Natural for Windows

プログラミングガイド

バージョン 6.3.3

October 2008

Natural

This document applies to Natural バージョン 6.3.3 and to all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

目次

1 プログラミングガイド

このガイドは、Natural リファレンスドキュメントを補足するものです。Natural プログラミングのさまざまな面に関する基本情報、および詳細情報を提供しています。 Natural アプリケーションの作成を始める前に、これらの情報を理解しておく必要があります。「ファーストステップ」も参照してください。 このチュートリアルには、Natural プログラミングの基礎を紹介する一連のセッションが含まれています。

	Natural プログラミングモード	レポーティングモードとストラクチャードモードという 2 つの Natural プログラミングモードの違いについて説明します。 アプリケーションの構造がより明確になるため、通常はストラクチャードモードを排他的に使用することをお勧めします。 したがって、このドキュメントのすべての説明と例は、ストラクチャードモードについて述べられています。 レポーティングモード独自の特性は考慮されていません。
	オブジェクトタイプ	アプリケーション内では、効率的なアプリケーション構造が可能となるよう、いくつかのタイプのプログラミングオブジェクトを使用できます。このドキュメントでは、データエリア、プログラム、サブプログラム、サブルーチン、ヘルプルーチン、マップなど、さまざまなタイプの Natural プログラミングオブジェクトについて説明します。
	フィールドの定義	プログラムで使用するフィールドを定義する方法について説明します。
	ユーザー定義関数	Natural プログラミングオブジェクト 「関数」 を使用する利点について説明し、ファンクションコールとサブプログラムコールの使用の違いを示し、関数の定義と呼び出しに使用できるメソッドについて説明します。
	データベース内のデータへのアクセス	Adabas データベースおよび Natural でサポートされる Adabas 以外の各種のデータベース内のデータに Natural を使用してアクセスする方法のさまざまな面について説明します。 原則として、Adabas に関して述べられている機能や例は、他のデータベース管理システムにも適用されます。 違いがある場合は、関連するインターフェイスドキュメントおよび『ステートメント』ドキュメントまたは『パラメータリファレンス』で説明しています。

データ出力制御	Naturalで作成した出力レポートのフォーマットを制御する方法、つまりデータが表示される方法のさまざまな面について説明します。
プログラミングのその他のポイント	Naturalでのプログラミングに関して、他のドキュメントで述べられていない面について説明します。
ポータブル **Natural** 生成プログラム	Natural 5 以降で生成したプログラムは、UNIX、OpenVMS、およびWindows のプラットフォーム間で移植可能です。
イベントドリブンプログラミングについて	イベントドリブンプログラミングの基本情報を提供します。
イベントドリブンプログラミングの手法	経験を積んだGUIプログラマを対象に、必要不可欠なプログラミング技法について説明します。
結果インターフェイス	結果インターフェイスにより、プログラマはNaturalスタジオの結果ウィンドウ内にデータを表示できるようになります。
アプリケーションに対するキャラクタユーザーインターフェイスの設計	アプリケーションで使用するためのキャラクタベースのユーザーインターフェイスを設計するために使用できる Natural のコンポーネントに関する情報を提供します。
Natural Native Interface	Natural 以外のアプリケーションで C のファンクションコールを使用して Natural コードを実行できるようにする Natural Native Interface について説明します。
NaturalX	オブジェクトベースアプリケーションを開発および配布する方法について説明します。
Natural 予約キーワード	Naturalプログラミング言語で予約されているすべてのキーワードと語のリストを示します。
参照プログラム例	『*プログラミングガイド*』の上記の各セクションには、Naturalプログラムの例が数多く含まれています。 これらのプログラム例には、その他の例（主にレポーティングモードに関するもの）へのリンクも提供されており、それがこの別個のセクションに記載されています。 **注意:** 1. 『*プログラミングガイド*』に示されているすべてのプログラム例は、Natural ライブラリ `SYSEXPG` でもソースコード形式で提供されています。プログラム例では、Software AG提供のデモ用ファイル `EMPLOYEES` と `VEHICLES` のデータを使用します。 2. Natural ステートメントを使用したその他のプログラム例は、Natural ライブラリ `SYSEXSYN` で提供され、『ステートメント』ドキュメントの「*参照プログラム例*」セクションで説明されています。 3. ライブラリ `SYSEXPG` および `SYSEXSYN` が使用可能であるかを Natural 管理者に確認してください。 4. Naturalプログラム例を使用してAdabasデータベースにアクセスするには、Naturalニュークリアスパラメータ `OPTIONS` を `TRUNCATION` に設定する必要があります。

Note: Naturalアプリケーションプログラミングインターフェイス（API）の詳細については、『ユーティリティ』ドキュメントの「*SYSEXT-Natural アプリケーションプログラミ*

ングインターフェイス」および「*SYSAPI - Natural アドオン製品の API*」を参照してください。

2 Natural プログラミングモード

このchapterでは、Natural で提供される 2 つのプログラミングモードについて説明します。 次のトピックについて説明します。

プログラミングモードの目的

Natural では次の 2 つのプログラミング方法が提供されます。

- レポーティングモード
- ストラクチャードモード

Note: アプリケーションの構造がより明確になるため、通常はストラクチャードモードを排他的に使用することをお勧めします。

レポーティングモード

レポーティングモードが役に立つのは、複雑なデータやプログラミング構成を必要としない、アドホックレポートおよび小さなプログラムを作成する場合のみです。 レポーティングモードでプログラムを作成すると、小さなプログラムが大きくなり、かつ複雑になりやすいので注意してください。

一部のNaturalステートメントはレポーティングモードでのみ使用できること、また、その他のステートメントはレポーティングモードで使用すると特別な構造になることに注意してください。 レポーティングモードで使用可能なステートメントの概要については、『ステートメント』ドキュメントの「*レポーティングモードのステートメント*」を参照してください。

ストラクチャードモード

ストラクチャードモードは、明確で適切に定義されたプログラム構成で複雑なアプリケーションを実装するときに使用します。 ストラクチャードモードの主な利点は、次のとおりです。

- プログラムを構造化して記述する必要があるため、読みやすく、管理しやすくなります。
- プログラムで使用するすべてのフィールドを、レポーティングモードでできるようなプログラム全体に散在させるのではなく、1つの一元的な場所に定義する必要があるため、データの全体的な制御がはるかに容易になります。

ストラクチャードモードでは、実際のプログラムをコーディングする前に、より詳細な計画を立てる必要があります。このため、多くのプログラミングエラーや非効率な作業を回避できます。

ストラクチャードモードで使用可能なステートメントの概要については、『ステートメント』ドキュメントの「*機能別ステートメント*」を参照してください。

プログラミングモードの設定と変更

デフォルトのプログラミングモードは、Natural 管理者がプロファイルパラメータ SM で設定します。設定されたモードは、システムコマンド GLOBALS とセッションパラメータ SM を使用して変更できます。

ストラクチャードモード：	GLOBALS SM=ON
レポーティングモード：	GLOBALS SM=OFF

Naturalのプロファイルおよびセッションパラメータ SMの詳細については、『パラメータリファレンス』の「*SM - ストラクチャードモードでのプログラミング*」を参照してください。

プログラミングモードの変更方法の詳細については、『パラメータリファレンス』の「*SM - ストラクチャードモードでのプログラミング*」を参照してください。

機能上の違い

レポーティングモードとストラクチャードモード間の機能上の主な違いは、次のとおりです。

- ループおよび機能ブロックを閉じるための構文
- レポーティングモードで処理ループを閉じる
- ストラクチャードモードで処理ループを閉じる
- プログラム内のデータ要素の位置
- データベース参照

Note: これら 2 つのモード間の機能上の違いの詳細については、『ステートメント』ドキュメントを参照してください。 モードが区別されるステートメントごとに、個別の構文図と構文要素を説明しています。 レポーティングモードで使用可能なステートメントの機能概要については、『ステートメント』ドキュメントの「*レポーティングモードのステートメント*」を参照してください。

ループおよび機能ブロックを閉じるための構文

レポーティングモード：	この目的には、(CLOSE) LOOP ステートメントと DO ... DOEND ステートメントを使用できます。 END-DEFINE、END-DECIDE、および END-SUBROUTINE 以外の END-... ステートメントは使用できません。
ストラクチャードモード：	すべてのループや論理構成は、対応する END-... ステートメントで明示的に閉じる必要があります。 したがって、どのループまたは論理構成がどこで終わるかがすぐにわかります。

	LOOP ステートメントおよび DO/DOEND ステートメントは使用できません。

次の 2 つの例は、処理ループと論理条件の構成における 2 つのモードの違いを示しています。

レポーティングモードの例：

レポーティングモードの例では、DO ステートメントおよび DOEND ステートメントを使用して、AT END OF DATA 条件に基づくステートメントブロックの開始と終了を示しています。 END ステートメントによって、アクティブな処理ループがすべて閉じます。

```
READ EMPLOYEES BY PERSONNEL-ID
DISPLAY NAME BIRTH
AT END OF DATA
   DO
      SKIP 2
      WRITE / 'LAST SELECTED:' OLD(NAME)
   DOEND
END
```

ストラクチャードモードの例：

ストラクチャードモードの例では、END-ENDDATA ステートメントを使用して AT END OF DATA 条件を閉じ、END-READ ステートメントを使用して READ ループを閉じています。 結果として、プログラムはさらに明確に構造化され、各構成がどこで開始し、どこで終了するかを即座に確認できます。

```
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 BIRTH

END-DEFINE
READ MYVIEW BY PERSONNEL-ID
   DISPLAY NAME BIRTH
   AT END OF DATA
      SKIP 2
      WRITE / 'LAST SELECTED:' OLD(NAME)
   END-ENDDATA
```

```
END-READ
END
```

レポーティングモードで処理ループを閉じる

処理ループを閉じるには、END、LOOP（またはCLOSE LOOP）、およびSORTのいずれかのステートメントを使用できます。

LOOPステートメントを使用すると複数のループを閉じることができ、ENDステートメントを使用するとアクティブなすべてのループを閉じることができます。1つのステートメントで複数のループを閉じることができるのは、ストラクチャードモードと基本的に異なる点です。

SORTステートメントはすべての処理ループを閉じてから、別の処理ループを開始します。

例 **1 - LOOP**：

```
FIND ...
   FIND ...
   ...
   ...
   LOOP      /* closes inner FIND loop
LOOP         /* closes outer FIND loop
...
...
```

例 **2 - END**：

```
FIND ...
   FIND ...
   ...
   ...
END             /* closes all loops and ends processing
```

例 **3 - SORT**：

```
FIND ...
   FIND ...
   ...
   ...
SORT ...        /* closes all loops, initiates loop
```

```
...
END             /* closes SORT loop and ends processing
```

ストラクチャードモードで処理ループを閉じる

ストラクチャードモードでは、それぞれの処理ループに対してループを閉じるための特定のステートメントを使用します。ENDステートメントで処理ループを閉じることはできません。SORTステートメントの前にEND-ALLステートメントを指定する必要があり、SORTループはEND-SORTステートメントで閉じる必要があります。

例 **1 - FIND**：

```
FIND ...
   FIND ...
   ...
   ...
   END-FIND       /* closes inner FIND loop
END-FIND          /* closes outer FIND loop
...
```

例 **2 - READ**：

```
READ ...
   AT END OF DATA
   ...
   END-ENDDATA
   ...
END-READ           /* closes READ loop
...
...
END
```

例 **3 - SORT**：

```
READ ...
   FIND ...
   ...
   ...
END-ALL         /* closes all loops
SORT            /* opens loop
...
...
```

```
END-SORT        /* closes SORT loop
END
```

プログラム内のデータ要素の位置

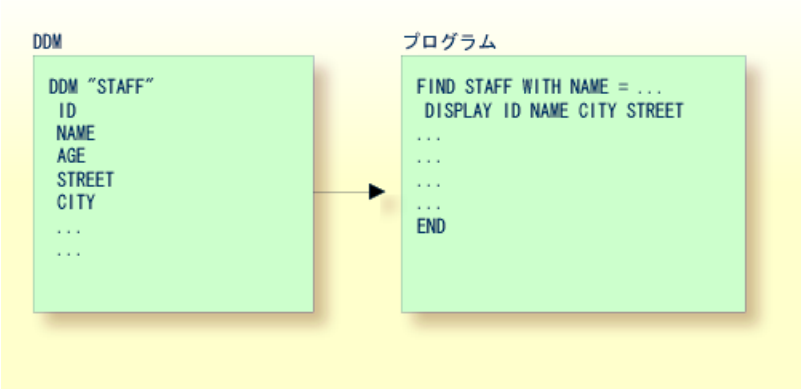
レポーティングモードでは、DEFINE DATAステートメントで定義する必要なくデータベースフィールドを使用することができるうえ、プログラムのどこにでもユーザー定義変数を定義できるので、ユーザー定義変数をプログラム全体に散在させることができます。

ストラクチャードモードでは、使用するすべてのデータ要素を1つの一元的な場所、つまりプログラムの先頭の DEFINE DATA ステートメント またはプログラム外部のデータエリアに定義する必要があります。

データベース参照

レポーティングモード：

レポーティングモードでは、データベースフィールドとDDMは、データエリアにあらかじめ定義しておかなくても参照できます。



ストラクチャードモード：

ストラクチャードモードでは、使用する各データベースフィールドを、「フィールドの定義」および「*Adabas* データベースのデータへのアクセス」で説明されているように、DEFINE DATA ステートメントに指定する必要があります。


DDM
DDM "STAFF"
ID
NAME
AGE
STREET
CITY
...
...
プログラム
DEFINE DATA LOCAL
1 VIEWXYZ VIEW OF STAFF
2 ID
2 NAME
2 AGE
2 STREET
2 CITY
END-DEFINE
*
FIND VIEWXYZ WITH NAME = ...
DISPLAY ID NAME CITY STREET
...
...
END-FIND
...
END

3 オブジェクトタイプ

ここでは、効率的なアプリケーション構造を実現するために使用できるNaturalオブジェクトの各種タイプについて説明します。 すべてのNaturalオブジェクトは、Naturalライブラリに保存されています。 Naturalライブラリは、Naturalシステムファイルに含まれています。

次のトピックについて説明します。

- 使用可能なオブジェクトのタイプ
- データエリア
- プログラム、関数、サブプログラム、およびサブルーチン
- リッチ **GUI** ページの処理 **-** アダプタ
- マップ
- ヘルプルーチン
- ソースコードの複数使用 **-** コピーコード
- **Natural** オブジェクトのドキュメント化 **-** テキスト
- イベントドリブンアプリケーションの作成 **-** ダイアログ
- コンポーネントベースのアプリケーションの作成 **-** クラス
- **Natural** 以外のファイルの使用 **-** リソース

4 使用可能なオブジェクトのタイプ

このchapterでは、次のトピックについて説明します。

プログラミングオブジェクトのタイプ

Naturalアプリケーションでは、いくつかのタイプのオブジェクトを使用して、効率的なアプリケーション構造を実現できます。

Natural オブジェクトのタイプは次のとおりです。

- プログラム
- クラス
- サブプログラム
- ファンクション
- アダプタ
- サブルーチン
- コピーコード
- ヘルプルーチン
- テキスト
- ダイアログ
- マップ
- ローカルデータエリア
- グローバルデータエリア
- パラメータデータエリア
- リソース

オブジェクトの作成および管理

これらすべてのオブジェクトは、Natural エディタを使用して作成および管理します。

- ローカルデータエリア、グローバルデータエリア、およびパラメータデータエリアは、データエリアエディタで作成および管理します。
- マップは、マップエディタで作成および管理します。
- ダイアログは、ダイアログエディタで作成および管理します。
- クラスは、クラスビルダで作成および管理します。
- 上記以外のすべてのオブジェクトは、プログラムエディタで作成および管理します。

Naturalオブジェクトに適用される命名規則の詳細については、「*オブジェクトの命名規則*」を参照してください。

『*Natural スタジオの使用*』ドキュメントの「*Natural オブジェクトの管理*」も参照してください。

5 データエリア

このchapterでは、次のトピックについて説明します。

データエリアの使用

「*フィールドの定義*」で説明されているように、プログラムで使用するすべてのフィールドは DEFINE DATA ステートメントに定義する必要があります。

フィールドは DEFINE DATA ステートメント内に直接定義できます。または、プログラム外部の別個のデータエリアにフィールドを定義し、DEFINE DATA ステートメントでそのデータエリアを参照できます。

別個のデータエリアとは、複数の Natural プログラム、サブプログラム、サブルーチン、ヘルプルーチン、ダイアログ、またはクラスで使用できる Natural オブジェクトを指します。 データエリアには、データ定義モジュール（DDM）から取得されるユーザー定義変数、定数、データベースフィールドなどのデータ要素定義が含まれています。

すべてのデータエリアは、データエリアエディタで作成および編集します。

Natural では、次の 3 つのタイプのデータエリアがサポートされます。

- ローカルデータエリア
- グローバルデータエリア
- パラメータデータエリア

ローカルデータエリア

ローカルとして定義された変数は、単一の Natural プログラミングオブジェクト内でのみ使用できます。 ローカルデータを定義する方法は次の 2 つです。

- プログラム内部に定義します。
- プログラム外部の別個の Natural プログラミングオブジェクトに定義します。

ローカルデータエリアは、そのローカルデータエリアを使用するプログラム、サブプログラム、または外部サブルーチンが実行を開始するときに初期化されます。

最初の例では、フィールドはプログラムの DEFINE DATA ステートメント内に直接定義されています。 2 つ目の例では、同じフィールドが LDA に定義され、DEFINE DATA ステートメントにはそのデータエリアへの参照のみが指定されています。

例 **1**：**DEFINE DATA** ステートメント内に直接定義されたフィールド

```
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 PERSONNEL-ID
1 #VARI-A (A20)
1 #VARI-B (N3.2)
1 #VARI-C (I4)
END-DEFINE
...
```

例 **2**：別のデータエリアに定義されたフィールド

プログラム：

```
DEFINE DATA LOCAL
       USING LDA39
END-DEFINE
  ...
```

ローカルデータエリア LDA39：

```
I T L  Name                            F Length     Miscellaneous
All -- --------------------------------- - ---------- -------------------------->
  V  1 VIEWEMP                                       EMPLOYEES
     2 PERSONNEL-ID                    A          8
     2 FIRST-NAME                      A         20
     2 NAME                            A         20
     1 #VARI-A                         A         20
     1 #VARI-B                         N        3.2
     1 #VARI-C                         I          4
```

アプリケーション構造を明確にして管理しやすくするために、通常はプログラム外部のデータエリアにフィールドを定義する方が便利です。

グローバルデータエリア

以下では次のトピックについて説明します。

- GDA の作成および参照
- GDA インスタンスの作成および削除

- データブロック

GDA の作成および参照

GDA は、Natural データエリアエディタで作成および変更します。 詳細については、『エディタ』ドキュメントの「データエリアエディタ」を参照してください。

Natural プログラミングオブジェクトで参照される GDA は、この GDA を参照するオブジェクトが保存されているものと同じ Natural ライブラリ、またはこのライブラリに定義されている steplib に保存する必要があります。

COMMON という名前の GDA がライブラリに存在する場合は、このライブラリに LOGON すると、ACOMMON という名前のプログラムが自動的に呼び出されます。

Important: 複数の Natural プログラミングオブジェクトが 1 つの GDA を参照するアプリケーションを構築するときは、この GDA 内のデータ要素定義を変更すると、このデータエリアを参照するすべての Natural プログラミングオブジェクトに影響が及ぶことに注意してください。 したがって、このようなオブジェクトは、GDA を変更した後に CATALOG または STOW コマンドを使って再コンパイルする必要があります。

GDA を使用するには、Natural プログラミングオブジェクトで DEFINE DATA ステートメントの GLOBAL 節を使用して GDA を参照する必要があります。 それぞれの Natural プログラミングオブジェクトは 1 つのグローバルデータエリアのみを参照できます。つまり、DEFINE DATA ステートメントに複数の GLOBAL 節を含めることはできません。

GDA インスタンスの作成および削除

GDA の最初のインスタンスは、これを参照する最初の Natural プログラミングオブジェクトが実行を開始したランタイムに作成され、初期化されます。

GDA インスタンスが作成されると、この中に含まれるデータ値は、この GDA を参照する（DEFINE DATA GLOBAL ステートメント）すべての Natural プログラミングオブジェクト、および PERFORM、INPUT、または FETCH ステートメントによって呼び出されるすべての Natural プログラミングオブジェクトで共有できます。 GDA インスタンスを共有するすべてのオブジェクトは、同じデータ要素上で機能します。

次の状況が該当する場合に、新しい GDA インスタンスが作成されます。

- いずれかの GDA を参照するサブプログラムが CALLNAT ステートメントで呼び出された。
- GDA を参照しないサブプログラムが、いずれかの GDA を参照するプログラミングオブジェクトを呼び出した。

GDA の新しいインスタンスが作成されると、現在の GDA インスタンスは中断され、含まれていたデータ値はスタックされます。 次にサブプログラムが、新しく作成された GDA インスタンス内のデータ値を参照します。 中断された GDA インスタンス内のデータ値にアクセスすることはできません。 プログラミングオブジェクトは 1 つの GDA インスタンスしか参照できず、

以前の GDA インスタンスにアクセスすることはできません。 GDA データ要素は、CALLNAT ステートメント内でパラメータとして定義することによって、サブプログラムに渡すことのみが可能です。

