarget=Current を指定します。
- IBM i 7.2 用に作成する場合、CodeGenTarget=Common を指定します。

POWER6、POWER7、および POWER8 の機能を使用しないようにする場合は、以下のいずれかを行います。

- CodeGenTarget=Legacy を指定します。
- IBM i 6.1 または IBM i 7.1 用に作成する場合、CodeGenTarget=Common を指定します。

POWER6 ハードウェアと関連付けられた機能には、以下のものがあります。

- ハードウェア 10 進浮動小数点装置
- ILE ポインター処理用の高効率ハードウェア・サポート

POWER7 ハードウェアと関連付けられた機能には、以下のものがあります。

- 整数値と浮動小数点値との変換など、ある種の計算速度を高める多くの新規命令。

POWER8 ハードウェアと関連付けられた機能には、以下のものがあります。

- 浮動小数点演算を高速化するための新しい命令

モジュール・オブジェクトの場合、DETAIL(*BASIC) を指定したモジュールの表示 (DSPMOD) コマンドを使用すると、そのモジュールに適用されている LICOPT オプションが表示されます。プログラムまたはサービス・プログラム・オブジェクトの場合、LICOPT オプションは、そのプログラム内の各モジュールと関連付けられています。 DETAIL(*MODULE) を指定してプログラム表示 (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、表示するモジュールにオプション 5 を指定します。これらの表示画面の 1 つにあるライセンス内部コード・オプションの値において、一部のモデルに CodeGenTarget=model が含まれている場合があります。これは、LICOPT を直接指定するか、QIBM_BN_CREATE_WITH_COMMON_CODEGEN 環境変数を設定するかのいずれかによって、モジュールが作成されたときのデフォルトの動作を LICOPT が指定変更していることを示します。そのような LICOPT がない場合、デフォルトの動作は指定変更されていません。

ライセンス内部コード・オプションの表示方法について詳しくは、 197 ページの『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』を参照してください。

QIBM_BN_CREATE_WITH_COMMON_CODEGEN 環境変数

大きなビルドの場合、すべてのモジュールおよびプログラムの作成コマンドに CodeGenTarget LICOPT を指定するのが不便なことがあります。また、オブジェクトの作成に使用するコマンドが、LICOPT パラメーターをサポートしていない場合もあります。これらの場合には、QIBM_BN_CREATE_WITH_COMMON_CODEGEN 環境変数を使用して、ビルドにおける適応コード生成の動作を設定することができます。

環境変数を使用するには、環境変数の使用 (WRKENVVAR) コマンドを使用します。

QIBM_BN_CREATE_WITH_COMMON_CODEGEN 環境変数で使用可能な値を、以下の表に示します。

値	意味
0	新規モジュールの作成時に、指定されたとおりの LICOPT CodeGenTarget を使用します。 CodeGenTarget が指定されていない場合には、CodeGenTarget=Current を使用します。 (これがデフォルトです。)
1	新規モジュールの作成時に、指定されたとおりの LICOPT CodeGenTarget を使用します。 CodeGenTarget が指定されていない場合には、CodeGenTarget=Common を使用します。
2	新規モジュールの作成時に、常に CodeGenTarget=Common を使用して、どのような LICOPT CodeGenTarget が指定されている場合でもそれを指定変更します。

- 値 0 は、デフォルトの動作を示しています。
- 値 1 は、ほとんどのパーツで共通の機能サブセットを使用する必要があるが、個々のパーツに関してこれを指定変更したい場合に便利です。
- 値 2 は、すべてのパーツを共通の機能サブセットを使用してビルドする必要がある場合に便利です。

注: この環境変数は、新規モジュールを作成する場合にのみ適用されます。プログラム変更 (CHGPGM) コマンドなど、変更コマンドの場合には適用されません。オブジェクトの変換時にも適用されません。

ACG 情報の表示

モジュールの表示 (DSPMOD)、プログラム表示 (DSPPGM)、およびサービス・プログラムの表示 (DSPSRVPGM) コマンドを使用して、モジュールおよびプログラム・オブジェクトに現行マシンとの互換性があるかどうか、また、CodeGenTarget LICOPT を使用してデフォルトの ACG 動作を指定変更するかどうかを判別することができます。