サブプログラムが、呼び出し元プログラミングオブジェクトに戻ると、参照されていたGDAインスタンスは削除され、それまで中断していたGDAインスタンスがそのデータ値とともに再開されます。

次のいずれかが該当する場合、GDA インスタンスとその内容は削除されます。

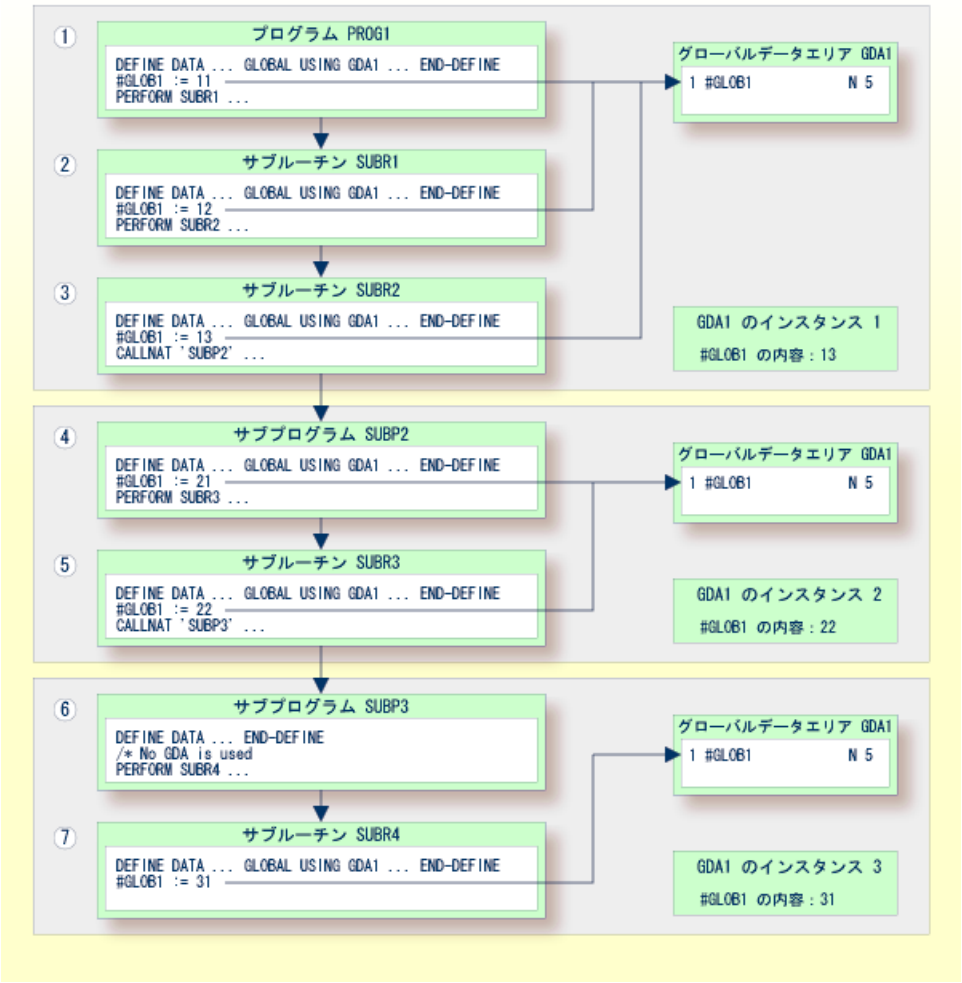
- 次の LOGON が実行された。
- 同じレベルで別の GDA が参照された（レベルについてはこのセクションで後述）。
- RELEASE VARIABLES ステートメントが実行された。 この場合、GDA インスタンス内の変数は、レベル 1 のプログラムの実行が終了するか、プログラムが FETCH または RUN ステートメントで別のプログラムを呼び出したときにリセットされます。

次のセクションの図は、プログラミングオブジェクトが GDA を参照し、GDA インスタンス内のデータ要素を共有する様子を示しています。

GDA インスタンスの共有

次の図は、GDA を参照するサブプログラムが、呼び出し元プログラムが参照する GDA インスタンス内のデータ値を共有できないことを示しています。 呼び出し元プログラムと同じ GDA を参照するサブプログラムは、この GDA の新しいインスタンスを作成します。 ただし、サブプログラムが参照するGDAに定義されているデータ要素は、このサブプログラムが呼び出したサブルーチンやヘルプルーチンで共有できます。

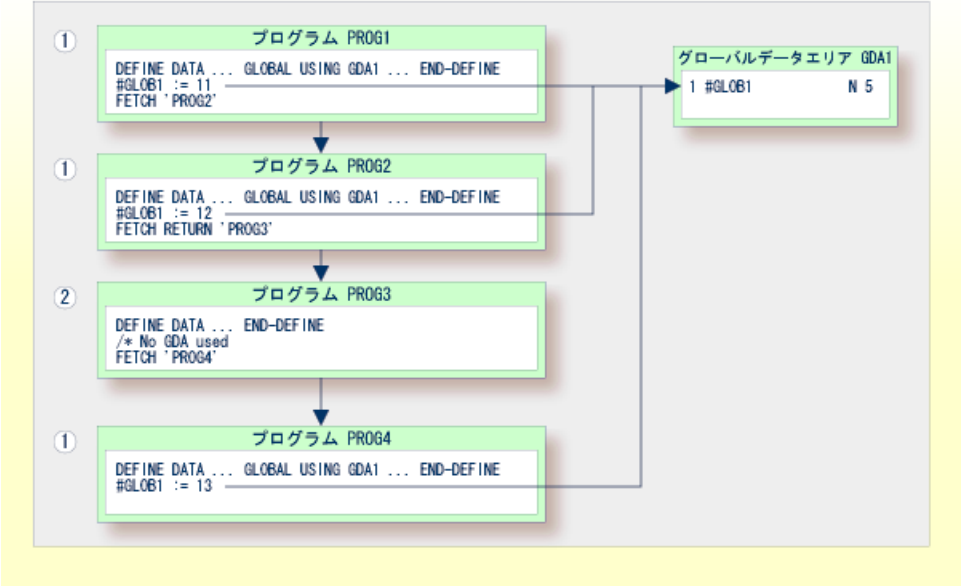
次の図は、GDA1 の 3 つの GDA インスタンスと、データ要素 #GLOB1 によって各 GDA インスタンスに割り当てられる最終値を示しています。 番号 ① ～ ⑦ は、プログラミングオブジェクトの階層レベルを示しています。



FETCH または **FETCH RETURN** の使用

次の図は、同じ GDA を参照し、FETCH または FETCH RETURN ステートメントを使用してそれぞれを呼び出す複数のプログラムが、このGDA内に定義されているデータ要素を共有することを示しています。 これらのプログラムのいずれかが GDA を参照しない場合は、以前に参照されていた GDA のインスタンスがアクティブなまま残り、データ要素の値が保持されます。

番号 ① および ② は、プログラミングオブジェクトの階層レベルを示しています。



異なる **GDA** での **FETCH** の使用

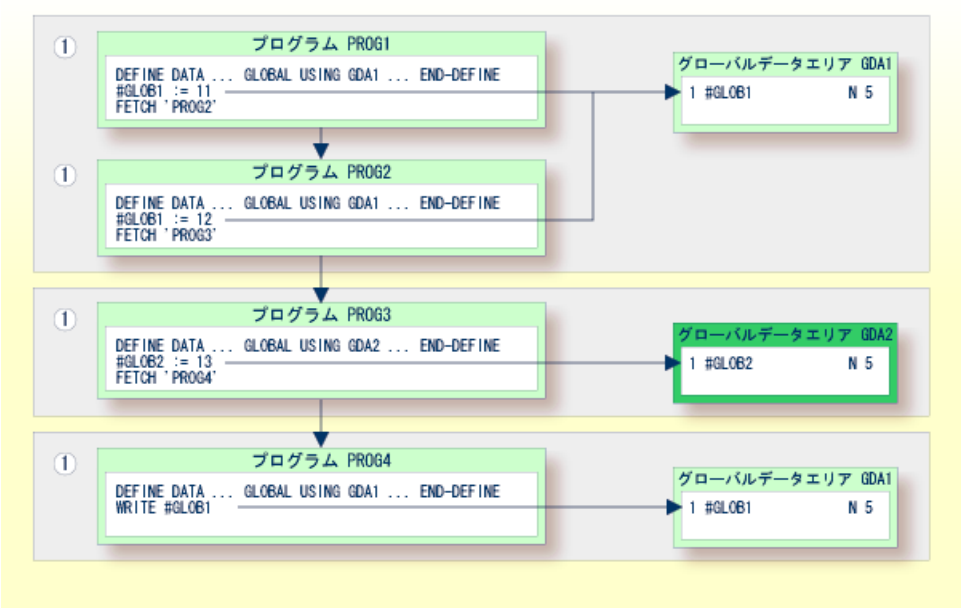
次の図は、プログラムが FETCH ステートメントを使用して、異なる GDA を参照する別のプログラムを呼び出すと、呼び出し元プログラムが参照する GDA（ここでは GDA1）の現在のインスタンスが削除されることを示しています。 その後、この GDA が他のプログラムから再び参照されると、この GDA の新しいインスタンスが作成され、その中のすべてのデータ要素にはそれぞれの初期値が指定されます。

FETCH RETURN ステートメントを使用して、異なる GDA を参照する別のプログラムを呼び出すことはできません。

番号 (1) は、プログラミングオブジェクトの階層レベルを示しています。

呼び出し元プログラム PROG3 および PROG4 によって、GDA インスタンスは次のような影響を受けます。

- PROG3 内のステートメント GLOBAL USING GDA2 によって、GDA2 のインスタンスが作成され、GDA1 の現在のインスタンスが削除されます。
- PROG4 内のステートメント GLOBAL USING GDA1 によって、GDA2 の現在のインスタンスが削除され、GDA1 の新しいインスタンスが作成されます。 この結果、WRITE ステートメントで値ゼロ（0）が表示されます。



データブロック

データストレージスペースを節約するために、データブロックが含まれる **GDA** を作成できます。

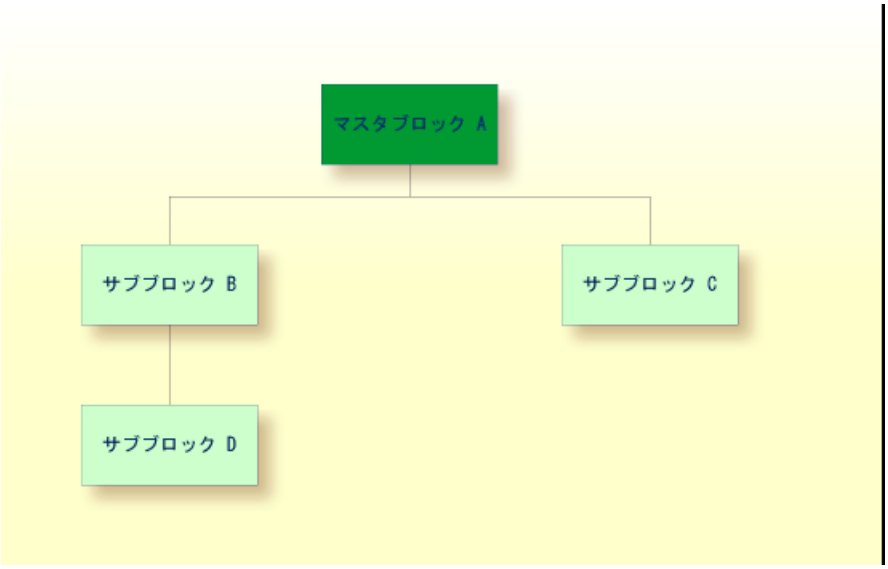
以下では次のトピックについて説明します。

- データブロックの使用例
- データブロックの定義
- ブロック階層

データブロックの使用例

データブロックは、プログラムの実行中に相互に重ねることができるため、ストレージスペースの節約になります。

例えば、次のような階層では、ブロック B とブロック C が同じストレージエリアに割り当てられます。 したがって、ブロック B とブロック C は同時に使用できません。 ブロック B を変更すると、ブロック C の内容が破壊されます。



データブロックの定義

データブロックはデータエリアエディタで定義します。 どのブロックがどのブロックの下位になるかを指定して、ブロック階層を設定します。これは、ブロック定義のコメントフィールドに「親」 ブロックの名前を入力して設定します。

次の例では、SUB-BLOCKB と SUB-BLOCKC が MASTER-BLOCKA の下位になり、SUB-BLOCKD が SUB-BLOCKB の下位になります。

ブロックレベルの最大値は 8（マスタブロックを含む）です。

例：

グローバルデータエリア G-BLOCK

```
I  T  L Name                            F Leng Index/Init/EM/Name/Comment
-  -  - --------------------------------- - ---- ---------------------------------
   B    MASTER-BLOCKA
     1  MB-DATA01                        A   10
   B    SUB-BLOCKB                           MASTER-BLOCKA
     1  SBB-DATA01                       A   20
   B    SUB-BLOCKC                           MASTER-BLOCKA
     1  SBC-DATA01                       A   40
   B    SUB-BLOCKD                           SUB-BLOCKB
     1  SBD-DATA01                       A   40
```

特定のブロックをプログラムで使用可能にするには、`DEFINE DATA` ステートメント内で次の構文を使用します。

プログラム 1：

```
DEFINE DATA GLOBAL
       USING G-BLOCK
       WITH MASTER-BLOCKA
END-DEFINE
```

プログラム 2：

```
DEFINE DATA GLOBAL
       USING G-BLOCK
       WITH MASTER-BLOCKA.SUB-BLOCKB
END-DEFINE
```

プログラム 3：

```
DEFINE DATA GLOBAL
       USING G-BLOCK
       WITH MASTER-BLOCKA.SUB-BLOCKC
END-DEFINE
```

プログラム 4：

```
DEFINE DATA GLOBAL
       USING G-BLOCK
       WITH MASTER-BLOCKA.SUB-BLOCKB.SUB-BLOCKD
END-DEFINE
```

この構造では、プログラム 1 がプログラム 2、プログラム 3、またはプログラム 4 と MASTER-BLOCKA 内のデータを共有できます。 ただし、プログラム 2 とプログラム 3 は SUB-BLOCKB と SUB-BLOCKC のデータエリアを共有できません。これらのデータブロックは構造と同じレベルにあるため、同じストレージエリアを占有しているからです。

ブロック階層

データブロック階層を使用するときは注意が必要です。3つのプログラムがデータブロック階層を使用する次のシナリオを想定します。

プログラム 1：

```
DEFINE DATA GLOBAL
       USING G-BLOCK
       WITH MASTER-BLOCKA.SUB-BLOCKB
END-DEFINE
*
MOVE 1234 TO SBB-DATA01
FETCH 'PROGRAM2'
END
```

プログラム 2：

```
DEFINE DATA GLOBAL
       USING G-BLOCK
       WITH MASTER-BLOCKA
END-DEFINE
*
FETCH 'PROGRAM3'
END
```

プログラム 3：

```
DEFINE DATA GLOBAL
       USING G-BLOCK
       WITH MASTER-BLOCKA.SUB-BLOCKB
END-DEFINE
*
WRITE SBB-DATA01
END
```

説明：

- プログラム 1 は、グローバルデータエリア G-BLOCK を MASTER-BLOCKA および SUB-BLOCKB と共同で使用します。このプログラムは SUB-BLOCKB 内のフィールドを変更して、プログラム 2 を FETCH しますが、プログラム 2 のデータ定義には MASTER-BLOCKA しか指定されていません。
- プログラム 2 は SUB-BLOCKB をリセット、つまりその内容を削除します。これは、レベル 1 のプログラム（例えば、FETCH ステートメントで呼び出されたプログラム）は、そのデータ定義に定義されているブロックの下位となるすべてのデータブロックをリセットするからです。

- 次にプログラム 2 はプログラム 3 を FETCH します。プログラム 3 はプログラム 1 で変更されたフィールドを表示するものですが、空の画面を返します。

プログラムレベルの詳細については、「*呼び出されるオブジェクトの複数レベル*」を参照してください。

パラメータデータエリア

サブプログラムは、CALLNAT ステートメントを使用して呼び出されます。CALLNAT ステートメントを使用して、パラメータを呼び出し元オブジェクトからサブプログラムに渡すことができます。

このようなパラメータは、サブプログラム内の DEFINE DATA PARAMETER ステートメントに、次の方法で定義する必要があります。

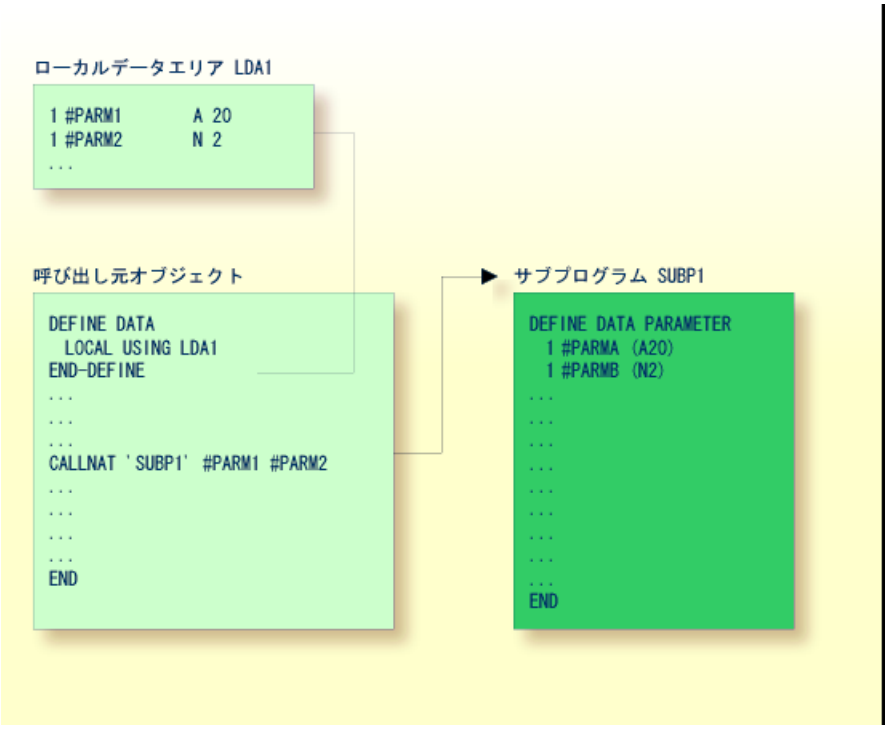
- DEFINE DATA ステートメントの PARAMETER 節内に直接定義します。
- 別のパラメータデータエリア（PDA）に定義して、その PDA を参照する DEFINE DATA PARAMETER ステートメントを指定します。

以下では次のトピックについて説明します。

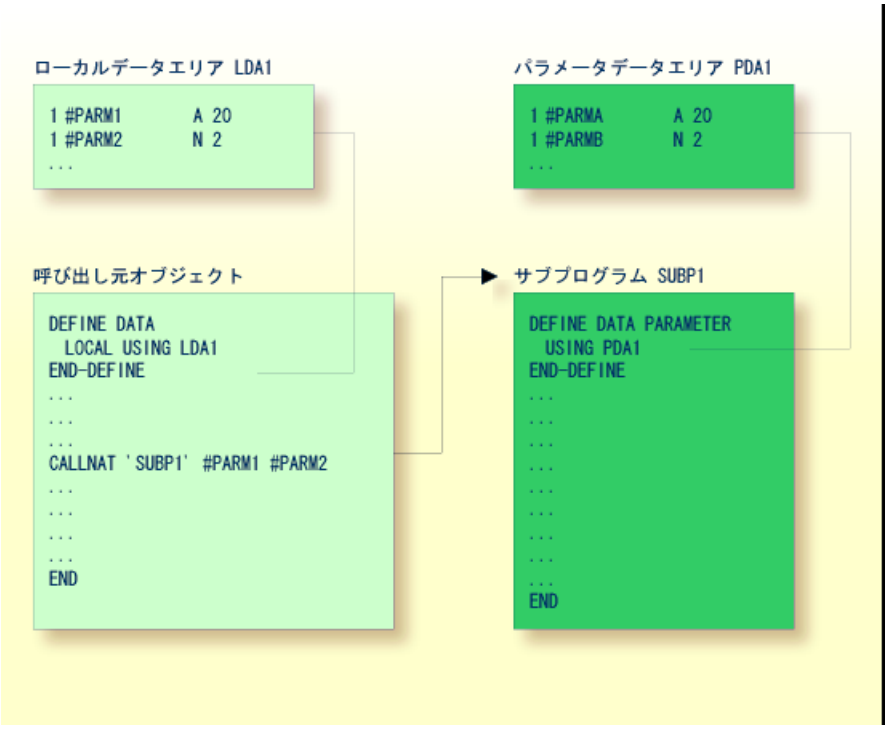
- **DEFINE DATA PARAMETER ステートメント内に定義されたパラメータ**

- パラメータデータエリアに定義されたパラメータ

DEFINE DATA PARAMETER ステートメント内に定義されたパラメータ



パラメータデータエリアに定義されたパラメータ



同様に、PERFORM ステートメントによって外部サブルーチンに渡されるパラメータは、外部サブルーチン内の DEFINE DATA PARAMETER ステートメントを使用して定義する必要があります。