オブジェクトの互換性

DETAIL(*BASIC) を指定したプログラム表示 (DSPPGM)、サービス・プログラムの表示 (DSPSRVPGM)、またはモジュールの表示 (DSPMOD) コマンドを使用すると、モジュールまたはプログラム・オブジェクトが最初に変換を行わなくても正常に実行可能かどうかが判別されます。必須フィールド「変換」の値は、*YES または *NO です。プログラム・オブジェクトで必須の「変換」の値が *YES になっている場合、そのプログラムは最初に活動化されたときに変換されます。あるいは、CHGPGM FRCCRT(*YES) またはオブジェクトの変換開始 (STROBJCVN) を使用して、都合のよいときに強制的に変換を実行することもできます。これら 2 つのコマンドを使用できる状況について詳しくは、 206 ページの『互換性のあるプログラムの最適化』を参照してください。

モジュール・オブジェクトの必須フィールド「変換」の値が *YES になっていて、かつ、そのモジュール・オブジェクトが古い形式 (以下の表の *FORMAT を参照) になっていない場合、このモジュール・オブジェクトをプログラムまたはサービス・プログラムにバインドした結果のプログラム・オブジェクトでは変換が必要になります。こうした結果になることは、作成システムよりも多くの機能を持ったシステムにデプロイするプログラムを作成している場合に便利です。必須フィールド「変換」の値が *NO の場合、プログラムまたはモジュール・オブジェクトは、すぐに使用できます。

モジュールまたはプログラム・オブジェクトが変換を必要とする理由は、「変換の詳細 (Conversion details)」フィールドで判別することができます。このフィールドには、以下のいずれかの値が入ります。

値	意味
*FORMAT	オブジェクトに現行マシンとの互換性はありません。オブジェクトは古い形式になっています。 (例えば、6.1 より前のリリースで作成されたオブジェクトは、6.1 で作成されたオブジェクトとは異なる形式になっています。) 古い形式のモジュール・オブジェクトをバインドすると、バインドされたモジュールが変換されます。
*FEATURE	オブジェクトに現行マシンとの互換性はありません。オブジェクトの形式に現行マシンとの互換性はありますが、現行マシンでサポートされていない機能が少なくとも 1 つ、オブジェクトで使用されています。
*COMPAT	オブジェクトには現行マシンとの互換性があります。オブジェクトの形式に現行マシンとの互換性があり、オブジェクトが使用するすべての機能が、現行マシンによってインプリメントされています。ただし、オブジェクトの作成が行われたバージョン、リリース、およびモディフィケーション・レベルでサポートされているハードウェアの共通レベルによってサポートされていない機能が少なくとも 1 つ、オブジェクトで使用されています。
*COMMON	オブジェクトには現行マシンとの互換性があります。オブジェクトの形式には互換性があり、オブジェクトが使用するすべての機能が、そのオブジェクトの作成が行われたバージョン、リリース、およびモディフィケーション・レベルによってサポートされているハードウェアの共通レベルによってインプリメントされています。

リリース間の考慮事項

適応コード生成の動作は、モジュールまたはプログラム・オブジェクトの作成時に選択したターゲットのバージョン、リリース、およびモディフィケーション・レベルによって決まります。 ACG は、6.1 以降が稼働しているシステム上でのみサポートされます。以前のターゲット・リリースのオブジェクトに CodeGenTarget LICOPT が指定されている場合、その LICOPT は許容はされますが、そのオブジェクトに対して生成されたコードでは効果を持ちません。

LICOPT('CodeGenTarget=Common') の意味も、ターゲットのバージョン、リリース、およびモディフィケーション・レベルによって決まります。この LICOPT を選択した場合の意味は、最適化変換プログラムが、ターゲット・リリース (このターゲット・リリースは、プログラムまたはモジュール・オブジェクトが作成されたリリースとは異なっている場合があります) がサポートしているすべてのマシン上で使用可能な機能セットを使用する、ということです。したがって、より新しいバージョンのオペレーティング・システムが稼働している作成マシンを使用して、前のバージョン (ただし、6.1 より前のバージョン以外) のオペレーティング・システムが稼働しているすべてのマシン上で実行される共通コードを作成することができます。