呼び出し元オブジェクトでは、サブプログラム／サブルーチンに渡されるパラメータ変数をPDA内に定義する必要はありません。上記の図では、呼び出し元オブジェクトが使用する LDA に定義されていますが、GDA に定義することも可能です。

呼び出し元オブジェクト内の CALLNAT/PERFORM ステートメントで指定されるパラメータの順序、フォーマット、および長さは、呼び出されるサブプログラム／サブルーチンの DEFINE DATA PARAMETER ステートメントに指定されるフィールドの順序、フォーマット、および長さと正確に一致する必要があります。 ただし、呼び出し元オブジェクト内の変数の名前は、呼び出されるサブプログラム／サブルーチンと同じである必要はありません。パラメータは名前ではなくアドレスで転送されるためです。

呼び出し元プログラムで使用されるデータ要素定義が、サブプログラムまたは外部サブルーチンで使用されるデータ要素定義と同一であることを確実にするために、DEFINE DATA LOCAL USING ステートメントに PDA を指定できます。 PDA を LDA として使用することにより、PDA と同じ構造を持つ LDA を作成する手間を省くことができます。

6 プログラム、関数、サブプログラム、およびサブルーチン

このドキュメントでは、ルーチン、つまり下位プログラムとして呼び出すことができるオブジェクトタイプについて説明します。

ヘルプルーチンとマップは他のオブジェクトからも呼び出されますが、厳密にはルーチンではないため、個別のドキュメントで説明しています。「ヘルプルーチン」および「マップ」を参照してください。

このchapterでは、次のトピックについて説明します。

モジュラーアプリケーション構造

一般に、Naturalアプリケーションは単一の巨大なプログラムで構成されるのではなく、複数のモジュールに分割されます。 これらの各モジュールは、管理可能なサイズの機能ユニットであり、各モジュールは明確に定義された方法でアプリケーションの他のモジュールに接続されています。 これにより、適切に構造化されたアプリケーションが提供されるため、開発およびその後のメンテナンスが大幅に容易かつ迅速に行うことができるようになります。

メインプログラムの実行中には、他のプログラム、サブプログラム、サブルーチン、ヘルプルーチン、およびマップを呼び出すことができます。 続いてこれらのオブジェクトが他のオブジェクトを呼び出すことができます。例えば、サブルーチンが別のサブルーチンを呼び出すこともできます。 したがって、アプリケーションのモジュラー構造は非常に複雑になり、複数のレベルに拡張される可能性があります。

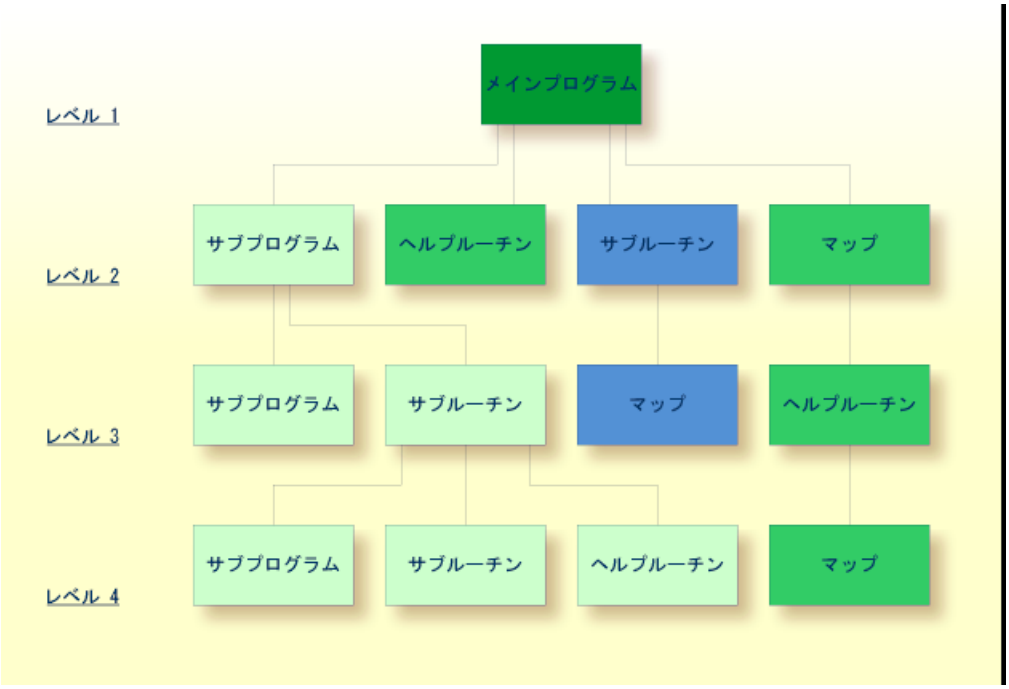
呼び出されるオブジェクトの複数レベル

呼び出される各オブジェクトは、それを呼び出したオブジェクトのレベルの1つ下のレベルになります。つまり、下位オブジェクトを呼び出すたびに、レベル番号は 1 つ増加します。

直接実行されるプログラムはすべてレベル1です。メインプログラムによって直接呼び出されるサブプログラム、サブルーチン、マップ、ヘルプルーチンはレベル2になります。このサブルーチンがサブルーチンを呼び出すと、呼び出されたサブルーチンはレベル 3 になります。

別のオブジェクト内から `FETCH` ステートメントで呼び出されたプログラムは、メインプログラムとして分類され、レベル 1 から動作します。 ただし、`FETCH RETURN` で呼び出されたプログラムは、下位プログラムとして分類され、呼び出し元オブジェクトの1つ下のレベルが割り当てられます。

次の図は、呼び出されるオブジェクトの複数レベルの例と、これらのレベルがどのように数えられるかを示しています。



現在実行中のオブジェクトのレベル番号を確認する場合は、システム変数 *LEVEL を使用できます。詳細については『システム変数』ドキュメントを参照してください。

このドキュメントでは、ルーチン、つまり下位プログラムとして呼び出すことができる次の Natural オブジェクトタイプについて説明します。

- プログラム
- ファンクション
- サブルーチン
- サブプログラム

ヘルプルーチンとマップは他のオブジェクトからも呼び出されますが、厳密にはルーチンではないため、個別のドキュメントで説明しています。「ヘルプルーチン」および「マップ」を参照してください。

基本的に、プログラム、サブプログラムおよびサブルーチンは、データの受け渡し方法やお互いのデータエリアを共有できるかどうかなどがそれぞれ異なります。 したがって、どのオブジェクトタイプをどの目的に使用するかの決定は、アプリケーションのデータ構造に大きく依存します。

プログラム

プログラムは単独で実行し、テストできます。

- ソースプログラムをコンパイルして実行するには、システムコマンド RUN を使用します。
- すでにコンパイルされた形式で存在するプログラムを実行するには、システムコマンド EXECUTE を使用します。

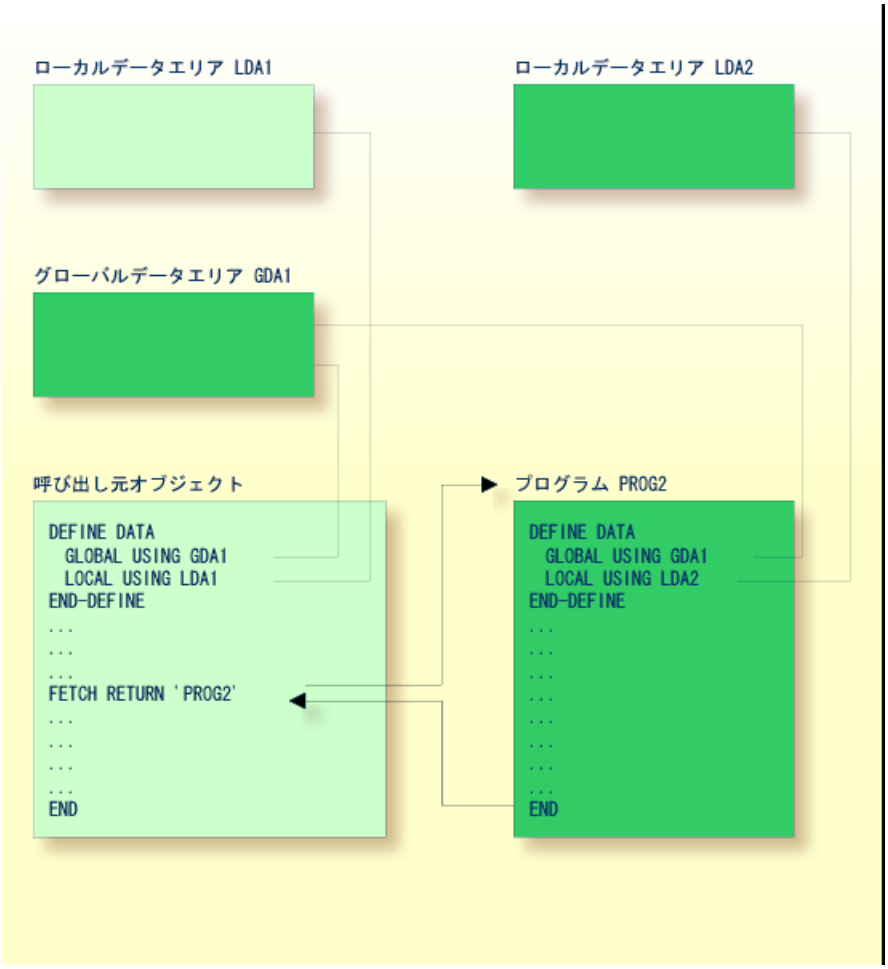
プログラムは、他のオブジェクトから FETCH または FETCH RETURN ステートメントで呼び出すこともできます。 呼び出し元オブジェクトは、別のプログラム、サブプログラム、ファンクション、サブルーチン、またはヘルプルーチンのいずれでもかまいません。

- プログラムが FETCH RETURN で呼び出されると、呼び出し元オブジェクトの実行は終了されるのではなく中断され、FETCH されたプログラムは *下位プログラム*としてアクティブになります。 FETCH されたプログラムの実行が終了すると、呼び出し元オブジェクトが再びアクティブになり、その実行は FETCH RETURN ステートメントの次のステートメントから継続します。
- プログラムが FETCH で呼び出されると、呼び出し元オブジェクトの実行は終了し、FETCH されたプログラムがメインプログラムとしてアクティブになります。呼び出し元オブジェクトが、FETCH されたプログラムの終了時に再びアクティブになることはありません。

以下では次のトピックについて説明します。

- FETCH で呼び出されるプログラム

FETCH RETURN で呼び出されるプログラム

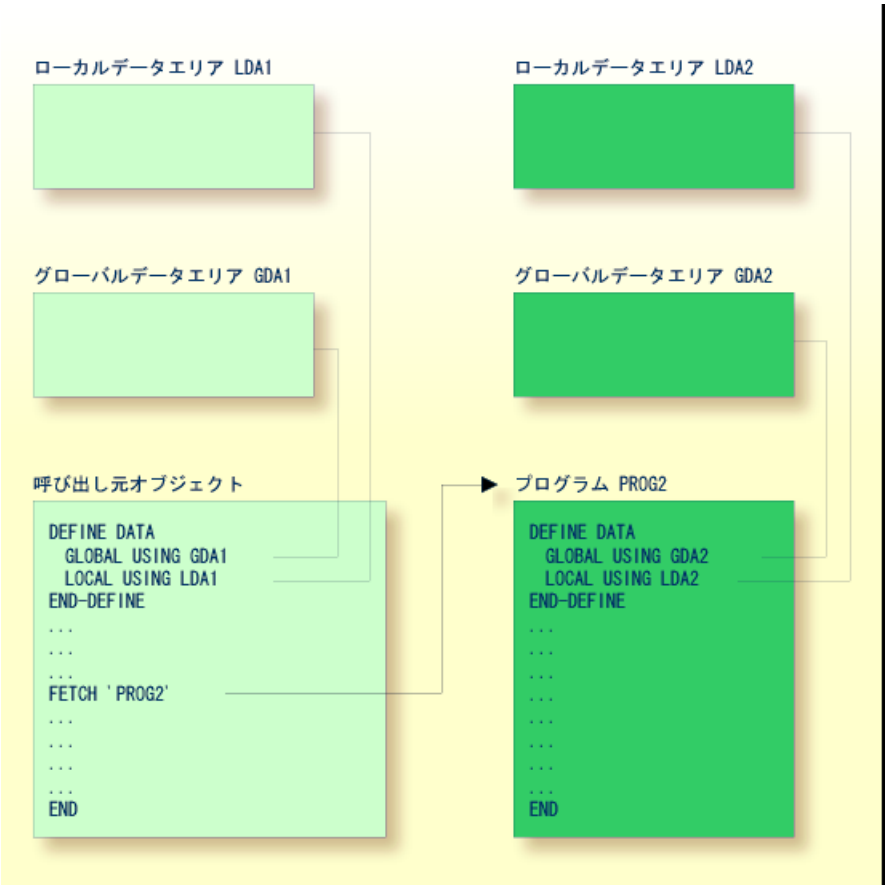


FETCH RETURN で呼び出されるプログラムは、呼び出し元オブジェクトが使用するグローバルデータエリアにアクセスできます。

また、あらゆるプログラムには独自のローカルデータエリアがあり、その中にはそのプログラム内だけで使用されるフィールドが定義されます。

ただし、FETCH RETURN で呼び出されたプログラムが独自のグローバルデータエリアを持つことはできません。

FETCH で呼び出されるプログラム



メインプログラムとして FETCH で呼び出されるプログラムは、上記の図に示したように、通常は独自のグローバルデータエリアを設定します。 ただし、呼び出し元オブジェクトが設定したものと同じグローバルデータエリアを使用することもできます。

Note: ソースプログラムは、RUN ステートメントで呼び出すこともできます。『ステートメント』ドキュメントの RUN ステートメントを参照してください。

ファンクション

「ファンクション」 タイプのオブジェクトには、単一のファンクションの定義が含まれ、次のコーディング例で示されるような構造が可能です。

```
DEFINE FUNCTION
   ...
   DEFINE SUBROUTINE
   ...
   END-SUBROUTINE
   ...
END-FUNCTION
```

DEFINE FUNCTION と END-FUNCTION の間のステートメントブロックには、ファンクションが呼び出されたときに実行されるすべてのステートメントを含める必要があります。

ファンクション定義内に内部サブルーチンを定義することは可能です。

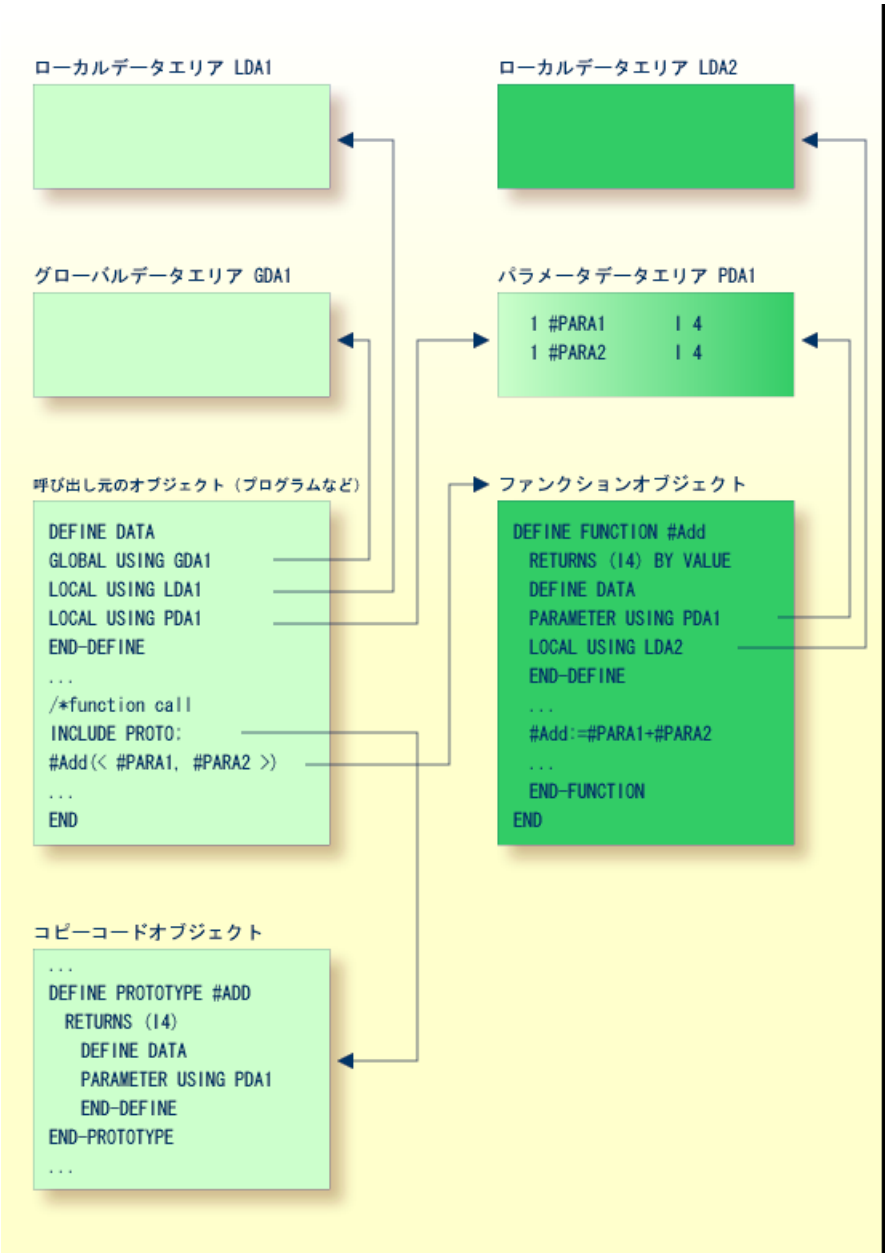
ファンクションはファンクションコール構文を使用して呼び出されます。

オブジェクト内で繰り返し実行されるコードブロックがある場合は、インラインサブルーチンを使用すると便利です。 次に、DEFINE SUBROUTINE ステートメントブロック内でこのブロックを 1 度だけコードし、いくつかの PERFORM ステートメントで呼び出すだけです。

呼び出し元オブジェクトのグローバルデータエリア（例：GDA1）は、ファンクション定義内では参照できません。また、ファンクションを入力すると、ランタイム環境によって新しいグローバルデータエリアが作成されるため、ファンクションによって呼び出されるオブジェクトは、このファンクションを呼び出すオブジェクトのグローバルデータエリア（GDA1）を参照できません。

パラメータデータエリア（例：PDA1）は、パラメータ変更時の管理工数を最小限にするために、ファンクションコールおよびファンクション定義のパラメータにアクセスするために使用できます。

プロトタイプ定義を含むコピーコードオブジェクトは、必要に応じて、ファンクションコール参照の戻り変数のタイプを決定してパラメータをチェックするのみの目的で、コンパイル時に使用されます。



サブルーチン

サブルーチンを構成するステートメントは、DEFINE SUBROUTINE ... END-SUBROUTINE ステートメントブロック内に定義する必要があります。

サブルーチンは PERFORM ステートメントによって呼び出されます。

サブルーチンには、*インラインサブルーチン*と*外部サブルーチン*があります。

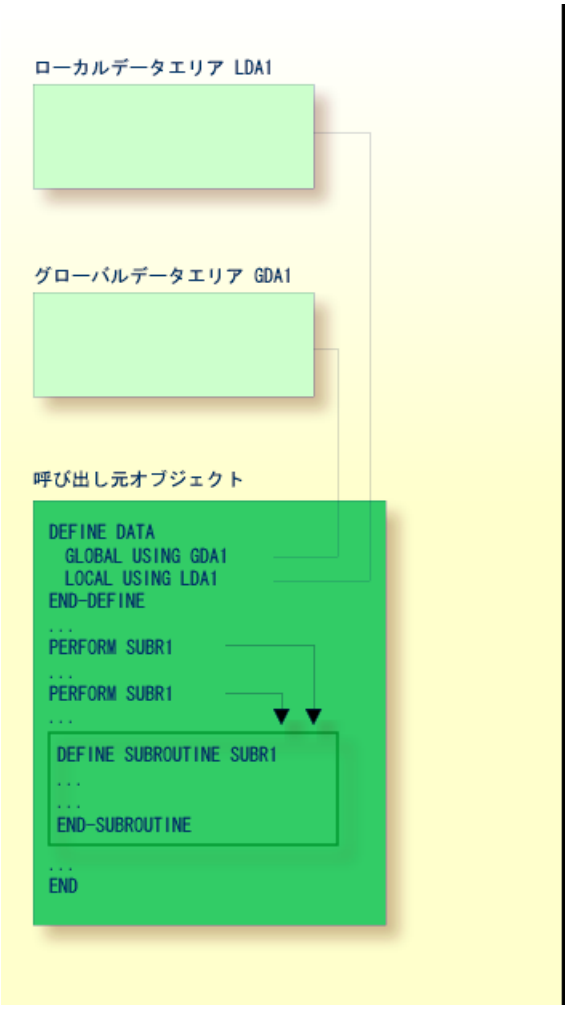
- インラインサブルーチン
  インラインサブルーチンは、このサブルーチンを呼び出す PERFORM ステートメントが含まれるオブジェクト内に定義されるものです。
- 外部サブルーチン
  外部サブルーチンは、このサブルーチンを呼び出すオブジェクトの外部にある別のサブルーチンタイプのオブジェクト内に定義されるものです。