注**:** 以前のバージョンにある最適化変換プログラムを使用して、より新しいバージョンで稼働する各マシンに共通のすべての機能を利用することはできません。

互換性のあるプログラムの最適化

互換性のあるオブジェクトをマシン上に復元する際、復元コマンドに FRCOBJCVN(*YES *ALL) パラメーターを指定していない限り、通常は、それらのオブジェクトが変換されることはありません。これは、オブジェクトがマシンの機能を十分に活用しない可能性があるということです。一部の機能は、最適化変換プログラムによって使用されていない可能性があります。マシンを十分に活用するように、モジュールおよびプログラム・オブジェクトを更新することが必要になる場合があります。

モジュールおよびプログラム・オブジェクトですべての作成データが使用できるかどうかを確認する必要があります。それには、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。

- すべての作成データが使用可能で、識別することができる場合、「すべての作成データ (All creation data)」フィールドは *YES になります。 FRCCRT(*YES) を指定してモジュールの変更 (CHGMOD)、プログラム変更 (CHGPGM)、およびサービス・プログラムの変更 (CHGSRVPGM) コマンドを使用すると、強制的にオブジェクトを再作成することができます。
- すべての作成データを使用することができるが、そのすべてが識別可能というわけではない場合、「すべての作成データ (All creation data)」フィールドは *UNOBS になります。復元コマンドに FRCOBJCNV(*YES *ALL) パラメーターを指定すると、復元操作時にこれらのオブジェクトを強制的に変換することができます。
- すべての作成データを使用できるわけではない場合、「すべての作成データ」フィールドは *NO になります。オブジェクトの変更のためには、何も行うことができません。このようにプログラムに作成データが欠落している場合、6.1 以降のリリースで実行できるように、プログラムをソースから再作成する必要があります。

ACG と論理区画

複数の論理区画を持つシステムの場合、物理的に取り付けられているプロセッサー以外のプロセッサーを持つシステムを模倣するように区画を構成することができます。適応コード生成の目的上、このような区画は、模倣されるプロセッサー上で実行されているものとして扱われます。プログラム内に模倣されたプロセッサーで使用できない機能がある場合、基礎となる物理プロセッサーではそれらの機能が使用できるとしても、その模倣されたプロセッサーで実行できるようにプログラムを強制的に変換しなければならなくなります。

第 16 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。このトピックでは、共用ストレージに関連するいくつかの問題について説明します。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。そのため、このトピックで後述する問題の例で、競合状態について簡単に触れます。
- 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。例えば、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。例えば、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。

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
ユーザー待ち行列	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

```
/* 共用データに対し何らかの作業を行う */
for (int i=0; i<10; ++i) {
  /* ロックされたフラグのクリアを待機*/
  while (locked == 1) {
    sleep(1);
  }

  locked = 1;  /* ロックを設定する */

  /* 共用データを更新する */
  data1 += 5;
  data2 += 10;

  locked = 0;  /* ロックをクリアする */
}
```

スレッド B

```
/* 共用データに対し何らかの作業を行う */
for (int i=0; i<10; ++i) {
 /* ロックされたフラグのクリアを待機*/
  while (locked == 1) {
    sleep(1);
  }

  locked = 1;     /* ロックを設定する */

  /* 共用データを更新する */
  data1 += 4;
  data2 += 6;

  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、その後でロック値の検査およびロック値のクリアを使用する代替ソリューションを示します。

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。例えば、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。 (ロックが既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) {

  /* CHKLKVAL を使用してロックの取得を
```

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) {

/* CHKLKVAL を使用してロックの取得を
```

```
    試みます。規則では、値 1 でロック
    されていることを示し、値 0 でアンロック
    されていることを示します。*/
  while (_CHKLKVAL(&LOCK, 0, 1) == 1) {
    sleep(1);  /* 少し待ち再度試行します。 */
  }

  /* 共用データを更新する */
  data1 += 5;
  data2 += 10;

  /* 共有データをアンロックします。CLRLKVAL
     を使用すると、ロックの開放前に、他の
     ジョブおよびスレッドで共有データへの
     更新を確実に確認できます。*/
  _CLRLKVAL(&LOCK, 0);
}
```

```
    試みます。規則では、値 1 でロック
    されていることを示し、値 0 でアンロック
    されていることを示します。*/
while (_CHKLKVAL(&LOCK, 0, 1) == 1) {
  sleep(1);  /* 少し待ち再度試行します。 */
        }

  /* 共用データを更新する */
        data1 += 4;
        data2 += 6;

  /* 共有データをアンロックします。CLRLKVAL
     を使用すると、ロックの開放前に、他の
     ジョブおよびスレッドで共有データへの
     更新を確実に確認できます。*/
       _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。既に述べたように、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。

第 17 章 CRTPGM、CRTSRVPGM、UPDPGM、または UPDSRVPGM コマンドからの出力リスト

このトピックでは、バインド・プログラムのリストの例を示し、バインド・プログラム言語の使用により発生する可能性のあ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 101 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

216 ページの図 46、 216 ページの図 47、および 217 ページの図 48 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

 Service program  . . . . . . . . . . . . . . . . . . :   FINANCIAL
   Library  . . . . . . . . . . . . . . . . . . . . . :     MYLIB
 Export . . . . . . . . . . . . . . . . . . . . . . . :   *SRCFILE
 Export source file . . . . . . . . . . . . . . . . . :   QSRVSRC
   Library  . . . . . . . . . . . . . . . . . . . . . :     MYLIB
 Export source member . . . . . . . . . . . . . . . . :   *SRVPGM
 Activation group . . . . . . . . . . . . . . . . . . :   *CALLER
 Allow update  . . . . . . . . . . . . . . . . . . .  :   *YES
 Allow bound *SRVPGM library name update . . . . . .  :   *NO
 Creation options . . . . . . . . . . . . . . . . . . :   *GEN        *NODUPPROC  *NODUPVAR   *DUPWARN
 Listing detail . . . . . . . . . . . . . . . . . . . :   *FULL
 User profile . . . . . . . . . . . . . . . . . . . . :   *USER
 Replace existing service program . . . . . . . . . . :   *YES
 Target release . . . . . . . . . . . . . . . . . . . :   *CURRENT
 Allow reinitialization . . . . . . . . . . . . . . . :   *NO
 Storage model  . . . . . . . . . . . . . . . . . . . :   *SNGLVL
 Argument optimization  . . . . . . . . . . . . . . . :   *NO
 Interprocedural analysis . . . . . . . . . . . . . . :   *NO
 IPA control file . . . . . . . . . . . . . . . . . . :   *NONE
Authority   . . . . . . . . . . . . . . . . . . . . . :   *LIBCRTAUT
 Text . . . . . . . . . . . . . . . . . . . . . . . . :

 Module      Library             Module      Library
 MONEY       MYLIB               CALCS       MYLIB
 RATES       MYLIB               ACCTS       MYLIB

 Service
 Program     Library     Activation
 *NONE

 Binding
 Directory   Library
 *NONE
```

図 *46. 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

 Program entry procedures . . . . . . . . . . . :   0

 Multiple strong definitions  . . . . . . . . . :   0

 Unresolved references  . . . . . . . . . . . . :   0


                    * * * * *  簡 略 要 約 表 の 終 わ り  * * * * *
```

図 *47.* 簡略要約表

```
                                サービス・プログラムの作成                    ページ   23

                                バインディング統計

 Symbol collection CPU time . . . . . :              .018
 Symbol resolution CPU time . . . . . :              .006
 Binding directory resolution CPU time  . . . . . :  .403
 Binder language compilation CPU time . . . . . :    .040
 Listing creation CPU time  . . . . . :             1.622
 Program/service program creation CPU time  . . . . . :  .178

 Total CPU time . . . . . :                         2.761
 Total elapsed time . . . . . :                    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 *48.* 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。以下のリストは、追加データの例です。

```
                                サービス・プログラムの作成                    ページ   2

                                拡張要約表

  Valid definitions  . . . . . :          418
    Strong . . . . . :          418
    Weak . . . . . :            0
  Resolved references  . . . . . :          21
    To strong definitions  . . . . . :          21
    To weak definitions  . . . . . :            0

                        * * * * *  拡 張 要 約 表 の 終 わ り  * * * * *

                                サービス・プログラムの作成                    ページ   4

                                バインド・プログラム情報リスト

Module . . . . . :   MONEY
  Library  . . . . . :     MYLIB
  Bound  . . . . . :     *YES

      Number    Symbol      Ref       Identifier                      Type      Scope       Export       Key