1つのオブジェクト内で繰り返し実行されるコードブロックがある場合は、インラインサブルーチンを使用すると便利です。次に、DEFINE SUBROUTINE ステートメントブロック内でこのブロックを 1 度だけコードし、いくつかの PERFORM ステートメントで呼び出すだけです。

以下では次のトピックについて説明します。

- 外部サブルーチンで使用できるデータ

インラインサブルーチン



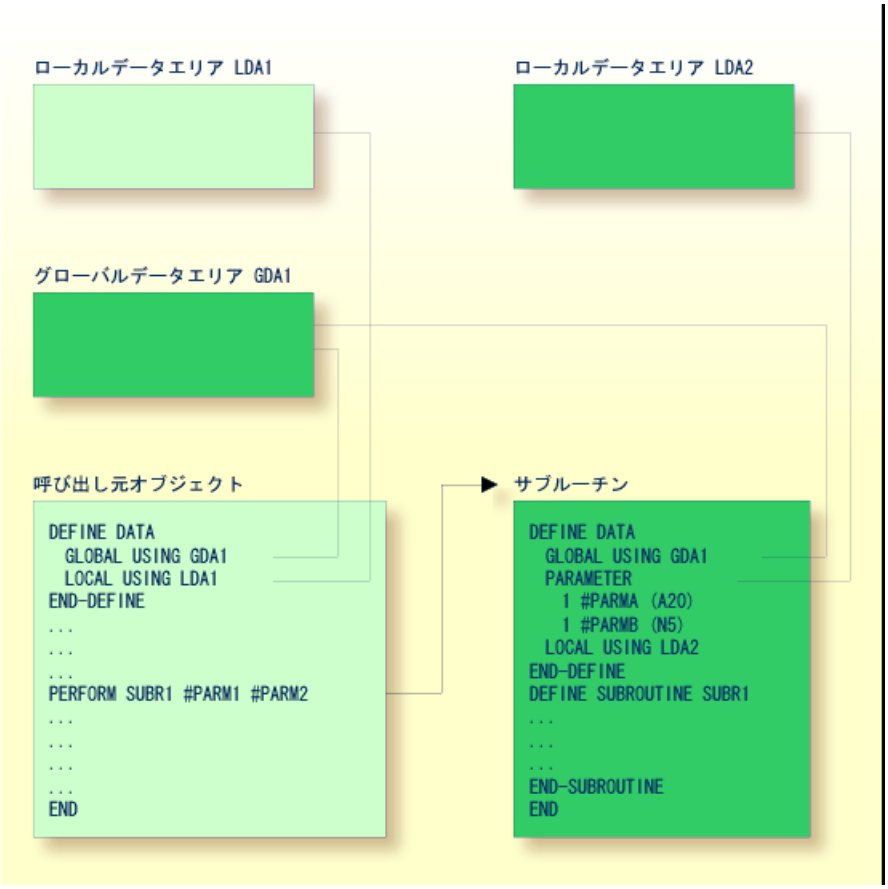
インラインサブルーチンは、タイプがプログラム、ファンクション、サブプログラム、サブルーチン、またはヘルプルーチンのプログラミングオブジェクト内に含めることができます。

インラインサブルーチンが非常に大きく、それを含むオブジェクトの読みやすさが損なわれる場合は、アプリケーションの読みやすさが改善されるように、外部サブルーチンに置き換えることを検討できます。

インラインサブルーチンで使用できるデータ

インラインサブルーチンは、ローカルデータエリアと、このサブルーチンを含むオブジェクトが使用するグローバルデータエリアにアクセスできます。

外部サブルーチン



外部サブルーチン、つまりサブルーチンタイプのオブジェクトは単独では実行できません。 これは、他のオブジェクトから呼び出す必要があります。 呼び出し元オブジェクトは、プログラム、関数、サブプログラム、サブルーチン、またはヘルプルーチンのいずれでもかまいません。

外部サブルーチンで使用できるデータ

外部サブルーチンは、呼び出し元オブジェクトが使用するグローバルデータエリアにアクセスできます。

また、PERFORMステートメントを使用して、呼び出し元オブジェクトから外部サブルーチンにパラメータを渡すことができます。これらのパラメータは、サブルーチンのDEFINE DATA PARAMETERステートメント、またはサブルーチンが使用するパラメータデータエリアに定義する必要があります。

また、外部サブルーチンは、そのサブルーチン内のみで使用するフィールドを定義するローカルデータエリアを持つことができます。

ただし、外部サブルーチンが独自のグローバルデータエリアを持つことはできません。

サブプログラム

通常、サブプログラムには、アプリケーション内のさまざまなオブジェクトに使用される一般に使用可能な標準ファンクションが含まれています。

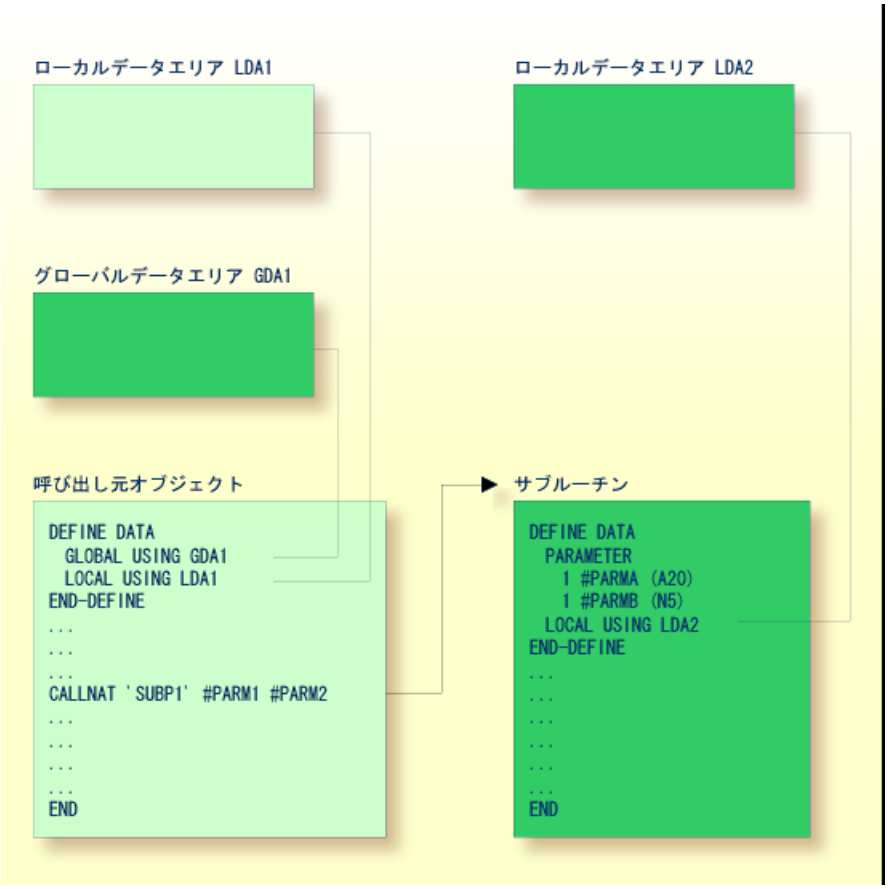
サブプログラムは単独では実行できません。これは、他のオブジェクトから呼び出す必要があります。呼び出し元オブジェクトは、プログラム、関数、サブプログラム、サブルーチン、またはヘルプルーチンのいずれでもかまいません。

サブプログラムは、CALLNATステートメントを使用して呼び出されます。

CALLNATステートメントが実行されると、呼び出し元オブジェクトの実行は中断され、サブプログラムが実行されます。このサブプログラムの実行後は、呼び出し元オブジェクトの実行がCALLNATステートメントの次のステートメントから続行されます。

サブプログラムで使用可能なデータ

CALLNATステートメントを使用して、パラメータを呼び出し元オブジェクトからサブプログラムに渡すことができます。これらのパラメータは、呼び出し元オブジェクトからサブプログラムが使用できる唯一のデータです。これらのパラメータは、サブプログラムのDEFINE DATA PARAMETERステートメント、またはサブプログラムが使用するパラメータデータエリアに定義する必要があります。



また、サブプログラムは、その中で使用するフィールドを定義する独自のローカルデータエリアを持つことができます。

サブプログラムがサブルーチンやヘルプルーチンを呼び出す場合は、そのようなサブルーチンやヘルプルーチンと共有する独自のグローバルデータエリアを設定することもできます。

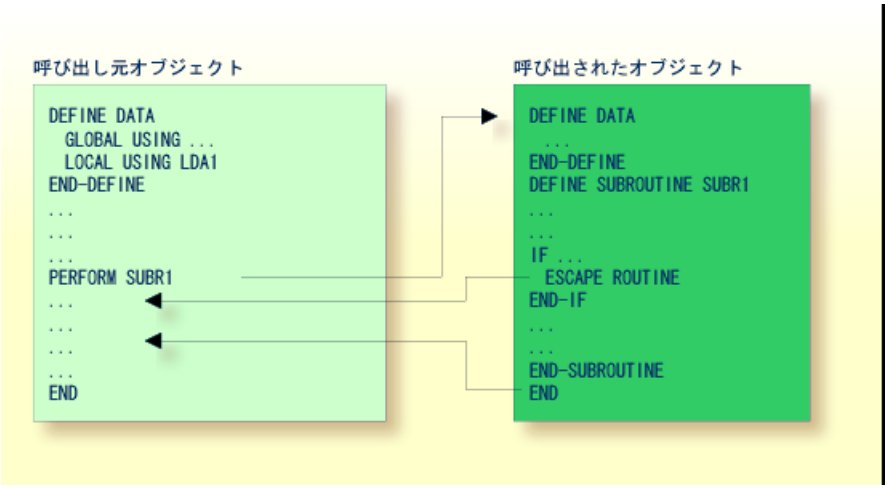
ルーチンを呼び出すときの処理フロー

ルーチン、つまりサブプログラム、外部サブルーチンまたはプログラムをそれぞれ呼び出す CALLNAT、PERFORM、または FETCH RETURN ステートメントが実行されると、呼び出し元オブジェクトの実行は中断され、該当するルーチンの実行が始まります。

ルーチンの実行は、END ステートメントに到達するまで、または ESCAPE ROUTINE ステートメントの実行によってルーチンの処理が停止されるまで継続します。

いずれの場合も、呼び出し元オブジェクトの処理は、ルーチンを呼び出すために使用された CALLNAT、PERFORM または FETCH RETURN ステートメントの次のステートメントから続行します。

例：


呼び出し元オブジェクト
DEFINE DATA
GLOBAL USING ...
LOCAL USING LDA1
END-DEFINE
...
...
...
PERFORM SUBR1
...
...
...
...
END
呼び出されたオブジェクト
DEFINE DATA
...
END-DEFINE
DEFINE SUBROUTINE SUBR1
...
...
IF ...
ESCAPE ROUTINE
END-IF
...
...
END-SUBROUTINE
END

7 リッチ GUI ページの処理 - アダプタ

アダプタタイプの Natural オブジェクトは、Natural アプリケーション内のリッチ GUI ページを表現するために使用します。 このオブジェクトタイプは、マップタイプのオブジェクトが端末 I/O 処理に対して行う役割と同様の役割を、リッチ GUI ページの処理に対して行います。 ただし、レイアウト情報を含まない点でマップタイプと異なります。

アダプタタイプのオブジェクトは、外部ページレイアウトから生成されます。 外部で定義および保存されたページレイアウトを使用して、Natural アプリケーションが表示や修正のために外部 I/O システムにデータを送信できるようにするインターフェイスとして機能します。 アダプタには、このタスクを実行するために必要な Natural コードが含まれています。

アプリケーションプログラムは、`PROCESS PAGE USING` ステートメント内でアダプタを参照します。

オブジェクトタイプアダプタの詳細については、『*Natural for Ajax*』ドキュメントを参照してください。

8 マップ

INPUT ステートメントは、ダイナミックな画面レイアウトの指定の代わりとして、Natural オブジェクトタイプ「マップ」を利用する定義済みマップレイアウトの機能を提供します。

このchapterでは、次のトピックについて説明します。

マップ使用の利点

ダイナミック画面レイアウトの指定ではなく定義済みマップレイアウトを使用すると、次のようなさまざまな利点があります。

- プログラムロジックと表示ロジックが結果的に分割されるため、アプリケーションが明確に構造化されます。
- メインプログラムに変更を行うことなくマップレイアウト修正が可能です。
- アプリケーションのユーザーインターフェイスの言語を国際化や地域化に容易に適合させることができます。

マップのようなプログラミングオブジェクトを使用する利点は、既存のNaturalアプリケーションを管理する場合に明白です。

マップのタイプ

マップ（画面レイアウト）は、ユーザーが画面で見ることができるアプリケーションの一部です。

マップには次のタイプがあります。

- 入力マップ
  ユーザーとの対話は入力マップを経由して行われます。
- 出力マップ
  アプリケーションによって出力レポートが生成される場合に、出力マップを使用してこのレポートを画面に表示できます。
- ヘルプマップ
  ヘルプマップは原則的には他のマップに似ていますが、ヘルプとして割り当てられる場合は、ヘルプの目的に使用できることを確実にするために追加チェックが実行されます。

「マップ」オブジェクトタイプは、次の要素で構成されます。

- 画面レイアウトを定義するマップ本文。
- 関連付けられたパラメータデータエリア（PDA）。一種のインターフェイスとして、特定のマップに表示される各フィールドの名前、フォーマット、長さなどのデータ定義が含まれます。

関連トピック：

- 入力フィールドに付加できる選択ボックスの詳細については、『ステートメント』ドキュメントの「INPUT」の「*SB - 選択ボックス*」および『*パラメータリファレンス*』の「*SB - 選択ボックス*」を参照してください。
- 上部を出力マップとして使用し、下部を入力マップとして使用できる画面分割マップの詳細については、『ステートメント』ドキュメントの「INPUT」で「*画面分割機能*」を参照してください。

マップの作成

マップおよびヘルプマップのレイアウトは、マップエディタで作成および編集します。 適切なLDA は、データエリアエディタで作成および管理します。

Naturalがインストールされているプラットフォームに応じて、これらのエディタは、キャラクタユーザーインターフェイスまたはグラフィカルユーザーインターフェイスのいずれかを備えています。

関連トピック：

- データエリアエディタの使用の詳細については、プラットフォーム固有の『*エディタ*』ドキュメントの「*データエリアエディタ*」を参照してください。
- マップエディタの使用の詳細については、プラットフォーム固有の『*エディタ*』ドキュメントの「*マップエディタ*」を参照してください。
- ダイナミックに指定された画面レイアウトを使用する入力処理の詳細については、『ステートメント』ドキュメントの「INPUT *構文1 - ダイナミック画面レイアウトの指定*」を参照してください。
- マップエディタで作成したマップレイアウトを使用する入力処理の詳細については、『ステートメント』ドキュメントの「INPUT *構文2 - 定義済みマップレイアウトの使用*」を参照してください。

マップ処理の開始／終了

*入力*マップは INPUT USING MAP ステートメントで呼び出されます。

*出力*マップは WRITE USING MAP ステートメントで呼び出されます。

マップの処理は処理ルール内の ESCAPE ROUTINE ステートメントで終了できます。

9 ヘルプルーチン

ヘルプルーチンには、ヘルプ要求の処理を容易にするための特徴があります。 複雑な対話形式のヘルプシステムを実装するために使用できます。 ヘルプルーチンはプログラムエディタで作成します。

このchapterでは、次のトピックについて説明します。

ヘルプの呼び出し

Naturalユーザーは、ヘルプ文字（デフォルト文字は"?"）をフィールドに入力するか、または、ヘルプキー（通常は PF1）を押して Natural ヘルプルーチンを呼び出すことができます。

- ヘルプ文字は 1 回しか入力できません。
- ヘルプ文字は入力文字列内で変更される唯一の文字である必要があります。
- ヘルプ文字は入力文字列内の最初の文字である必要があります。

ヘルプルーチンが数値フィールドに対して指定されている場合、Naturalではそのフィールドのヘルプルーチンを呼び出す目的で疑問符を入力できます。 その場合も、Naturalではフィールド入力として有効な数値データが提供されることをチェックします。

まだ指定されていない場合は、SET KEY ステートメントでヘルプキーを指定できます。

```
SET KEY PF1=HELP
```

ヘルプルーチンは、それが呼び出されるプログラムまたはマップに指定されている場合にのみ、ユーザーが呼び出すことができます。

ヘルプルーチンの指定

ヘルプルーチンは次のように指定できます。

- プログラム内では、ステートメントレベルおよびフィールドレベルで指定。
- マップ内では、マップレベルおよびフィールドレベルで指定。

ヘルプが指定されていないフィールドに対してヘルプが要求された場合や、参照するフィールドのないヘルプが要求された場合は、ステートメントレベルまたはマップレベルで指定されたヘルプルーチンが呼び出されます。

ヘルプルーチンは、プログラム自体または処理ルール内の REINPUT USING HELP ステートメントを使用して呼び出すこともできます。 REINPUT USING HELP ステートメントに MARK オプションが含まれている場合は、MARK されたフィールドに割り当てられているヘルプルーチンが呼び出されます。 フィールド固有のヘルプルーチンが割り当てられていない場合は、マップのヘルプルーチンが呼び出されます。

ヘルプルーチン内の REINPUT ステートメントは、同じヘルプルーチン内の INPUT ステートメントにのみ適用できます。

ヘルプルーチンの名前は、次に示すような INPUT ステートメントのセッションパラメータ HE で指定できます。

```
INPUT (HE='HELP2112')
```

または、マップエディタの拡張フィールド編集機能を使用して指定することもできます。「*マップの作成*」および『エディタ』ドキュメントを参照してください。

ヘルプルーチンの名前は、英数字定数または名前が含まれる英数字変数で指定できます。 定数の場合は、ヘルプルーチン名をアポストロフィで囲む必要があります。

ヘルプルーチンのプログラミングについて

ヘルプルーチンの処理は、ESCAPE ROUTINE ステートメントで終了できます。

ヘルプルーチンで END OF TRANSACTION または BACKOUT TRANSACTION ステートメントを使用すると、メインプログラムのトランザクションロジックに影響を与えるため注意してください。

ヘルプルーチンとのパラメータの受け渡し

ヘルプルーチンは、現在アクティブなグローバルデータエリアにアクセスできますが、独自のグローバルデータエリアを持つことはできません。 また、独自のローカルデータエリアを持つことができます。

パラメータを使用した、ヘルプルーチンとのデータの受け渡しも可能です。1つのヘルプルーチンに、20 個までの明示的なパラメータと 1 個の暗黙的なパラメータを指定できます。 明示的なパラメータは、ヘルプルーチン名の後に HE オペランドを付けて指定します。

```
HE='MYHELP','001'
```

暗黙的なパラメータとは、ヘルプルーチンが呼び出されたフィールドです。

```
INPUT #A (A5) (HE='YOURHELP','001')
```

上記の 001 は明示的なパラメータであり、#A は暗黙的なパラメータ／フィールドです。

これは、ヘルプルーチンの DEFINE DATA PARAMETER ステートメント内で次のように指定します。

```
DEFINE DATA PARAMETER
1 #PARM1 (A3)          /* explicit parameter
1 #PARM2 (A5)          /* implicit parameter
END-DEFINE
```

暗黙的なパラメータ（上記例の #PARM2）は省略できることに注意してください。 暗黙的なパラメータは、ヘルプが要求されたフィールドにアクセスして、このフィールドにヘルプルーチンからデータを返すために使用します。 例えば、計算機能プログラムをヘルプルーチンとして実装し、計算結果をフィールドに返すことができます。

ヘルプが呼び出されると、データが画面からプログラムデータエリアに渡される前に、ヘルプルーチンが呼び出されます。 これは、ヘルプルーチンは同じ画面トランザクション内で入力されたデータにアクセスできないことを意味しています。

ヘルプ処理が完了すると、画面データはリフレッシュされます。ヘルプルーチンによって変更されたフィールドは更新されます。ヘルプルーチンが呼び出される前にユーザーによって変更されていたフィールドは更新から除外されますが、ヘルプが要求されたフィールドは更新に含まれます。 *例外：*ヘルプが要求されたフィールドが、ダイナミック属性（DY セッションパラメータ）によって複数の部分に分割され、疑問符が入力された部分が、ユーザーによって変更された部分の*後*にある場合、フィールドの内容はヘルプルーチンでは変更されません。

属性制御変数は、ヘルプルーチン処理の後では、たとえヘルプルーチン内で変更されている場合でも再評価されることはありません。

等号オプション

等号（=）は明示的なパラメータとして指定できます。

```
INPUT PERSONNEL-NUMBER (HE='HELPROUT',=)
```

このパラメータは、フィールド名（マップレベルで指定された場合はマップ名）を含む内部フィールド（A65）として処理されます。 対応するヘルプルーチンは、次のように始まります。

```
DEFINE DATA PARAMETER
1 FNAME (A65)              /* contains 'PERSONNEL-NUMBER'
1 FVALUE (N8)              /* value of field (optional)
END-DEFINE
```

このオプションは、フィールド名を読み取り、アプリケーションのオンラインドキュメントまたは Predict データディクショナリにアクセスしてフィールド固有のヘルプを提供する 1 つの共通ヘルプルーチンにアクセスするために使用します。