      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                 データ
      00000005  参照          0000017E  Q_LE_AG_user_rc                 データ
      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

Module . . . . . :   RATES
  Library  . . . . . :     MYLIB
  Bound  . . . . . :     *YES

      Number    Symbol      Ref       Identifier                      Type      Scope       Export       Key
```

```
      0000000A  定義                Term                          PROC      SRVPGM      強
      0000000B  定義                Rate                          PROC      SRVPGM      強
      0000000C  参照        0000017F  Q LE AG_prod_rc               データ
      0000000D  参照        0000017E  Q LE AG_user_rc               データ
      0000000E  参照        00000180  Q LE leDefaultEh              PROC
      0000000F  参照        00000181  Q LE mhConversionEh           PROC
      00000010  参照        00000125  _C_exception_router           PROC

Module . . . . . :   CALCS
  Library  . . . . . :     MYLIB
  Bound  . . . . . :     *YES

      Number    Symbol      Ref       Identifier                    Type      Scope       Export       Key

      00000011  定義                  Calc1                         PROC      MODULE      強
      00000012  定義                  Calc2                         PROC      MODULE      強
      00000013  参照        0000017F  Q LE AG_prod_rc               データ
      00000014  参照        0000017E  Q LE AG_user_rc               データ
      00000015  参照        00000180  Q LE leDefaultEh              PROC
      00000016  参照        00000181  Q LE mhConversionEh           PROC
      00000017  参照        00000125  _C_exception_router           PROC

Module . . . . . :   ACCTS
  Library  . . . . . :     MYLIB
  Bound  . . . . . :     *YES

      Number    Symbol      Ref       Identifier                    Type      Scope       Export       Key

      00000018  定義                  OpenAccount                   PROC      SRVPGM      強
      00000019  定義                  CloseAccount                  PROC      SRVPGM      強
      0000001A  参照        0000017F  Q LE AG_prod_rc               データ
      0000001B  参照        0000017E  Q LE AG_user_rc               データ
      0000001C  参照        00000180  Q LE leDefaultEh              PROC
      0000001D  参照        00000181  Q LE mhConversionEh           PROC
      0000001E  参照        00000125  _C_exception_router           PROC
Service program  . . . . . :   QC2SYS
  Library  . . . . . :     *LIBL
  Bound  . . . . . :     *NO

      Number    Symbol      Ref       Identifier                    Type      Scope       Export       Key

      0000001F  定義                  system                        PROC                  強

Service program  . . . . . :   QLEAWI
  Library  . . . . . :     *LIBL
  Bound  . . . . . :     *YES

      Number    Symbol      Ref       Identifier                    Type      Scope       Export       Key

      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
                                        サービス・プログラムの作成                        ページ  14

                                        バインド・プログラム言語リスト

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: 00000000000000000ADC89D09E0C6E7

                        *****  END OF BINDER LANGUAGE LISTING  *****
```

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。以下のリストは、提供される追加データの例の一部です。

相互参照リスト

ID	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

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

215 ページの『基本リスト』、 217 ページの『拡張リスト』、および 218 ページの『フル・リスト』のリストは、 107 ページの図 35 の FINANCIAL サービス・プログラムの作成のために DETAIL(*FULL) が指定されたときに生成されるリスト・データの一部を示します。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 (217 ページの『拡張リスト』) 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。 例えば 217 ページの『拡張リスト』 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

218 ページの『フル・リスト』 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

217 ページの図 48 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に可変文字が入っている。
- 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 49 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。

```
                                Binder Language Listing

              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

           * * * * *  E N D  O F  B I N D E R  L A N G U A G E  L I S T I N G  * * * * *
```

図 *49.* 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

226 ページの図 50 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。

```
                                Binder Language Listing

            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

          * * * * *  E N D  O F  B I N D E R  L A N G U A G E  L I S T I N G  * * * * *
```

図 *50.* 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 51 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。

```
                                Binder Language Listing

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: 0000000000000000000000000000CD2
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: 0000000000000000000000000000CDE3
******** 現行エクスポート・ブロック限界インターフェース。


          * * * * *  E N D  O F  B I N D E R  L A N G U A G E  L I S T I N G  * * * * *
```

図 *51. 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 52 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。