配列インデックス

ヘルプ文字またはヘルプキーで選択されたフィールドが配列要素である場合、そのインデックスは暗黙的なパラメータ（明示的なパラメータに関係なく、ランクに依存して 1～3）として指定されます。

これらのパラメータのフォーマット／長さは I2 です。

```
INPUT A(*,*)  (HE='HELPROUT',=)
```

対応するヘルプルーチンは、次のように始まります。

```
DEFINE DATA PARAMETER
1 FNAME   (A65)            /* contains 'A'
1 FVALUE  (N8)             /* value of selected element
1 FINDEX1 (I2)             /* 1st dimension index
1 FINDEX2 (I2)             /* 2nd dimension index
END-DEFINE
...
```

ウィンドウとしてのヘルプ

表示するヘルプのサイズを画面サイズよりも小さくすることができます。 この場合、ヘルプは次に示すような、フレームで囲まれたウィンドウとして画面に表示されます。

```
********************************************************************************
                          PERSONNEL INFORMATION
PLEASE ENTER NAME: ?_________________
PLEASE ENTER CITY: ____________________
                   +--------------------------+
                   !                          !
                   ! Type in the name of an   !
                   ! employee in the first    !
                   ! field and press ENTER.   !
                   ! You will then receive    !
                   ! a list of all employees  !
                   ! of that name.            !
                   !                          !
                   ! For a list of employees  !
                   ! of a certain name who    !
                   ! live in a certain city,  !
                   ! type in a name in the    !
                   ! first field and a city   !
```

```
                    ! in the second field       !
                    ! and press ENTER.          !
********************!                           !*******************************
                    +---------------------------+
```

ヘルプルーチン内では、ウィンドウのサイズを次の方法で指定できます。

- FORMAT ステートメントを使用します。例えば、FORMAT PS=15 LS=30 のようにページサイズと行サイズを指定します。
- INPUT USING MAP ステートメントを使用します。この場合は、マップのマップ設定で定義されたサイズが使用されます。
- DEFINE WINDOW ステートメントを使用します。この場合は、ウィンドウサイズを明示的に指定することも、内容に応じてNaturalで自動的にウィンドウサイズを決定することもできます。

ヘルプウィンドウの位置は、ヘルプが要求されたフィールドの位置から自動的に計算されます。Naturalでは、対応するフィールドに重ならずにできるだけ近くなる位置にウィンドウを配置します。 DEFINE WINDOW ステートメントを使用すると、自動位置決めを回避して、ユーザー自身がウィンドウ位置を決定できます。

ウィンドウ処理の詳細については、『ステートメント』ドキュメントの「DEFINE WINDOW」ステートメントおよび『*端末コマンド*』ドキュメントの端末コマンド「%W」を参照してください。

10 ソースコードの複数使用 - コピーコード

このchapterでは、コピーコードの利点と使用について説明します。

次のトピックについて説明します。

コピーコードの使用

コピーコードは、INCLUDE ステートメントを使用して別のオブジェクトに組み込むことができるソースコードの一部です。

したがって、複数オブジェクトに同じ形式で現れるステートメントブロックがある場合は、そのステートメントブロックを複数回コーディングする代わりにコピーコードを使用できます。 これにより、コーディングの労力が削減されるうえ、そのブロックが本当に同一であることも保証されます。

コピーコードの処理

コピーコードはコンパイル時に組み込まれます。つまり、コピーコードのソースコード行は、INCLUDE ステートメントを含むオブジェクトに物理的に挿入されるのではなく、コンパイル処理で組み込まれて、結果的に作成されるオブジェクトモジュールの一部となります。

したがって、コピーコードのソースコードを修正するときは、そのコピーコードを使用するすべてオブジェクトも新しくコンパイル（STOW）する必要があります。

注意：

- コピーコードは独自に実行できません。 STOW することはできず、SAVE のみが可能です。
- END ステートメントをコピーコード内に指定することはできません。

詳細については、『ステートメント』ドキュメントの INCLUDE ステートメントを参照してください。

11 Natural オブジェクトのドキュメント化 - テキスト

Natural オブジェクトタイプ「テキスト」は、プログラムではなくテキストを記述するために使用します。

このchapterでは、次のトピックについて説明します。

テキストオブジェクトの使用

このオブジェクトタイプを使用すると、プログラムのソースコードなどに記述するよりも詳細にNatural オブジェクトをドキュメント化することができます。

「テキスト」オブジェクトは、プログラムをドキュメント化する目的でPredictを使用できない環境でも役立ちます。

テキストの記述

テキストは Natural プログラムエディタを使用して記述します。

プログラムの記述との操作の唯一の違いは、テキストが記述されたまま変わらない点です。つまり、小文字から大文字へ変換されたり、空行が省略されたりすることはありません。ただし、これはエディタプロファイルの空行の省略を"N"に、小文字での編集を"Y"に設定している場合に限ります。詳細については『エディタ』ドキュメントを参照してください。

任意のテキストを記述できます。構文チェックはありません。

「テキスト」オブジェクトは SAVE のみが可能です。STOW することはできません。 エディタに表示されるだけで、RUN で実行することはできません。

12 イベントドリブンアプリケーションの作成 - ダイアログ

ダイアログは、グラフィカルユーザーインターフェイス（GUI）用のNaturalアプリケーションを作成するときに、イベントドリブンプログラミングとともに使用します。

ダイアログおよびイベントドリブンプログラミングの詳細については、「*イベントドリブンプログラミング*」を参照してください。

13 コンポーネントベースのアプリケーションの作成 - クラス

クラスは、クライアント／サーバー環境でコンポーネントベースのアプリケーションを作成および配布するために使用します。 詳細については、『*プログラミングガイド*』の「***NaturalX***」セクションおよび『*エディタ*』ドキュメントの「*クラスビルダ*」を参照してください。

14 Natural 以外のファイルの使用 - リソース

このセクションでは、リソースタイプの Natural オブジェクトについて説明します。

このchapterでは、次のトピックについて説明します。

リソースの使用

Natural では次の 2 種類のリソースが区別されます。

- 共有リソース
  共有リソースは、Natural アプリケーションで使用される Natural 以外の任意のファイルで、Natural ライブラリシステムで管理されます。
- プライベートリソース
  プライベートリソースは、唯一のNaturalオブジェクトのみに割り当てられ、そのオブジェクトの一部であるとみなされるファイルです。オブジェクトは最大で1つのプライベートリソースファイルを持つことができます。 現時点でプライベートリソースを持つのは **Natural** ダイアログのみです。

Natural ライブラリに属する共有リソースもプライベートリソースも、ファイルシステム内の Natural ライブラリを表すディレクトリの *..\RES* という名前のサブディレクトリで管理されます。

共有リソース

共有リソースは、Natural アプリケーションで使用される Natural 以外の任意のファイルで、Natural ライブラリシステムで管理されます。 共有リソースとして使用する Natural 以外のファイルは、Natural ライブラリの *..\RES* という名前のサブディレクトリに含める必要があります。

共有リソースの使用の例

ビットマップ *MYPICTURE.BMP* は、ライブラリ `MYLIB` に含まれるダイアログ `MYDLG` のビットマップコントロールに表示されます。 最初にビットマップは、ディレクトリ *..\MYLIB\RES* に移されることによって Natural ライブラリ `MYLIB` に挿入されます。 次のコードは、ダイアログ `MYDLG` のコードの一部で、その後このダイアログをビットマップコントロールに割り当てる方法を示しています。

```
DEFINE DATA LOCAL
01 #BM-1 HANDLE OF BITMAP
...
END-DEFINE
* (Creation of the Bitmap control omitted.)
...
#BM-1.BITMAP-FILE-NAME := "MYPICTURE.BMP" ...
```

ビットマップを共有リソースとして使用する利点は次のとおりです。

- パス名を指定しなくても Natural ダイアログにファイル名を指定できます。
- ファイルは、ファイルを使用する Natural オブジェクトとともに Natural ライブラリに保持できます。

Note: 以前の Natural バージョンでは、Natural 以外のファイルは、通常は環境変数 NATGUI_BMP で定義されたディレクトリに保管されていました。 この方法を使用する既存のアプリケーションは、これまでと同様に作動します。現在のライブラリで共有リソースファイルが見つからない場合、Natural は常にこのディレクトリを検索するからです。

プライベートリソース

プライベートリソースは、Natural オブジェクトの一部であるバイナリデータを保存するために Natural によって内部的に使用されます。 この場合のファイルは、ファイル名の拡張子 *NR** で認識されます。"*" は Natural オブジェクトのタイプに依存する 1 文字です。 Natural ではプライベートリソースファイルとその内容が自動的に管理されます。Natural オブジェクトは、最大 1 つのプライベートリソースファイルを持つことができます。

現時点でプライベートリソースファイルを持つのは **Natural** ダイアログのみです。 このファイルは、ダイアログに定義され、独自のプロパティページで構成される ActiveX コントロールのコンフィグレーションを保存するために使用されます。

ActiveX コントロールの構成方法については、「*ダイアログおよびダイアログエレメントの属性ウィンドウ*」および「*ActiveX コントロールプロパティページ*」を参照してください。

プライベートリソースの例

ダイアログ MYDLG のプライベートリソースファイルの名前は *MYDLG.NR3* になります。

ダイアログの作成、修正、削除などが行われると、Naturalでは必要に応じてこのファイルが自動的に作成、修正、削除されます。

プライベートリソースファイルは、ダイアログ MYDLG に関連するバイナリデータを保存するために使用されます。

リソースを処理する API

ライブラリ SYSEXT には次のアプリケーションプログラミングインターフェイス（API）が存在し、ユーザーアプリケーションはこれを使用してリソース独自のユーザー出口ルーチンにアクセスできます。

API	目的
USR4208N	ショートネームまたはロングネームを使用して、リソースの書き込み、読み取り、削除を行います。

15 フィールドの定義

このセクションでは、プログラムで使用するフィールドを定義する方法について説明します。これらのフィールドは、データベースフィールドおよびユーザー定義フィールドです。

以下のトピックについて説明します。

- **DEFINE DATA** ステートメントの使用と構造
- ユーザー定義変数
- ファンクションコール
- ダイナミック変数およびフィールドについて
- ダイナミック変数およびラージ変数の使用
- ユーザー定義定数
- 初期値（および **RESET** ステートメント）
- フィールドの再定義
- 配列
- **X-array**

ここでは、`DEFINE DATA` ステートメントの主要なオプションのみを説明します。 オプションの詳細については、『ステートメント』ドキュメントを参照してください。

データベースフィールドの詳細については、「*Adabas データベースのデータへのアクセス*」を参照してください。 そのセクションで説明されている Adabas 用の 機能と例は、基本的に他のデータベース管理システムにも適用できます。 システムごとの差異については、『ステートメ

ント』ドキュメントまたは『パラメータリファレンス』ドキュメントの該当するデータベースインターフェイスの説明を参照してください。

16 DEFINE DATA ステートメントの使用と構造

Natural プログラム内にストラクチャードモードで記述されている最初のステートメントは常に、プログラムで使用するフィールドを定義するために使用される DEFINE DATA ステートメントである必要があります。

このchapterでは、次のトピックについて説明します。

ソースプログラムのインデントについては、Natural システムコマンド STRUCT の説明を参照してください。

DEFINE DATA ステートメントにおけるフィールド定義

DEFINE DATA ステートメントで、プログラムで使用するすべてのフィールド（ユーザー定義変数およびデータベースフィールド）を定義します。

使用するすべてのフィールドは、DEFINE DATA ステートメントで定義する*必要があります*。

フィールドを定義するには、以下の 2 つの方法があります。

- DEFINE DATA ステートメントそのものでフィールドを定義します（下記参照）。
- プログラム外のローカルデータエリアまたはグローバルデータエリアでフィールドを定義し、DEFINE DATA ステートメントでそのデータエリアを参照します（下記参照）。

複数のプログラム／ルーチンでフィールドを使用する場合は、プログラム外のデータエリアで定義する必要があります。

アプリケーション構造を明確にするために、通常はプログラム外のデータエリアにフィールドを定義することをお勧めします。

データエリアは、データエリアエディタ（『エディタ』ドキュメントを参照）で作成およびメンテナンスします。

以下の最初の例では、プログラムの DEFINE DATA ステートメント内でフィールドを定義しています。 **2** 番目の例では、同じフィールドをローカルデータエリア（LDA）で定義し、DEFINE DATA ステートメントではそのデータエリアへの参照だけを指定しています。

DEFINE DATA ステートメント内でのフィールド定義

以下の例は、DEFINE DATA ステートメント内でのフィールドの定義方法を示しています。

```
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 PERSONNEL-ID
1 #VARI-A (A20)
1 #VARI-B (N3.2)
1 #VARI-C (I4)
END-DEFINE
...
```

別のデータエリアでのフィールドの定義

以下の例は、ローカルデータエリア（LDA）でのフィールドの定義方法を示しています。

プログラム：

```
DEFINE DATA LOCAL
  USING LDA39
END-DEFINE
...
```

ローカルデータエリア LDA39：

```
I T L Name                            F Leng Index/Init/EM/Name/Comment
- - - --------------------------------- - ---- ---------------------------------
  V 1 VIEWEMP                                 EMPLOYEES
    2 NAME                            A   20
    2 FIRST-NAME                      A   20
    2 PERSONNEL-ID                    A    8
    1 #VARI-A                         A   20
    1 #VARI-B                         N  3.2
```

```
    1 #VARI-C                          I     4
```

レベル番号を使用した DEFINE DATA ステートメントの構造化

以下のトピックについて説明します。

- 定義の構造化およびグループ化
- ビュー定義のレベル番号
- フィールドグループのレベル番号
- 再定義のレベル番号

定義の構造化およびグループ化

レベル番号は、定義の構造およびグループ化を示すために、DEFINE DATA ステートメント内で使用します。 これは以下に関連します。

- ビュー定義
- フィールドグループ
- 再定義

レベル番号は、01～99 の範囲内の 1 桁または 2 桁の数字です（先頭の "0" は任意）。

一般的に、変数定義はレベル 1 です。

ビュー定義、再定義、およびグループでのレベル番号は、連続している必要があります。レベル番号はスキップできません。

ビュー定義のレベル番号

ビューを定義する場合、ビュー名はレベル 1、ビューを構成するフィールドはレベル 2 で指定する必要があります。 ビュー定義の詳細については、データベースアクセスの説明を参照してください。

ビュー定義のレベル番号の例：

```
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 BIRTH
```

```
...
END-DEFINE
```

フィールドグループのレベル番号

グループを定義すると、連続する一連のフィールドを簡単に参照できます。 共通のグループ名の下に複数のフィールドを定義すると、後で、個々のフィールド名の代わりにグループ名のみを指定することによって、プログラム内でフィールドを参照できます。

グループ名はレベル1で指定し、グループに含まれるフィールドは1つ低いレベルにする必要があります。

グループ名には、ユーザー定義変数と同じ命名規則が適用されます。

グループのレベル番号の例：

```
DEFINE DATA LOCAL
1 #FIELDA (N2.2)
1 #FIELDB (I4)
1 #GROUPA
  2 #FIELDC (A20)
  2 #FIELDD (A10)
  2 #FIELDE (N3.2)
1 #FIELDF (A2)
...
END-DEFINE
```

この例では、フィールド #FIELDC、#FIELDD、および #FIELDE は、共通のグループ名 #GROUPA の下に定義されています。 他の 3 つのフィールドはグループの一部ではありません。 #GROUPA はグループ名として機能するだけで、それ自体はフィールドではないことに注意してください（したがって、フォーマット／長さ定義を持っていません）。

再定義のレベル番号

フィールドを再定義する場合、REDEFINE オプションは元のフィールドと同じレベルにする必要があります。再定義から作成するフィールドは、1つ低いレベルにする必要があります。 再定義の詳細については、「*フィールドの再定義*」を参照してください。

再定義のレベル番号の例：

```
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF STAFFDDM
  2 BIRTH
  2 REDEFINE BIRTH
    3 #YEAR-OF-BIRTH  (N4)
    3 #MONTH-OF-BIRTH (N2)
    3 #DAY-OF-BIRTH   (N2)
1 #FIELDA (A20)
1 REDEFINE #FIELDA
  2 #SUBFIELD1 (N5)
  2 #SUBFIELD2 (A10)
  2 #SUBFIELD3 (N5)
...
END-DEFINE
```

この例では、データベースフィールド BIRTH は、3 つのユーザー定義変数として再定義され、ユーザー定義変数の #FIELDA は、他の 3 つのユーザー定義変数として再定義されています。

17 ユーザー定義変数

ユーザー定義変数は、プログラム内でユーザーが定義するフィールドです。 ユーザー定義変数は、追加の処理または表示のためにプログラム処理の特定のポイントで得られた値または中間結果を保存するために使用します。

このchapterでは、次のトピックについて説明します。

『*Natural スタジオの使用*』ドキュメントの「*ユーザー定義変数の命名規則*」も参照してください。

変数の定義

ユーザー定義変数を定義するには、`DEFINE DATA` ステートメントで名前とフォーマット／長さを指定します。

以下の表記に従って、変数の特徴を定義します。

(*r*,*format-length*/*index*)

この表記は変数名の後に、任意で 1 つ以上の空白で区切って記述します。

各表記要素の間に空白を入れることはできません。

各要素は、必要に応じて個別に指定できます。まとめて指定する場合は、上記の文字で区切る必要があります。

例：

以下の例では、英数字フォーマットで 10 桁の長さのユーザー定義変数が、`#FIELD1` という名前で定義されています。

```
DEFINE DATA LOCAL
1 #FIELD1 (A10)
...
END-DEFINE
```

Notes:

1. ストラクチャードモードで操作している場合、またはプログラムに `DEFINE DATA LOCAL` 節が含まれている場合、ステートメント内で変数をダイナミックに定義することはできません。
2. これは、アプリケーションに依存しない変数（AIV）には適用されません。「*アプリケーションに依存しない変数の定義*」も参照してください。

表記 (*r*) を使用したデータベースフィールドの参照

ステートメントラベルまたはソースコード行番号は、前のNaturalステートメントを参照するために使用できます。この参照は、Naturalのデフォルトの参照を変更したり（各ステートメントの記述に従って実行可能な場合）、処理内容を説明したりするために使用できます。「*ループ処理*」の「*プログラム内のステートメント参照*」も参照してください。

以下では次のトピックについて説明します。

- データベースフィールドのデフォルトの参照
- ステートメントラベルによる参照
- ソースコード行番号による参照

データベースフィールドのデフォルトの参照

ステートメント参照表記を指定しない場合、通常は以下の参照が適用されます。

- デフォルトでは、最も内側のアクティブなデータベースループ（FIND、READ、HISTOGRAM）で読み込まれているフィールドが参照されます。
- アクティブなデータベースループでフィールドが読み込まれていない場合、まだクローズされていないループ内にあり、かつ、当該フィールドが読み込まれている、直前のGETステートメント（レポーティングモードではFIND FIRSTまたはFIND UNIQUEステートメントも可）が参照されます。

ステートメントラベルによる参照

ループ処理を開始したり、データ要素によるデータベースへのアクセスを実行したりするNaturalステートメントには、後で参照するための記号ラベルをマークできます。

ラベルは、参照するオブジェクトの前に"*label*."という形式で指定するか、または参照するオブジェクトの後に"(*label*.)"という形式で指定します。ただし、同時に指定することはできません。

ラベルの命名規則は、変数の名前の命名規則と同一です。 ラベル名の後のピリオドは、そのエントリをラベルとして識別するために機能します。

例：

```
...
 RD. READ PERSON-VIEW BY NAME STARTING FROM 'JONES'
   FD. FIND AUTO-VIEW WITH PERSONNEL-ID = PERSONNEL-ID (FD.)
         DISPLAY NAME (RD.) FIRST-NAME (RD.) MAKE (FD.)
    END-FIND
 END-READ
 ...
```

ソースコード行番号による参照

ステートメントは、ステートメントが記述されているソースコード行を使用して参照することもできます。

4 桁すべての行番号を指定する必要があります（先行ゼロは省略不可能）。

例：

```
...
0110 FIND EMPLOYEES-VIEW WITH NAME = 'SMITH'
0120   FIND VEHICLES-VIEW WITH MODEL = 'FORD'
0130     DISPLAY NAME (0110) MODEL (0120)
0140   END-FIND
0150 END-FIND
...
```

参照するソースコード行番号の変更

Natural ソースプログラムの行番号が変更されると、ステートメントを参照する 4 桁のソースコード行番号（「*表記 (r) を使用したデータベースフィールドの参照*」を参照）も変更されます。 ユーザーの使いやすさ、読みやすさ、およびデバッグのしやすさのために、ステートメント、英数字定数、またはコメントで行われるソースコード行番号への参照はすべて、番号が振り直されます。 ステートメントや英数字定数内のソースコード行番号参照の位置（先頭、中間、最後）は影響しません。

有効なソースコード行番号参照として認識され、番号が振り直されるパターンは以下のとおりです（*nnnn* は 4 桁の数字）。

パターン	ステートメントの例
(*nnnn*)	`ESCAPE BOTTOM (0150)`
(*nnnn* /	`DISPLAY ADDRESS-LINE(0010/1:5)`
(*nnnn*,	`DISPLAY NAME(0010,A10/1:5)`

左側のカッコまたは 4 桁の数字 *nnnn* の後に空白がある、または 4 桁の数字 *nnnn* の後にピリオドがある場合、そのパターンは有効なソースコード行参照とみなされません。

英数字定数内の4桁の数字が不用意に変更されないようにするには、定数を複数に分割し、ハイフンを使用して単一の値に連結する必要があります。

例：

```
Z := 'XXXX (1234,00) YYYY'
```

上記の例は、以下のように変更します。

```
Z := 'XXXX (1234' - ',00) YYYY'
```

ユーザー定義変数のフォーマットおよび長さ

ユーザー定義変数のフォーマットおよび長さは、変数名の後のカッコ内に指定します。

固定長の変数は、以下のフォーマットおよび対応する長さで定義できます。

ダイナミック変数のフォーマットおよび長さについては、「*ダイナミック変数の定義*」を参照してください。

フォーマット		定義可能な長さ	内部的な長さ（バイト）
A	英数字	1～1073741824（1 GB）	1 - 1073741824
B	バイナリ	1～1073741824（1 GB）	1 - 1073741824
C	属性制御	-	2
D	日付	-	4
F	浮動小数点	4 または 8	4 または 8
I	整数	1、2 または 4	1、2 または 4
L	論理	-	1
N	数値（アンパック）	1 - 29	1 - 29

フォーマット		定義可能な長さ	内部的な長さ（バイト）
P	パック型数値	1 - 29	1 - 15
T	時刻	-	7
U	Unicode（UTF-16）	1～536870912（0.5 GB）	2 - 1073741824

長さは、フォーマットを指定した場合にのみ指定できます。 フォーマットによっては、長さを明示的に指定する必要のないものもあります（上記の表を参照）。

フォーマット N または P で定義するフィールドには、*nn.m* という形式で小数点の位置を指定できます。 *nn* は小数点の前の桁数を、*m* は小数点の後の桁数を表します。 *nn* と *m* を合計した値は 29 を超えることはできません。また、*m* の値は 7 を超えることはできません。

Notes:

1. フォーマット P のユーザー定義変数を DISPLAY、WRITE、INPUT の各ステートメントで出力する場合、Natural によって、出力のために内部的にフォーマット N に変換されます。
2. レポーティングモードでは、ユーザー定義変数のフォーマットおよび長さを指定しません。デフォルトの割り当てがプロファイル／セッションパラメータ FS によって無効化されていない限り、デフォルトのフォーマット／長さの N7 が使用されます。

データベースフィールドの場合、DDM でフィールドに対して定義されているフォーマット／長さが適用されます。レポーティングモードでは、データベースフィールドに対し、異なるフォーマット／長さをプログラム内で定義することもできます。

ストラクチャードモードでは、データエリア定義または DEFINE DATA ステートメントでのみ、フォーマットおよび長さを指定できます。

ストラクチャードモードでのフォーマット／長さの定義例：

```
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
1 #NEW-SALARY (N6.2)
END-DEFINE
...
FIND EMPLOY-VIEW ...
...
COMPUTE #NEW-SALARY = ...
...
```

レポーティングモードでは、DEFINE DATA ステートメントを使用していなければ、プログラムの本体でフォーマット／長さを定義できます。

レポーティングモードでのフォーマット／長さの定義例：

```
...
...
 FIND EMPLOYEES
... ... COMPUTE #NEW-SALARY(N6.2) = ...
...
```

特殊フォーマット

英数字（A）、数値（B、F、I、N、P）の各標準フォーマットの他に、Natural では以下の特殊フォーマットがサポートされています。

- フォーマット C - 属性制御
- フォーマット D - 日付、およびフォーマット T - 時刻
- フォーマット L - 論理
- フォーマット - ハンドル

フォーマット C - 属性制御

フォーマット C で定義されている変数は、DISPLAY、INPUT、WRITE の各ステートメントで使用されているフィールドに属性をダイナミックに割り当てるために使用できます。

フォーマットCの変数には、長さを指定できません。この変数には常に2バイトの長さがNatural によって割り当てられます。

例：

```
DEFINE DATA LOCAL
1 #ATTR (C)
1 #A (N5)
END-DEFINE
...
MOVE (AD=I CD=RE) TO #ATTR
INPUT #A (CV=#ATTR)
...
```

詳細については、セッションパラメータ CV の説明を参照してください。

フォーマット D - 日付、およびフォーマット T - 時刻

フォーマット D および T で定義されている変数は、日付および時刻の演算や表示に使用できます。 フォーマット D には、日付情報のみを指定できます。 フォーマット T には、日付および時刻の両方の情報を指定できます。つまり、日付情報は時刻情報のサブセットです。 時刻は、1/10 秒単位でカウントされます。

フォーマット D および T の変数には、長さを指定できません。 フォーマット D の変数には常に 4 バイトの長さ（P6）が、フォーマット T の変数には常に 7 バイトの長さ（P12）が、Natural によって割り当てられます。 プロファイルパラメータ MAXYEAR が 9999 に設定されている場合、フォーマット D の変数には常に 4 バイトの長さ（P7）が、フォーマット T の変数には常に 7 バイトの長さ（P13）が、Natural によって割り当てられます。

例：

```
DEFINE DATA LOCAL
1 #DAT1 (D)
END-DEFINE
*
MOVE *DATX TO #DAT1
ADD 7 TO #DAT1
WRITE '=' #DAT1
END
```

詳細については、セッションパラメータ EM と、システム変数 *DATX および *TIMX の説明を参照してください。

日付フィールドの値は、1582 年 1 月 1 日～2699 年 12 月 31 日の範囲内の値である必要があります。

フォーマット L - 論理

フォーマット L で定義されている変数は、論理条件の基準として使用できます。 この変数の値は、"TRUE" または "FALSE" になります。

フォーマット L の変数には、長さを指定できません。 フォーマット L の変数には常に 1 バイトの長さが Natural によって割り当てられます。

例：

```
DEFINE DATA LOCAL
1 #SWITCH(L)
END-DEFINE
MOVE TRUE TO #SWITCH
...
IF #SWITCH
   ...
   MOVE FALSE TO #SWITCH
ELSE
   ...
   MOVE TRUE TO #SWITCH
END-IF
```

論理値表示の詳細については、セッションパラメータ EM の説明を参照してください。

フォーマット - ハンドル

"HANDLE OF OBJECT" として定義されている変数は、オブジェクトハンドルとして使用できます。

オブジェクトハンドルの詳細については、「***NaturalX***」を参照してください。

"HANDLE OF *dialog-element-type*" として定義されている変数は、GUI ハンドルとして使用できます。

GUI ハンドルの詳細については、「*イベントドリブンプログラミングの手法*」の「***HANDLE OF GUI***」を参照してください。

インデックス表記

インデックス表記は、配列フィールドに対して使用します。

インデックス表記には、整数値定数またはユーザー定義変数を使用できます。 ユーザー定義変数は、フォーマット N（数値）、P（パック型）、I（整数）、B（バイナリ）のいずれかを使用して指定できます。フォーマット B は、長さが 4 以下の場合にのみ使用できます。

システム変数、システム関数、修飾された変数は、インデックス表記に使用できません。

配列の定義例：

1. #ARRAY (3)
   オカレンス数 3 の 1 次元配列を定義します。
2. FIELD (*label*.,A20/5) または*label*.FIELD(A20/5)
   "*label*."でマークしたステートメントを参照するデータベースフィールドを基にした、英数字フォーマットで長さ 20、オカレンス数 5 の配列を定義します。
3. #ARRAY (N7.2/1:5,10:12,1:4)
   最初のオカレンス数が 5、2 番目のオカレンス数が 3、3 番目のオカレンス数が 4 である、フォーマット／長さが N7.2 の 3 次元配列を定義します。
4. FIELD (*label*./i:i + 5) または*label*.FIELD(i:i + 5)
   "*label*."でマークしたステートメントを参照するデータベースフィールドを基にした配列を定義します。

   FIELDは、マルチプルバリューフィールド、またはピリオディックグループのフィールドを表します。"*i*"を使用して、データベースオカレンス内のインデックスのオフセットを指定します。プログラム内では、この配列のサイズはオカレンス数 6（i:i + 5）と定義されます。マルチプルバリューフィールドまたはピリオディックグループのオカレンスに対する、プログラムの配列の位置を指定するには、データベースインデックスのオフセットを変数として指定します。"*i*" を再設定すると、GET または GET SAME ステートメントを使用してデータベースに新規にアクセスする必要があります。

Natural では、インデックスが "1" で始まらない配列を定義できます。ランタイムに Natural によって、参照で指定しているインデックス値が、定義で指定されている次元の最大サイズを超えていないかどうかがチェックされます。

Notes:

1. 以前の Natural バージョンとの互換性を維持するために、コロン（:）の代わりにハイフン（-）を使用して配列の範囲を指定することもできます。
2. ただし、両方の表記を混在させることはできません。
3. ハイフン表記は、レポーティングモードでのみ使用できます。ただし、DEFINE DATA ステートメントでは使用できません。

インデックスの最大値は 1,073,741,824 です。プログラミングオブジェクトごとのデータエリアの最大サイズは 1,073,741,824 バイト（1 GB）です。

インデックス参照には、"+" および "-" 演算子を使用した単純な演算式を使用できます。演算式をインデックスとして使用する場合、演算子 "+" または "-" の前後に空白を入れる必要があります。

グループ構造内の配列は、Naturalによって、グループのオカレンスごとではなくフィールドごとに分解されます。

グループ配列の分解例：

```
DEFINE DATA LOCAL
 1 #GROUP (1:2)
   2 #FIELDA (A5/1:2)
   2 #FIELDB (A5)
 END-DEFINE
 ...
```

上記のように定義されているグループを、以下の WRITE ステートメントで出力します。

```
WRITE #GROUP (*)
```

オカレンスは、以下の順序で出力されます。

```
#FIELDA(1,1) #FIELDA(1,2) #FIELDA(2,1) #FIELDA(2,2) #FIELDB(1) #FIELDB(2)
```

以下の順序ではありません。

```
#FIELDA(1,1) #FIELDA(1,2) #FIELDB(1) #FIELDA(2,1) #FIELDA(2,2) #FIELDB(2)
```

配列の参照例：

1. #ARRAY (1)
   1 次元配列の最初のオカレンスを参照します。
2. #ARRAY (7:12)
   1 次元配列の 7～12 番目のオカレンスを参照します。
3. #ARRAY (i + 5)
   1 次元配列の i + 5 番目のオカレンスを参照します。
4. #ARRAY (5,3:7,1:4)
   3 次元配列の、第 1 次元の 5 番目のオカレンス、第 2 次元の 3～7 番目のオカレンス（オカレンス数5）、第3次元のオカレンス1～4番目のオカレンス（オカレンス数4）を参照します。
5. アスタリスクは、次元内のすべてのオカレンスを参照するために使用できます。

```
DEFINE DATA LOCAL
1 #ARRAY1 (N5/1:4,1:4)
1 #ARRAY2 (N5/1:4,1:4)
END-DEFINE
...
ADD #ARRAY1 (2,*) TO #ARRAY2 (4,*)
...
```

配列オカレンスの前のスラッシュの使用

変数名の後にカッコで囲んだ4桁の数字を指定すると、この番号はステートメントに対する行番号参照だと Natural によって解釈されます。 したがって、4 桁の配列オカレンスを指定する場合、以下の例のように、番号の前にスラッシュ "/" を指定して、その番号が配列オカレンスであることを示す必要があります。

```
#ARRAY(/1000)
```

以下のように指定しないでください。

```
#ARRAY(1000)
```

後者の例は、ソースコード行 1000 への参照と解釈されます。

インデックス変数名がフォーマット／長さ指定と間違って解釈される可能性がある場合、スラッシュ "/" を指定して、インデックスが指定されていることを示す必要があります。 例えば、配列のオカレンスを "N7" という変数の値で定義する場合、以下のようにオカレンスを指定する必要があります。

```
#ARRAY (/N7)
```

以下のように指定しないでください。

```
#ARRAY (N7)
```

後者の例は、7 バイトの数値フィールド定義と解釈されます。

データベース配列の参照

以下では次のトピックについて説明します。

- マルチプルバリューフィールドとピリオディックグループフィールドの参照
- 定数を使用して定義されている配列の参照
- 変数を使用して定義されている配列の参照
- 複数定義配列の参照

Note: 以下のサンプルプログラムを実行する前に、SYSEXPG ライブラリ内の INDEXTST プログラムを実行して、10 種類の言語コードを使用するサンプルレコードを作成してください。

マルチプルバリューフィールドとピリオディックグループフィールドの参照

ビュー／DDM 内のマルチプルバリューフィールドまたはピリオディックグループフィールドは、さまざまなインデックス表記を使用して定義および参照できます。

例えば、以下のように、データベースレコード内の同じマルチプルバリューフィールド／ピリオディックグループフィールドの 1～10 番目の値、および I～I + 10 番目の値を参照できます。

```
DEFINE DATA LOCAL
1 I (I2)
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 LANG (1:10)
  2 LANG (I:I+10)
END-DEFINE
```

または

```
RESET I (I2)
...
READ EMPLOYEES
OBTAIN LANG(1:10) LANG(I:I+10)
```

Notes:

1. 同じ下限のインデックス値を使用できるのは、配列ごとに一度だけです。これは、変数インデックスだけでなく定数インデックスの場合も同様です。
2. 変数インデックスを使用した配列定義では、変数自身を使用して下限を、下限と同じ変数 + 定数を使用して上限を指定する必要があります。

定数を使用して定義されている配列の参照

定数を使用して定義されている配列は、定数または変数のいずれかを使用して参照できます。ただし、配列の上限を超えることはできません。 定数を使用すると、コンパイル時に Natural によって、上限を超えているかどうかがチェックされます。

レポーティングモードの例：

```
** Example 'INDEX1R': Array definition with constants  (reporting mode)
************************************************************************
*
READ (1) EMPLOYEES WITH NAME = 'WINTER' WHERE CITY = 'LONDON'
  OBTAIN LANG (1:10)
  /*
  WRITE 'LANG(1:10):' LANG (1:10) //
  WRITE 'LANG(1)   :' LANG (1)    /  'LANG(5:9) :' LANG (5:9)
LOOP
*
END
```

ストラクチャードモードの例：

```
** Example 'INDEX1S': Array definition with constants (structured mode)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
  2 LANG (1:10)
END-DEFINE
*
READ (1) EMPLOY-VIEW WITH NAME = 'WINTER' WHERE CITY = 'LONDON'
  WRITE 'LANG(1:10):' LANG (1:10) //
  WRITE 'LANG(1)   :' LANG (1)    /  'LANG(5:9) :' LANG (5:9)
END-READ
END
```

定数を使用して複数回定義されているマルチプルバリューフィールドまたはピリオディックグループフィールドを、変数を使用して参照する場合、以下のような構文を使用します。

レポーティングモードの例：

```
** Example 'INDEX2R': Array definition with constants (reporting mode)
**                   (multiple definition of same database field)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 LANG (1:5)
  2 LANG (4:8)
END-DEFINE
*
READ (1) EMPLOY-VIEW WITH NAME = 'WINTER' WHERE CITY = 'LONDON'
  DISPLAY 'NAME'         NAME
```

```
          'LANGUAGE/1:3' LANG (1.1:3)
          'LANGUAGE/6:8' LANG (4.3:5)
LOOP
*
END
```

ストラクチャードモードの例：

```
** Example 'INDEX2S': Array definition with constants (structured mode)
**                    (multiple definition of same database field)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 LANG (1:5)
  2 LANG (4:8)
END-DEFINE
*
READ (1) EMPLOY-VIEW WITH NAME = 'WINTER' WHERE CITY = 'LONDON'
  DISPLAY 'NAME'         NAME
          'LANGUAGE/1:3' LANG (1.1:3)
          'LANGUAGE/6:8' LANG (4.3:5)
END-READ
*
END
```

変数を使用して定義されている配列の参照

変数を使用して定義されている配列内のマルチプルバリューフィールドまたはピリオディックグループフィールドは、同じ変数を使用して参照する必要があります。

レポーティングモードの例：

```
** Example 'INDEX3R': Array definition with variables (reporting mode)
************************************************************************
RESET I (I2)
*
I := 1
READ (1) EMPLOYEES WITH NAME = 'WINTER' WHERE CITY = 'LONDON'
  OBTAIN LANG (I:I+10)
  /*
  WRITE 'LANG(I)      :' LANG (I) /
        'LANG(I+5:I+7):' LANG (I+5:I+7)
LOOP
```

```
*
END
```

ストラクチャードモードの例：

```
** Example 'INDEX3S': Array definition with variables (structured mode)
************************************************************************
DEFINE DATA LOCAL
1 I (I2)
*
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 LANG (I:I+10)
END-DEFINE
*
I := 1
READ (1) EMPLOY-VIEW WITH NAME = 'WINTER' WHERE CITY = 'LONDON'
  WRITE 'LANG(I)      :' LANG (I) /
        'LANG(I+5:I+7):' LANG (I+5:I+7)
END-READ
END
```

別のインデックスを使用すると、変数インデックスを使用した配列の最初の定義を明確に参照できます。 これは、以下のようなインデックス表現を使用すると実行できます。

レポーティングモードの例：

```
** Example 'INDEX4R': Array definition with variables (reporting mode)
************************************************************************
RESET I (I2) J (I2)
*
I := 2
J := 3
*
READ (1) EMPLOYEES WITH NAME = 'WINTER' WHERE CITY = 'LONDON'
  OBTAIN LANG (I:I+10)
  /*
  WRITE 'LANG(I.J)  :' LANG (I.J) /
        'LANG(I.1:5):' LANG (I.1:5)
LOOP
*
END
```

ストラクチャードモードの例：

```
** Example 'INDEX4S': Array definition with variables (structured mode)
************************************************************************
DEFINE DATA LOCAL
1 I (I2)
1 J (I2)
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 LANG (I:I+10)
END-DEFINE
*
I := 2
J := 3
READ (1) EMPLOY-VIEW WITH NAME = 'WINTER' WHERE CITY = 'LONDON'
  WRITE 'LANG(I.J)  :' LANG (I.J) /
        'LANG(I.1:5):' LANG (I.1:5)
END-READ
END
```

式"I."は、読み込み配列の範囲（LANG(I.1:5)）内の最初の「位置」を明示的に参照するために使用します。

データベースにアクセスした時点での "I" の値によって、データベース配列の最初のオカレンスが決まります。

複数定義配列の参照

複数定義配列の場合は通常、適切な配列の範囲を明確に参照できるように、インデックス表現に修飾子を使用する必要があります。

レポーティングモードの例：

```
** Example 'INDEX5R': Array definition with constants (reporting mode)
**                   (multiple definition of same database field)
************************************************************************
DEFINE DATA LOCAL                   /* For reporting mode programs
1 EMPLOY-VIEW VIEW OF EMPLOYEES     /* DEFINE DATA is recommended
  2 NAME                            /* to use multiple definitions
  2 CITY                            /* of same database field
  2 LANG (1:10)
  2 LANG (5:10)
*
1 I (I2)
1 J (I2)
END-DEFINE
*
I := 1
J := 2
```

```
*
READ (1) EMPLOY-VIEW WITH NAME = 'WINTER' WHERE CITY = 'LONDON'
  WRITE 'LANG(1.1:10) :' LANG (1.1:10) /
        'LANG(1.I:I+2):' LANG (1.I:I+2) //
  WRITE 'LANG(5.1:5)  :' LANG (5.1:5)  /
        'LANG(5.J)    :' LANG (5.J)
LOOP
END
```

ストラクチャードモードの例：

```
** Example 'INDEX5S': Array definition with constants (structured mode)
**                    (multiple definition of same database field)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 LANG (1:10)
  2 LANG (5:10)
*
1 I (I2)
1 J (I2)
END-DEFINE
*
*
I := 1
J := 2
*
READ (1) EMPLOY-VIEW WITH NAME = 'WINTER' WHERE CITY = 'LONDON'
  WRITE 'LANG(1.1:10) :' LANG (1.1:10) /
        'LANG(1.I:I+2):' LANG (1.I:I+2) //
  WRITE 'LANG(5.1:5)  :' LANG (5.1:5)  /
        'LANG(5.J)    :' LANG (5.J)
END-READ
END
```

インデックス変数を使用してマルチプルバリューフィールドまたはピリオディックグループフィールドが定義されている場合も、同様の構文を使用します。

レポーティングモードの例：

```
** Example 'INDEX6R': Array definition with variables (reporting mode)
**                   (multiple definition of same database field)
************************************************************************
DEFINE DATA LOCAL
1 I (I2) INIT <1>
1 J (I2) INIT <2>
1 N (I2) INIT <1>
1 EMPLOY-VIEW VIEW OF EMPLOYEES    /* For reporting mode programs
  2 NAME                           /* DEFINE DATA is recommended
  2 CITY                           /* to use multiple definitions
  2 LANG (I:I+10)                  /* of same database field
  2 LANG (J:J+5)
  2 LANG (4:5)
*
END-DEFINE
*
READ (1) EMPLOY-VIEW WITH NAME = 'WINTER' WHERE CITY = 'LONDON'
*
  WRITE 'LANG(I.I)    :' LANG (I.I) /
        'LANG(1.I:I+2):' LANG (I.I:I+10) //
*
  WRITE 'LANG(J.N)    :' LANG (J.N) /
        'LANG(J.2:4)  :' LANG (J.2:4) //
*
  WRITE 'LANG(4.N)    :' LANG (4.N) /
        'LANG(4.N:N+1):' LANG (4.N:N+1) /
LOOP
END
```