```
                        Binder Language Listing

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: 000000000000000000000000000000CD2
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: 000000000000000000000000000000CD2
******** 重複エクスポート・ブロック。

          *****  END OF BINDER LANGUAGE LISTING  *****
```

図 *52.* 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 53 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。

```
                                Binder Language Listing

 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
   EXPORT SYMBOL(C)
 ENDPGMEXP
******** エクスポート・インターフェース識別値: 00000000000000000000000000CDED3


          * * * * *  E N D  O F  B I N D E R  L A N G U A G E  L I S T I N G  * * * * *
```

図 *53.* 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。

229 ページの図 54 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。

```
STRPGMEXP  PGMLVL(*CURRENT) LVLCHK(*NO)
  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) LVLCHK(*NO)
******** 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

          *****  E N D  O F  B I N D E R  L A N G U A G E  L I S T I N G  *****
```

図 *54.* 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

230 ページの図 55 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。

```
                                Binder Language Listing

 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


           * * * * *  E N D  O F  B I N D E R  L A N G U A G E  L I S T I N G  * * * * *
```

図 *55. 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 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。

```
                                Binder Language Listing

 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。


           * * * * *  E N D  O F  B I N D E R  L A N G U A G E  L I S T I N G  * * * * *
```

図 *56. 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 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。

```
                        Binder Language Listing

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。

          * * * * *  E N D  O F  B I N D E R  L A N G U A G E  L I S T I N G  * * * * *
```

図 *57. 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 58 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。

```
                        Binder Language Listing

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。


          *****  E N D  O F  B I N D E R  L A N G U A G E  L I S T I N G  *****
```

図 *58. 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

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。

```
                        Binder Language Listing

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: 0000000000000000000000000000000C1


          *****  E N D  O F  B I N D E R  L A N G U A G E  L I S T I N G  *****
```

図 *59. 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 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。

```
                        Binder Language Listing

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: 000000000000000000000000000000CD2
  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。

          * * * * *  E N D  O F  B I N D E R  L A N G U A G E  L I S T I N G  * * * * *
```

図 *60.* 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、既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

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。

```
                    Binder Language Listing

STRPGMEXP PGMLVL(*CURRENT)
EXPORT ("A"<<<)
***ERROR 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 *61.* 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 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。

```
                        Binder Language Listing

STRPGMEXP  PGMLVL(*PRV)
  EXPORT SYMBOL(A)
ENDPGMEXP
******** エクスポート・インターフェース識別値: 000000000000000000000000000000C1
***ERROR 「現行」エクスポート・ブロックがない。


          * * * * *  E N D  O F  B I N D E R  L A N G U A G E  L I S T I N G  * * * * *
```

図 *62. 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

236 ページの図 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。

```
                                Binder Language Listing

STRPGMEXP PGMLVL(*CURRENT)
EXPORT ("Z"<<<)
***ERROR 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: 0000000000000000000000000000FFC2

    * * * * *  B I N D プ ロ グ ラ ム 言 語 リ ス ト の 終 わ り  * * * * *
```

図 *63.* 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 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。

```
                                Binder Language Listing

 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: 0000000000000000000000000000CD2
 STRPGMEXP  PGMLVL(*PRV)
 ENDPGMEXP
 ******** エクスポート・インターフェース識別値: 00000000000000000000000000000000
 ***ERROR 前のエクスポート・ブロックが空である。

           * * * * *  E N D  O F  B I N D E R  L A N G U A G E  L I S T I N G  * * * * *
```

図 *64. 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いる

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。

```
                                Binder Language Listing

             STRPGMEXP  SIGNATURE('\!CDEFGHIJKLMNOP')
***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: E05A838485868788899192939495969 7

           * * * * *  E N D  O F  B I N D E R  L A N G U A G E  L I S T I N G  * * * * *
```

図 *65.* インターフェース識別値に可変文字が入っている

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. 可変文字を除去します。
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 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。

```
                                Binder Language Listing

             STRPGMEXP  SIGNATURE('ABCDEFGHIJKLMNOP')  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: C1C2C3C4C5C6C7C8C9D1D2D3D4D5D6D7

           * * * * *  E N D  O F  B I N D E R  L A N G U A G E  L I S T I N G  * * * * *
```

図 *66. 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 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。

```
                              Binder Language Listing

             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

            * * * * *  E N D  O F  B I N D E R  L A N G U A G E  L I S T I N G  * * * * *
```

図 *67.* 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 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。

```
                              Binder Language Listing

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: 00000000000000000000000000000C1


            * * * * *  E N D  O F  B I N D E R  L A N G U A G E  L I S T I N G  * * * * *
```

図 *68.* 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、ソース行 EXPORT SYMBOL("") は、バインド・プログラム言語ソース・ファイルから削除されています。

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 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。

```
                        Binder Language Listing

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

          *****  E N D  O F  B I N D E R  L A N G U A G E  L I S T I N G  *****
```

図 *69.* 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 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。

```
                                Binder Language Listing

 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

            * * * * *  E N D  O F  B I N D E R  L A N G U A G E  L I S T I N G  * * * * *
```

図 *70.* 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。

第 18 章 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。このトピックでは、このメッセージが出される例を 1 つ挙げて説明します。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。

第 19 章 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 14. 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 15. 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 16. 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 17. 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 18. 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
CRTSQLCPPI	SQL ILE C++ 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 19. 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 20. 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 21. 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

第 20 章 関連情報

IBM i オペレーティング・システムの ILE 環境に関するトピックの詳細は、以下の資料を参照してください。

- バックアップおよび回復カテゴリーでは、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 プログラミング資料は、オブジェクトとライブラリー、CL プログラミング、プログラム間の制御の流れと連絡、CL プログラムでのオブジェクトの処理、および CL プログラムの作成についての概要説明など、プログラミングに関するトピックを広範囲にわたって説明しています。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。
- 「通信管理」  は、通信環境での実行管理、通信状況、通信問題のトレースと診断、エラーの処理と回復、パフォーマンス、および特定の回線速度とサブシステム・ストレージ情報に関する情報を提供します。
- 「ICF Programming」  は、通信機能およびシステム間通信機能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。
- Rational® Developer for i: ILE C/C++ プログラマーの手引き」  は、ILE C プログラムと ILE C++ プログラムの作成、コンパイル、デバッグ、および実行について説明しています。このガイドには、ILE および IBM i のプログラミング機能、ファイル・システム、装置、および機能、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。
- 「Rational Developer for i: ILE C/C++ 解説書」  は、プログラミング言語 - *C* 規格とプログラミング言語 - *C++* 規格への言語の適合性について説明しています。
- 「Rational Developer for i: ILE C/C++ コンパイラー参照」  には、プリプロセッサー・ステートメント、マクロ、プラグマ、IBM i および Qshell 環境

でのコマンド行の使用、ならびに入出力に関する考慮事項を含む、ILE C/C++ コンパイラーの参照情報が記載されています。

- 「ILE C/C++ ランタイム・ライブラリー関数」 は、ILE C/C++、ランタイム・ライブラリー関数、組み込みファイル、およびランタイムに関する考慮事項の参照情報を提供します。

- 「Rational Developer for i: ILE COBOL プログラマーの手引き」 は、IBM i オペレーティング・システムで ILE COBOL プログラムを作成、コンパイル、バインド、実行、デバッグ、および保守する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。

- 「Rational Developer for i: ILE COBOL 解説書」 は、ILE COBOL プログラミング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。

- 「Rational Developer for i: ILE RPG プログラマーの手引き」 は、IBM i オペレーティング・システムの統合化言語環境 (ILE) での ILE RPG のインプリメンテーションである、RPG IV プログラミング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 IBM i のファイルと装置の使用方法を説明しています。この資料のトピックには、RPG IV への移行に関する情報とコンパイラー・リストのサンプルが含まれます。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。

- 「Rational Developer for i: ILE RPG 解説書」 は、 RPG IV プログラミング言語を使用して IBM i オペレーティング・システムのプログラムを作成するために必要な情報を提供します。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、同じシステム上の 2 つのアプリケーション・プログラム間の対話式通信に関する情報を提供します。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、システム間通信機能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。

- 「機密保護解説書」  は、適切な許可のないユーザーがシステムやデータを使用するのを防止するため、データが意図的または偶発的に損傷または破壊されるのを防止するため、セキュリティー情報を常に最新に保つため、および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。
- IBM i 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。

第 2 部 付録

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。実施権についてのお問い合わせは、書面にて下記宛先にお送りください。