ストラクチャードモードの例：

```
** Example 'INDEX6S': Array definition with variables (structured mode)
**                   (multiple definition of same database field)
************************************************************************
DEFINE DATA LOCAL
1 I (I2) INIT <1>
1 J (I2) INIT <2>
1 N (I2) INIT <1>
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 LANG (I:I+10)
  2 LANG (J:J+5)
  2 LANG (4:5)
*
END-DEFINE
*
READ (1) EMPLOY-VIEW WITH NAME = 'WINTER' WHERE CITY = 'LONDON'
*
  WRITE 'LANG(I.I)    :' LANG (I.I) /
```

```
        'LANG(1.I:I+2):' LANG (I.I:I+10) //
*
  WRITE 'LANG(J.N)    :' LANG (J.N) /
        'LANG(J.2:4)  :' LANG (J.2:4) //
*
  WRITE 'LANG(4.N)    :' LANG (4.N) /
        'LANG(4.N:N+1):' LANG (4.N:N+1) /
END-READ
END
```

データベース配列の内部カウントの参照（C* 表記）

レコードにいくつの値／オカレンスが存在するかが不明なマルチプルバリューフィールドやピリオディックグループの参照が必要になることがあります。 Adabas は、各マルチプルバリューフィールドの値の個数および各ピリオディックグループのオカレンス数の内部カウントを管理します。 このカウントは、フィールド名の直前に C* を指定することにより参照できます。

Adabas 以外のデータベースに関する注意事項：

Tamino	XML データベースでは、C* 表記は使用できません。
SQL	SQL データベースでは、C* 表記は使用できません。

『エディタ』ドキュメントに記載されている、データエリア行コマンド .* の説明も参照してください。

C* フィールドで明示的に宣言できるフォーマットおよび長さは、以下のいずれかです。

- 2 バイト（I2）または 4 バイト（I4）の長さの整数値（I）
- 整数の桁数のみ（精度なし）が指定されている数値（N）またはパック型（P）。例：(N3)。

フォーマットおよび長さが明示的に指定されていない場合、フォーマット／長さ（N3）がデフォルトとして使用されます。

例：

C*LANG	マルチプルバリューフィールド LANG の値の個数を返します。
C*INCOME	ピリオディックグループ INCOME のオカレンス数を返します。
C*BONUS(1)	ピリオディックグループの最初のオカレンスのマルチプルバリューフィールド BONUS 値の個数を返します（BONUSはピリオディックグループ内のマルチプルバリューフィールドとします）。

C* 変数を使用したプログラム例：

```
** Example 'CNOTX01': C* Notation
************************************************************************
DEFINE DATA LOCAL
1 EMPL-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 C*INCOME
  2 INCOME
    3 SALARY  (1:5)
    3 C*BONUS (1:2)
    3 BONUS   (1:2,1:2)
  2 C*LANG
  2 LANG      (1:2)
*
1 #I (N1)
END-DEFINE
*
LIMIT 2
READ EMPL-VIEW BY CITY
  /*
  WRITE NOTITLE 'NAME:' NAME /
       'NUMBER OF LANGUAGES SPOKEN:' C*LANG 5X
       'LANGUAGE 1:' LANG (1) 5X
       'LANGUAGE 2:' LANG (2)
  /*
  WRITE 'SALARY DATA:'
  FOR #I FROM 1 TO C*INCOME
    WRITE 'SALARY' #I SALARY (1.#I)
  END-FOR
  /*
  WRITE 'THIS YEAR BONUS:' C*BONUS(1)  BONUS (1,1) BONUS (1,2)
      / 'LAST YEAR BONUS:' C*BONUS(2)  BONUS (2,1) BONUS (2,2)
  SKIP 1
END-READ
END
```

プログラム CNOTX01 の出力：

```
NAME: SENKO
NUMBER OF LANGUAGES SPOKEN:    1     LANGUAGE 1: ENG     LANGUAGE 2:
SALARY DATA:
SALARY  1      36225
SALARY  2      29900
SALARY  3      28100
SALARY  4      26600
SALARY  5      25200
THIS YEAR BONUS:    0          0          0
LAST YEAR BONUS:    0          0          0

NAME: CANALE
```

```
NUMBER OF LANGUAGES SPOKEN:    2     LANGUAGE 1: FRE     LANGUAGE 2: ENG
SALARY DATA:
SALARY  1     202285
THIS YEAR BONUS:    1      23000          0
LAST YEAR BONUS:    0          0          0
```

ピリオディックグループ内のマルチプルバリューフィールドに対する C*

ピリオディックグループ内のマルチプルバリューフィールドに対し、インデックス範囲を使用して C* 変数を定義することもできます。

以下の例は、ピリオディックグループ INCOME の一部分であるマルチプルバリューフィールド BONUS を使用しています。 以下の 3 つの例は、いずれも同じ結果になります。

例 **1 -** レポーティングモード：

```
** Example 'CNOTXO2': C* Notation (multiple-value fields)
************************************************************************
*
LIMIT 2
READ EMPLOYEES BY CITY
  OBTAIN C*BONUS (1:3)
         BONUS   (1:3,1:3)
  /*
  DISPLAY NAME C*BONUS (1:3) BONUS (1:3,1:3)
LOOP
*
END
```

例 **2 -** ストラクチャードモード：

```
** Example 'CNOTXO3': C* Notation (multiple-value fields)
************************************************************************
DEFINE DATA LOCAL
1 EMPL-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 INCOME    (1:3)
    3 C*BONUS
    3 BONUS   (1:3)
END-DEFINE
*
LIMIT 2
READ EMPL-VIEW BY CITY
  /*
  DISPLAY NAME C*BONUS (1:3) BONUS (1:3,1:3)
END-READ
```

```
*
END
```

例 **3 -** ストラクチャードモード：

```
** Example 'CNOTX04': C* Notation (multiple-value fields)
************************************************************************
DEFINE DATA LOCAL
1 EMPL-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 C*BONUS  (1:3)
  2 INCOME   (1:3)
    3 BONUS  (1:3)
END-DEFINE
*
LIMIT 2
READ EMPL-VIEW BY CITY
  /*
  DISPLAY NAME C*BONUS (*) BONUS (*,*)
END-READ
*
END
```

Caution: Adabas フォーマットバッファではカウントフィールドの範囲指定が認められていないため、個別のフィールドとして生成されます。したがって、大規模な配列に対して C* インデックス範囲を指定すると、Adabas フォーマットバッファがオーバーフローする可能性があります。

データ構造の条件指定

参照するフィールドを指定するために、そのフィールドを修飾できます。つまり、フィールド名の前に、そのフィールドが配置されているレベル 1 のデータ要素名とピリオドを指定します。

フィールド名によってフィールドを一意に識別できない場合（複数のグループ／ビューで同じ名前のフィールド名が使用されている場合など）、参照するフィールドを修飾する必要があります。

レベル 1 のデータ要素とフィールド名の組み合わせは一意である必要があります。

例：

```
DEFINE DATA LOCAL
1 FULL-NAME
  2 LAST-NAME (A20)
  2 FIRST-NAME (A15)
1 OUTPUT-NAME
  2 LAST-NAME (A20)
  2 FIRST-NAME (A15)
END-DEFINE
...
MOVE FULL-NAME.LAST-NAME TO OUTPUT-NAME.LAST-NAME
...
```

修飾語はレベル 1 のデータ要素である必要があります。

例：

```
DEFINE DATA LOCAL
1 GROUP1
  2 SUB-GROUP
    3 FIELD1 (A15)
    3 FIELD2 (A15)
END-DEFINE
...
MOVE 'ABC' TO GROUP1.FIELD1
...
```

データベースフィールドの条件指定

同じ名前のユーザー定義変数とデータベースフィールドを使用している場合（好ましくない状態）、参照するときにデータベースフィールドを修飾する必要があります。

Caution: 参照するときにデータベースフィールドを修飾しない場合、代わりにユーザー定義変数が参照されます。

ユーザー定義変数の例

```
DEFINE DATA LOCAL
1 #A1 (A10)       /* Alphanumeric, 10 positions.
1 #A2 (B4)        /* Binary, 4 positions.
1 #A3 (P4)        /* Packed numeric, 4 positions and 1 sign position.
1 #A4 (N7.2)      /* Unpacked numeric,
                  /* 7 positions before and 2 after decimal point.
1 #A5 (N7.)       /* Invalid definition!!!
1 #A6 (P7.2)     /* Packed numeric, 7 positions before and 2 after decimal point
                  /* and 1 sign position.
```

```
1 #INT1 (I1)      /* Integer, 1 byte.
1 #INT2 (I2)      /* Integer, 2 bytes.
1 #INT3 (I3)      /* Invalid definition!!!
1 #INT4 (I4)      /* Integer, 4 bytes.
1 #INT5 (I5)      /* Invalid definition!!!
1 #FLT4 (F4)      /* Floating point, 4 bytes.
1 #FLT8 (F8)      /* Floating point, 8 bytes.
1 #FLT2 (F2)      /* Invalid definition!!!
1 #DATE (D)       /* Date (internal format/length P6).
1 #TIME (T)       /* Time (internal format/length P12).
1 #SWITCH (L)     /* Logical, 1 byte (TRUE or FALSE).
                  /*
END-DEFINE
```

18 ファンクションコール

```
call-name(<([prototype-cast][intermediate-result-definition])[parameter][,[parameter
]]...>)
```

このchapterでは、次のトピックについて説明します。

関連トピック：「DEFINE FUNCTION」、「DEFINE PROTOTYPE」

ユーザー定義関数の呼び出し

ファンクションコールは、タイプ **function** の特殊オブジェクト内で定義されているユーザー定義関数を呼び出すために使用できます。

ファンクションコールには、以下の方法があります。

- 記号ファンクションコール
- 変数を使用したファンクションコール

記号ファンクションコール

記号ファンクションコールを使用する場合は、ランタイムに実行する正確なファンクション名を指定します。

Naturalソース内で記号ファンクションコールのみを指定すると、適切なプロトタイプ定義がすでに指定されていない限り、対応する Natural ファンクション定義が自動的に取得されます。Naturalファンクション定義が含まれているオブジェクトの名前は、記号論理ファンクション名に従って取得されます。 これは、*FILEDIR.SAG* ファイルのリンクレコードを使用して実行します。この場合、リンクレコードを最初に生成する前に、対応するファンクション定義をStowしておく必要があります。

この機能では、Natural ファンクションコールのすべてのパラメータ定義に対し、常にフォーマット／長さ定義の有効性チェックが行われます。

変数を使用したファンクションコール

変数を使用したファンクションコールでは、必要なファンクション定義名を英数字変数に格納します。 ランタイム時は、対応するファンクション定義、つまり変数内に格納されている名前の定義に、Natural は 処理を飛ばします。

これらの2種類のファンクションコールを識別するために、対応するプロトタイプ定義を指定する必要があります。 また、プロトタイプにはファンクション定義の署名全体を指定できます。署名が指定されていない場合、署名の足りない部分を指定するために、ファンクションコールで PT 節を指定する必要があります。したがって、PT 節内で指定されているプロトタイプの VARIABLE キーワードは効力を持ちません。 変数ファンクションコールの場合、ファンクション名を格納する英数字変数と同じ名前の有効なプロトタイプが存在する必要があります。

プロトタイプがファンクションコールに割り当てられていない場合、コンパイル時に戻り値のフォーマット／長さを定義するために、特殊な *prototype-cast* を指定する必要があります。*prototype-cast* およびパラメータリストは、構文図に示されているように、山カッコで囲む必要があります。

変数による方法を使用する場合、キーワード VARIABLE を使用して *variable-name* と同じ名前でプロトタイプを定義する必要があります。

例：

```
DEFINE PROTOTYPE VARIABLE variable-name
```

Note: オペランドを変更できない場合にのみ、ファンクションコールを使用できます。ただし、ファンクションコールが INPUT ステートメント内で使用されている場合、戻り値は「出力専用」 フィールド（AD=O）として表示されます。

制限事項

ファンクションコールは、以下の状況には使用できません。

- DEFINE DATA ステートメント内
- データベースアクセスまたは更新ステートメント内（READ、FIND、SELECT、UPDATE、STOREなど）
- AT BREAK ステートメントまたは IF BREAK ステートメント内
- システム関数 AVER、COUNT、MAX、MIN、NAVER、NCOUNT、NMIN、OLD、SUM、TOTAL の引数
- インデックス表記

構文説明

ファンクションコールは、以下の構文要素によって構成されます。

- *call-name*
- *prototype-cast*
- *intermediate-result-definition*

- *parameter*

call-name

```
{ function-name          }
{ prototype-variable-name }
```

オペランド定義テーブル：

オペランド	構文要素	フォーマット	ステートメント参照	ダイナミック定義
prototype-variable-name	S A	A	可	不可

構文要素の説明：

function-name	*function-name* 節は、記号ファンクション名です。 対応するファンクション定義は、特定のファンクションオブジェクトファイルに定義します。
prototype-variable-name	*prototype-variable-name* は、呼び出すファンクションの実際の名前が格納されている変数名です。同じ名前の英数字変数がすでに定義されている必要があります。

prototype-cast

```
PT= { prototype-name          }
    { prototype-variable-name }
```

prototype-cast は、対応するファンクションプロトタイプに署名が指定されていないファンクションコールに対して使用する必要があります（プロトタイプ定義の signature 節が UNKNOWN として定義されている場合など）。

intermediate-result-definition

```
IR= { format-length [/array-definition]                                  }
    { ( { A } [/array-definition] } DYNAMIC )                            }
    { ( { B }                     }                                      }
    { ( { U }                     }                                      }
```

この節を使用すると、明示的または暗黙的にプロトタイプ定義を使用しなくても、ファンクションコールの戻り値の *format-length*/*array-definition* を指定できます。つまり、中間結果を明示的に指定できます。

さらに、プロトタイプがファンクションコールに対して有効な場合、ファンクション定義の戻り値の *format-length*/*array-definition* が中間結果と MOVE での互換性があるかどうかが確認されます。 互換性がない場合はエラーになり、 中間結果が戻り値として使用されます。

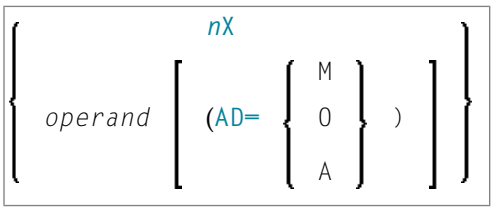
また、戻り値として配列を使用できます。つまり、配列定義を中間結果として指定できます。*array-definition* には、配列定義の次元の下限と上限を定義します。 『ステートメント』ドキュメントの「*配列の次元の定義*」を参照してください。

format-length	フィールドのフォーマットおよび長さ。ユーザー定義変数のフォーマットおよび長さの定義については、「*ユーザー定義変数のフォーマットおよび長さ*」を参照してください。
A、B、U	ダイナミック変数に対するデータタイプ：英数字、バイナリ、または Unicode
array-definition	*array-definition* には、配列定義の次元の下限と上限を定義します。 『ステートメント』ドキュメントの「*配列の次元の定義*」を参照してください。
DYNAMIC	フィールドは、DYNAMIC として定義できます。 ダイナミック変数の処理の詳細については、「*ダイナミック変数およびフィールドについて*」を参照してください。

parameter

各パラメータは、ファンクションコール時にオペランドとして使用できます。 サブプログラムの DEFINE DATA PARAMETER 内でキーワード OPTIONAL を使用してパラメータが定義されている場合、対応するオペランドの値をファンクションコールで省略できます。 この場合、*n*X 表記（*n* は 1 以上の整数）を使用するか、または単にこの引数を省略します。

セッションパラメータ AD は、引数ごとに指定できます。



オペランド定義テーブル：

オペランド	構文要素					フォーマット												ステートメント参照	ダイナミック定義
operand	C	S	A	G		A	N	P	I	F	B	D	T	L	C	G	O	可	可

ファンクションコールの適切な使用例については、DEFINE PROTOTYPE ステートメントの説明にある例を参照してください。

***n*X**	省略されるパラメータ： 表記 *n*X を使用して、次の *n* 個のパラメータをスキップするように指定することができます（例えば、1X を使用すると次のパラメータがスキップされ、3X を使用すると次の 3 つのパラメータがスキップされます）。これは、次の *n* 個のパラメータでは、値がサブプログラムに渡されないことを意味します。 スキップするパラメータは、サブプログラムの DEFINE DATA PARAMETER ステートメント内のキーワード OPTIONAL を使用して定義する必要があります。 OPTIONAL は、値を呼び出し側オブジェクトからこのようなパラメータに渡すこともできるということを意味します。	
AD=	属性定義： *operand* が変数の場合は、次のいずれかの方法でマークすることができます。	
	AD=O	変更不可。セッションパラメータ AD=O を参照してください。 **注意:** 内部的に、AD=O は BY VALUE と同様に処理されます（DEFINE DATA ステートメントの説明にある *parameter-data-definition* を参照）。
	AD=M	変更可。セッションパラメータ AD=M を参照してください。 これはデフォルト設定です。
	AD=A	入力のみ。セッションパラメータ AD=A を参照してください。
	operand が定数の場合は、AD を明示的に指定することはできません。 定数には常に AD=O が適用されます。	

19 ダイナミック変数およびフィールドについて

このchapterでは、次のトピックについて説明します。

ダイナミック変数の用途

アプリケーションの開発時に大きなデータ構造（画像、音声、ビデオなど）の最大サイズが正確にわからない場合があるため、Naturalでは、英数字変数およびバイナリ変数をDYNAMICという属性を使用して定義できるようにしています。この属性で定義されている変数の値スペースは、実行時に必要となったとき（割り当て操作#picture1 := #picture2を実行するときなど）にダイナミックに拡張されます。つまり、Naturalでは、開発時に制限を定義せずに、大きなバイナリデータ構造および英数字データ構造を処理できます。ダイナミック変数の割り当て時には当然、使用可能なメモリ量の制限を受けます。ダイナミック変数の割り当てに対し、オペレーティングシステムからメモリ不足という結果が返された場合、ON ERRORステートメントを使用してこのエラー条件をインターセプトできます。インターセプトしない場合、Naturalによってエラーメッセージが返されます。

Naturalシステム変数*LENGTHは、指定されたダイナミック変数に現在使用されている値スペースの長さ（コード単位）を取得するために使用できます。AおよびBフォーマットでは、1つのコード単位のサイズは1バイトです。Uフォーマットでは、1つのコード単位のサイズは2バイトです（UTF-16）。*LENGTHは、ダイナミック変数を使用する割り当て中に、Naturalによってソースオペランドの長さに自動的に設定されます。したがって、*LENGTH(*field*)は、Naturalのダイナミックフィールドまたはダイナミック変数に現在使用されている長さ（コード単位）を返します。

ダイナミック変数のスペースが不要になった場合、REDUCEステートメントまたはRESIZEステートメントを使用して、ダイナミック変数に使用されていたスペースをゼロ（または他の任意のサイズ）に減らすことができます。特定のダイナミック変数に対するメモリの使用量がわかっている場合、EXPANDステートメントを使用して、ダイナミック変数に使用するスペースをその特定のサイズに設定できます。

ダイナミック変数を初期化する必要がある場合、MOVE ALL UNTILステートメントを使用して初期化できます。

ダイナミック変数の定義

アプリケーションの開発時に大きな英数字データ構造またはバイナリデータ構造の実際のサイズが正確にわからない場合があるため、フォーマットA、B、Uのダイナミック変数の定義を、これらの構造を管理するために使用できます。ラージ変数のダイナミックアロケーションおよび拡張（再割り当て）は、アプリケーションプログラミングロジックに対して透過的です。ダイナミック変数は長さを指定せずに定義されます。メモリは、実行時にダイナミック変数がターゲットオペランドとして使用されるときに暗黙的に割り当てられるか、EXPANDステートメントまたはRESIZEステートメントの使用によって明示的に割り当てられます。

ダイナミック変数は、以下の構文を使用して、DEFINE DATA ステートメント内でのみ定義できます。

```
level variable-name( A ) DYNAMIC
level variable-name( B ) DYNAMIC
level variable-name( U ) DYNAMIC
```

以下の制限が、ダイナミック変数に適用されます。

- ダイナミック変数は再定義できません。
- ダイナミック変数は REDEFINE 節には使用できません。

ダイナミック変数の現在の値スペース

ダイナミック変数の現在の値スペースの長さ（コード単位）は、システム変数 *LENGTH から取得できます。 *LENGTH は、割り当て時に自動的に、ソースオペランドの（使用されている）長さに設定されます。

Caution: パフォーマンスを考慮して、ダイナミック変数の値を保持するために割り当てられているストレージエリアが *LENGTH の値（プログラマが使用できるサイズ）より大きい場合があります。 *LENGTH が示す使用長を超えて割り当てられているストレージは、対応するダイナミック変数にアクセスしていなくても解放される可能性が常にあるため、信頼しないでください。 現在割り当てられているサイズに関する情報を取得する方法はありません。 これは、内部値です。

*LENGTH(*field*) は、Natural のダイナミックフィールドまたはダイナミック変数の使用長（コード単位）を返します。 A および B フォーマットでは、1 つのコード単位のサイズは 1 バイトです。U フォーマットでは、1つのコード単位のサイズは2バイトです（UTF-16）。 *LENGTH は、ダイナミック変数の現在の使用長を取得するためにのみ使用します。