〒103-8510
東京都中央区日本橋箱崎町19番21号
日本アイ・ビー・エム株式会社
法務・知的財産
知的財産権ライセンス渉外

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 YBWA
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。

この文書に含まれるいかなるパフォーマンス・データも、管理環境下で決定されたものです。そのため、他の操作環境で得られた結果は、異なる可能性があります。一部の測定が、開発レベルのシステムで行われた可能性がありますが、その測定値が、一般に利用可能なシステムのものと同じである保証はありません。さらに、一部の測定値が、推定値である可能性があります。実際の結果は、異なる可能性があります。お客様は、お客様の特定の環境に適したデータを確かめる必要があります。

IBM 以外の製品に関する情報は、その製品の供給者、出版物、もしくはその他の公に利用可能なソースから入手したものです。 IBM は、それらの製品のテストは行っておりません。したがって、他社製品に関する実行性、互換性、またはその他の要求については確証できません。 IBM 以外の製品の性能に関する質問は、それらの製品の供給者にお願いします。

IBM の将来の方向または意向に関する記述については、予告なしに変更または撤回される場合があり、単に目標を示しているものです。

本書はプランニング目的としてのみ記述されています。記述内容は製品が使用可能になる前に変更になる場合があります。

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。これらのサンプル・プログラムは特定物として現存するままの状態で提供されるものであり、いかなる保証も提供されません。 IBM は、お客様の当該サンプル・プログラムの使用から生ずるいかなる損害に対しても一切の責任を負いません。

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_.

プログラミング・インターフェース情報

本書「ILE C/C++ コンパイラー参照」には、プログラムを作成するユーザーが IBM i のサービスを使用するためのプログラミング・インターフェースが記述されています。

商標

IBM、IBM ロゴおよび ibm.com は、世界の多くの国で登録された International Business Machines Corporation の商標です。他の製品名およびサービス名等は、それぞれ IBM または各社の商標である場合があります。現時点での IBM の商標リストについては、『www.ibm.com/legal/copytrade.shtml』 をご覧ください。

Adobe、Adobe ロゴ、PostScript、PostScript ロゴは、Adobe Systems Incorporated の米国およびその他の国における登録商標または商標です。

Linux は、Linus Torvalds の米国およびその他の国における登録商標です。

Microsoft、Windows、Windows NT 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。

UNIX は The Open Group の米国およびその他の国における登録商標です。

Java™ およびすべての Java 関連の商標およびロゴは Oracle やその関連会社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。

他の製品名およびサービス名等は、それぞれ IBM または各社の商標である場合があります。

使用条件

これらの資料は、以下の条件に同意していただける場合に限りご使用いただけます。

個人使用**:** これらの資料は、すべての著作権表示その他の所有権表示をしていただくことを条件に、非商業的な個人による使用目的に限り複製することができます。ただし、IBM の明示的な承諾をえずに、これらの資料またはその一部について、二次的著作物を作成したり、配布 (頒布、送信を含む) または表示 (上映を含む) することはできません。

商業的使用**:** これらの資料は、すべての著作権表示その他の所有権表示をしていただくことを条件に、お客様の企業内に限り、複製、配布、および表示することができます。 ただし、IBM の明示的な承諾をえずにこれらの資料の二次的著作物を作成したり、お客様の企業外で資料またはその一部を複製、配布、または表示することはできません。

ここで明示的に許可されているもの以外に、資料や資料内に含まれる情報、データ、ソフトウェア、またはその他の知的所有権に対するいかなる許可、ライセンス、または権利を明示的にも黙示的にも付与するものではありません。

資料の使用が IBM の利益を損なうと判断された場合や、上記の条件が適切に守られていないと判断された場合、IBM はいつでも自らの判断により、ここで与えた許可を撤回できるものとさせていただきます。

お客様がこの情報をダウンロード、輸出、または再輸出する際には、米国のすべての輸出入関連法規を含む、すべての関連法規を遵守するものとします。

IBM は、これらの資料の内容についていかなる保証もしません。これらの資料は、特定物として現存するままの状態で提供され、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なしで提供されます。

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］

プログラム番号: 5770-SS1

Printed in Japan

日本アイ・ビー・エム株式会社
〒103-8510 東京都中央区日本橋箱崎町19-21