サイズ制限チェック

プロファイルパラメータ **USIZE**

ダイナミック変数の場合、長さが定義されていないため、コンパイル時にサイズ制限チェックを行うことができません。 ユーザーバッファエリアのサイズ（USIZE）は、仮想メモリ内のユーザーバッファのサイズを示します。ユーザーバッファには、Natural によってダイナミックに割り当てられているすべてのデータが含まれます。 実行時にダイナミック変数の割り当てまたは拡張が行われて USIZE の制限を超えると、エラーメッセージが返されます。

ダイナミック変数のメモリスペースの割り当て／解放

ステートメント EXPAND、REDUCE、および RESIZE は、ダイナミック変数のメモリスペースを明示的に割り当てたり解放したりするために使用します。

構文：

```
EXPAND [SIZE OF] DYNAMIC [VARIABLE] operand1 TO operand2
REDUCE [SIZE OF] DYNAMIC [VARIABLE] operand1 TO operand2
RESIZE [SIZE OF] DYNAMIC [VARIABLE] operand1 TO operand2
```

- *operand1* はダイナミック変数、*operand2* は 0 以上の整数のサイズ値です。

EXPAND

機能

EXPAND ステートメントは、割り当てられているダイナミック変数（*operand1*）の長さを指定された長さ（*operand2*）に増やすために使用します。

指定されたサイズの変更

ダイナミック変数の現在の使用長（Natural システム変数 *LENGTH によって示される。上記参照）は変更されません。

指定された長さ（*operand2*）が、ダイナミック変数に割り当てられている長さに満たない場合、このステートメントは無視されます。

REDUCE

機能

REDUCE ステートメントは、割り当てられているダイナミック変数（*operand1*）の長さを指定された長さ（*operand2*）に減らすために使用します。

指定された長さ（*operand2*）を超える、ダイナミック変数（*operand1*）に割り当てられたストレージは、ステートメントの実行時または実行後に随時解放されます。

指定された長さの変更

ダイナミック変数の現在の使用長（Natural システム変数 *LENGTH によって示される。上記参照）が指定された長さ（*operand2*）を超えている場合、このダイナミック変数の *LENGTH は指定された長さに設定されます。 変数の内容は切り捨てられますが、変更は行われません。

指定された長さが、現在割り当てられているダイナミック変数のストレージを超えている場合、このステートメントは無視されます。

RESIZE

機能

RESIZE ステートメントは、現在割り当てられているダイナミック変数（*operand1*）の長さを指定された長さ（*operand2*）に調整するために使用します。

指定された長さの変更

指定された長さが、ダイナミック変数の使用長（Natural システム変数 *LENGTH によって示される。上記参照）に満たない場合、必要に応じて、ダイナミック変数の使用長が縮小されます。

指定された長さが、現在割り当てられているダイナミック変数の長さを超えている場合、ダイナミック変数に割り当てられる長さが増やされます。 *LENGTH によって示される、ダイナミック変数の現在の使用長は影響を受けず、そのままになります。

指定された長さが、現在割り当てられているダイナミック変数の長さと同じ場合、RESIZE ステートメントを実行しても効力はありません。

20 ダイナミック変数およびラージ変数の使用

このchapterでは、次のトピックについて説明します。

全般的な注意事項

通常、次の規則が適用されます。

- ダイナミック英数字フィールドは、英数字フィールドを使用できる場所であればどこにでも使用できます。
- ダイナミックバイナリフィールドは、バイナリフィールドを使用できる場所であればどこにでも使用できます。
- ダイナミック Unicode フィールドは、Unicode フィールドを使用できる場所であればどこにでも使用できます。

例外

ダイナミック変数は、SORT ステートメント内では使用できません。ダイナミック変数を DISPLAY、WRITE、PRINT、REINPUT、INPUT の各ステートメント内で使用するには、セッションパラメータ AL または EM のいずれかを使用して変数の長さを定義する必要があります。

ダイナミック変数に割り当てられているストレージの使用長（Natural システム変数 *LENGTH によって示される。「*ダイナミック変数の現在の値スペース*」を参照）およびサイズは、その変数がターゲットオペランドとして最初にアクセスされるまではゼロです。 割り当て操作またはその他の操作によって、ダイナミック変数はソースオペランドの正確なサイズに初めて割り当てまたは拡張（再割り当て）されます。

以下のステートメントで変更可能なオペランド（ターゲットオペランド）としてダイナミック変数を使用する場合、ダイナミック変数のサイズを拡張できます。

ASSIGN	*operand1*（割り当て内の応答先オペランド）
CALLNAT	「*ダイナミック変数を使用したパラメータ引き渡し*」を参照してください（AD=O、または 対応する PDA 内に BY VALUE が存在する場合を除く）。
COMPRESS	*operand2*。「*処理*」を参照してください。
EXAMINE	DELETE REPLACE 節内の *operand1*
MOVE	*operand2*（応答先オペランド）。「*機能*」を参照してください。
PERFORM	（AD=O、または対応する PDA 内に BY VALUE が存在する場合以外）
READ WORK FILE	*operand1* および *operand2*。「*ラージおよびダイナミック変数の処理*」を参照してください。
SEPARATE	*operand4*.
SELECT (SQL)	INTO 節内のパラメータ
SEND METHOD	*operand3*（AD=O でない場合）

現時点では、ラージ変数の使用には以下の制限があります。

CALL	各パラメータのサイズは64 KB未満です（INTERFACE4オプションを使用したCALLには制限はありません）。

以下のセクションでは、ダイナミック変数の使用について、例を使用してさらに詳しく説明します。

ダイナミック変数を使用した割り当て

通常は、ソースオペランドの現在の使用長（Naturalシステム変数*LENGTHによって示される）で割り当てが行われます。 応答先オペランドがダイナミック変数の場合、ソースオペランドを切り捨てずに移動できるよう、現在割り当てられているサイズが拡張される場合があります。

例：

```
#MYDYNTEXT1 := OPERAND
MOVE OPERAND TO #MYDYNTEXT1
/* #MYDYNTEXT1 IS AUTOMATICALLY EXTENDED UNTIL THE SOURCE OPERAND CAN BE COPIED
```

MOVE ALL、およびダイナミックターゲットオペランドを使用したMOVE ALL UNTILは、以下のように定義されています。

- MOVE ALLでは、ターゲットオペランドの使用長（*LENGTH）に到達するまで、ソースオペランドをターゲットオペランドに繰り返し移動します。 *LENGTHは変更できません。 *LENGTHがゼロの場合、このステートメントは無視されます。
- MOVE ALL *operand1* TO *operand2* UNTIL operand3では、*operand3*に指定した長さに到達するまで、*operand1*を*operand2*に繰り返し移動します。 *operand3*が*LENGTH(*operand2*)より大きい場合、*operand2*は拡張され、*LENGTH(*operand2*)は*operand3*に設定されます。 *operand3*が*LENGTH(*operand2*)より小さい場合、使用長は*operand3*に縮小されます。 *operand3*が*LENGTH(*operand2*)と同じ場合、MOVE ALLと同じ動作をします。

例：

```
#MYDYNTEXT1 := 'ABCDEFGHIJKLMNO'          /* *LENGTH(#MYDYNTEXT1) = 15
MOVE ALL 'AB' TO #MYDYNTEXT1              /* CONTENT OF #MYDYNTEXT1 =
'ABABABABABABABA';
                                          /* *LENGTH IS STILL 15
MOVE ALL 'CD' TO #MYDYNTEXT1 UNTIL 6      /* CONTENT OF #MYDYNTEXT1 = 'CDCDCD';
                                          /* *LENGTH = 6
```

```
MOVE ALL 'EF' TO #MYDYNTEXT1 UNTIL 10     /* CONTENT OF #MYDYNTEXT1 = 'EFEFEFEFEF';
                                            /* *LENGTH = 10
```

ターゲットオペランドがダイナミック変数の場合、MOVE JUSTIFIED はコンパイル時に拒否されます。

MOVE SUBSTR および MOVE TO SUBSTR は実行できます。 ダイナミック変数の使用長（*LENGTH）を超えるサブストリングを参照すると、MOVE SUBSTR はランタイムエラーになります。 *LENGTH + 1 を超えるサブストリングの位置を参照すると、ダイナミック変数の内容に未定義のギャップが生じるため、MOVE TO SUBSTR はランタイムエラーになります。 ターゲットオペランドを MOVE TO SUBSTR で拡張する必要がある場合（第 2 オペランドが *LENGTH+1 に設定されている場合など）、第 3 オペランドは必須です。

有効な構文：

```
#OP2 := *LENGTH(#MYDYNTEXT1)
MOVE SUBSTR (#MYDYNTEXT1, #OP2) TO OPERAND                 /* MOVE LAST CHARACTER
TO OPERAND
#OP2 := *LENGTH(#MYDYNTEXT1) + 1
MOVE OPERAND TO SUBSTR(#MYDYNTEXT1, #OP2, #LEN_OPERAND)   /* CONCATENATE OPERAND
TO #MYDYNTEXT1
```

無効な構文：

```
#OP2 := *LENGTH(#MYDYNTEXT1) + 1
MOVE SUBSTR (#MYDYNTEXT1, #OP2, 10) TO OPERAND          /* LEADS TO RUNTIME ERROR;
UNDEFINED SUB-STRING
#OP2 := *LENGTH(#MYDYNTEXT1 + 10)
MOVE OPERAND TO SUBSTR(#MYDYNTEXT1, #OP2, #EN_OPERAND)    /* LEADS TO RUNTIME ERROR;
UNDEFINED GAP
#OP2 := *LENGTH(#MYDYNTEXT1) + 1
MOVE OPERAND TO SUBSTR(#MYDYNTEXT1, #OP2)               /* LEADS TO RUNTIME ERROR;
UNDEFINED LENGTH
```

割り当ての互換性

例：

```
#MYDYNTEXT1   := #MYSTATICVAR1
#MYSTATICVAR1 := #MYDYNTEXT2
```

ソースオペランドがスタティック変数の場合、ダイナミックな応答先オペランドの使用長（*LENGTH(#MYDYNTEXT1)）はスタティック変数のフォーマット長に設定され、末尾の空白（フォーマット A および U）またはバイナリゼロ（フォーマット B）を含め、この長さでソースオペランドがコピーされます。

応答先オペランドがスタティックでソースオペランドがダイナミックの場合、ダイナミック変数は現在の使用長でコピーされます。 この長さがスタティック変数のフォーマット長より短い場合、残りの部分は空白（英数字フィールドおよび Unicode フィールドの場合）またはバイナリゼロ（バイナリフィールドの場合）で埋められます。そうでない場合、値は切り捨てられます。ダイナミック変数の現在の使用長が0の場合、スタティックのターゲットオペランドは空白（英数字フィールドおよび Unicode フィールドの場合）またはバイナリゼロ（バイナリフィールドの場合）で埋められます。

ダイナミック変数の初期化

ダイナミック変数は、RESET ステートメントを使用して、最大で現在の使用長（= *LENGTH）まで、空白（英数字フィールドおよび Unicode フィールドの場合）またはゼロ（バイナリフィールドの場合）で初期化できます。 *LENGTH は変更できません。

例：

```
DEFINE DATA LOCAL
  1 #MYDYNTEXT1   (A)  DYNAMIC
END-DEFINE
#MYDYNTEXT1 := 'SHORT TEXT'
WRITE *LENGTH(#MYDYNTEXT1)          /* USED LENGTH = 10
RESET #MYDYNTEXT1                   /* USED LENGTH = 10, VALUE = 10 BLANKS
```

ダイナミック変数を特定のサイズと値で初期化するには、MOVE ALL UNTIL ステートメントを使用します。

例：

```
MOVE ALL 'Y' TO #MYDYNTEXT1 UNTIL 15         /* #MYDYNTEXT1 CONTAINS 15 'Y'S, USED
LENGTH = 15
```

ダイナミック英数字変数での文字列操作

変更可能なオペランドがダイナミック変数の場合、切り捨てまたはエラーメッセージを発生させずに操作できるよう、現在割り当てられているサイズが拡張される場合があります。 これは、ダイナミック変数の連結（COMPRESS）および分割（SEPARATE）で有効です。

例：

```
** Example 'DYNAMX01': Dynamic variables (with COMPRESS and SEPARATE)
************************************************************************
DEFINE DATA LOCAL
1 #MYDYNTEXT1 (A)   DYNAMIC
1 #TEXT       (A20)
1 #DYN1       (A)   DYNAMIC
1 #DYN2       (A)   DYNAMIC
1 #DYN3       (A)   DYNAMIC
END-DEFINE
*
MOVE ' HELLO WORLD ' TO #MYDYNTEXT1
WRITE #MYDYNTEXT1 (AL=25) 'with length' *LENGTH (#MYDYNTEXT1)
/*  dynamic variable with leading and trailing blanks
*
MOVE ' HELLO WORLD ' TO #TEXT
*
MOVE #TEXT TO #MYDYNTEXT1
WRITE #MYDYNTEXT1 (AL=25) 'with length' *LENGTH (#MYDYNTEXT1)
/*  dynamic variable with whole variable length of #TEXT
*
COMPRESS #TEXT INTO #MYDYNTEXT1
WRITE #MYDYNTEXT1 (AL=25) 'with length' *LENGTH (#MYDYNTEXT1)
/*  dynamic variable with leading blanks of #TEXT
*
*
#MYDYNTEXT1 := 'HERE COMES THE SUN'
SEPARATE #MYDYNTEXT1 INTO #DYN1 #DYN2 #DYN3 IGNORE
*
WRITE / #MYDYNTEXT1 (AL=25) 'with length' *LENGTH (#MYDYNTEXT1)
WRITE #DYN1 (AL=25) 'with length' *LENGTH (#DYN1)
WRITE #DYN2 (AL=25) 'with length' *LENGTH (#DYN2)
WRITE #DYN3 (AL=25) 'with length' *LENGTH (#DYN3)
/*  #DYN1, #DYN2, #DYN3 are automatically extended or reduced
```

```
*
EXAMINE #MYDYNTEXT1 FOR 'SUN' REPLACE 'MOON'
WRITE / #MYDYNTEXT1 (AL=25) 'with length' *LENGTH (#MYDYNTEXT1)
/*  #MYDYNTEXT1 is automatically extended or reduced
*
END
```

Note: 非ダイナミック変数の場合、エラーメッセージが返されます。

ダイナミック変数を使用した論理条件の基準（LCC）

ダイナミック変数を使用して読み取り専用の操作（比較など）を行う場合、通常は現在の使用長を使用して実行されます。 読み取り操作（変更なし）で使用される場合、ダイナミック変数はスタティック変数と同様に処理されます。

例：

```
IF  #MYDYNTEXT1  =  #MYDYNTEXT2  OR  #MYDYNTEXT1 =  "**"  THEN ...
IF  #MYDYNTEXT1  <  #MYDYNTEXT2  OR  #MYDYNTEXT1 <  "**"  THEN ...
IF  #MYDYNTEXT1  >  #MYDYNTEXT2  OR  #MYDYNTEXT1 >  "**"  THEN ...
```

英数字変数および Unicode 変数の末尾の空白またはバイナリ変数の先頭のバイナリゼロは、スタティック変数とダイナミック変数で同様に処理されます。 例えば、AA および空白が続く AA を値として持つ英数字変数は同一とみなされ、H'00003031' および H'3031' という値を持つバイナリ変数は同一とみなされます。 値が完全に同じ場合にのみ比較結果を TRUE とする場合は、ダイナミック変数の使用長も比較する必要があります。 一方の変数ともう一方の変数の値が完全に同じであれば、その使用長もまた同じです。

例：

```
#MYDYNTEXT1  :=  'HELLO'                                    /* USED LENGTH IS 5
#MYDYNTEXT2  :=  'HELLO     '                               /* USED LENGTH IS 10
IF   #MYDYNTEXT1  =  #MYDYNTEXT2   THEN  ...                /* TRUE
IF   #MYDYNTEXT1  =  #MYDYNTEXT2 AND
     *LENGTH(#MYDYNTEXT1) = *LENGTH(#MYDYNTEXT2)  THEN  ...  /* FALSE
```

2 つのダイナミック変数は、どちらか短い方の使用長に達するまで 1 ポジションずつ比較されます（英数字変数の場合は左から右、バイナリ変数の場合は右から左）。 最初に値が異なったポジションで、1 番目の変数と 2 番目の変数のどちらがより大きいか、小さいか、または同じかが判断されます。 変数の短い方の使用長まで値が同じで、長い方の残りの値がダイナミック英数字変数の場合は空白のみ、ダイナミックバイナリ変数の場合はバイナリゼロのみの場合、それらの変数は同じとみなされます。 2 つのダイナミック Unicode 変数を比較する場合、両方の値の

末尾の空白を削除してから、ICU照合アルゴリズムを使用して2つの値が比較されます。『*Unicode とコードページのサポート*』ドキュメントの「*論理条件の基準*」も参照してください。

例：

```
#MYDYNTEXT1  :=  'HELLO1'                    /* USED LENGTH IS 6
#MYDYNTEXT2  :=  'HELLO2'                    /* USED LENGTH IS 10
IF #MYDYNTEXT1  <  #MYDYNTEXT2 THEN  ...     /* TRUE
#MYDYNTEXT2  :=  'HALLO'
IF #MYDYNTEXT1  >  #MYDYNTEXT2 THEN  ...     /* TRUE
```

比較の互換性

ダイナミック変数とスタティック変数の比較は、ダイナミック変数間の比較と同じです。 スタティック変数のフォーマット長が使用長として使用されます。

例：

```
#MYSTATTEXT1 :=  'HELLO'                      /* FORMAT LENGTH OF MYSTATTEXT1 IS
A20
#MYDYNTEXT1  :=  'HELLO'                      /* USED LENGTH IS 5
IF  #MYSTATTEXT1  =  #MYDYNTEXT1  THEN  ...   /* TRUE
IF  #MYSTATTEXT1  >  #MYDYNTEXT1  THEN  ...   /* FALSE
```

ダイナミックコントロールフィールドの AT／IF-BREAK

ブレイクコントロールフィールドの元の値との比較は、左から右に向かって1ポジションずつ実行されます。 ダイナミック変数の元の値と新しい値で長さが異なる場合、比較するために長さが短い方の値の右側に空白（ダイナミック英数字変数またはダイナミックUnicode変数の場合）またはバイナリゼロ（バイナリ値の場合）が追加されます。

英数字または Unicode のブレイクコントロールフィールドの場合、比較において末尾の空白は意味を持ちません。つまり、末尾の空白は値の変更を意味しないため、ブレイクは発生しません。

バイナリのブレイクコントロールフィールドの場合、比較において末尾のバイナリゼロは意味を持ちません。つまり、末尾のバイナリゼロは値の変更を意味しないため、ブレイクは発生しません。

ダイナミック変数を使用したパラメータ引き渡し

ダイナミック変数は、呼び出されたプログラムオブジェクト（CALLNAT、PERFORM）へのパラメータとして渡すことができます。 ダイナミック変数の値スペースは連続しているので、参照渡しが可能です。 値渡しを使用すると、呼び出し元の変数定義がソースオペランドとして割り当てられ、パラメータ定義が応答先オペランドとして割り当てられます。 また、値渡しの結果では、逆方向にデータが移動します。

参照渡しを使用する場合、変数定義およびパラメータ定義は DYNAMIC である必要があります。 そのうちの 1 つだけが DYNAMIC の場合、ランタイムエラーが発生します。 値渡し（結果）の場合、すべての組み合わせを使用できます。 以下の表に、有効な組み合わせを示します。

参照渡し

呼び出し元	パラメータ	
	スタティック	ダイナミック
スタティック	◯	×
ダイナミック	×	◯

ダイナミック変数 A または B のフォーマットは一致する必要があります。

値渡し（結果）

呼び出し元	パラメータ	
	スタティック	ダイナミック
スタティック	◯	◯
ダイナミック	◯	◯

Note: スタティック／ダイナミック定義またはダイナミック／スタティック定義を使用する場合、割り当てのデータ転送規則によって値が切り捨てられることがあります。

例 **1**：

```
** Example 'DYNAMX02': Dynamic variables (as parameters)
************************************************************************
DEFINE DATA LOCAL
1 #MYTEXT (A) DYNAMIC
END-DEFINE
*
#MYTEXT := '123456'      /* extended to 6 bytes, *LENGTH(#MYTEXT) = 6
*
CALLNAT 'DYNAMX03' USING #MYTEXT
*
```

```
WRITE *LENGTH(#MYTEXT)          /* *LENGTH(#MYTEXT) = 8
*
END
```

サブプログラム DYNAMX03：

```
** Example 'DYNAMX03': Dynamic variables (as parameters)
************************************************************************
DEFINE DATA PARAMETER
1 #MYPARM (A) DYNAMIC  BY VALUE RESULT
END-DEFINE
*
WRITE *LENGTH(#MYPARM)                  /* *LENGTH(#MYPARM) = 6
#MYPARM := '1234567'                    /* *LENGTH(#MYPARM) = 7
#MYPARM := '12345678'                   /* *LENGTH(#MYPARM) = 8
EXPAND DYNAMIC VARIABLE #MYPARM TO 10   /* 10 bytes are allocated
*
WRITE *LENGTH(#MYPARM)                  /* *LENGTH(#MYPARM) = 8
*
/* content of #MYPARM is moved back to #MYTEXT
/* used length of #MYTEXT = 8
*
END
```

例 **2**：

```
** Example 'DYNAMX04': Dynamic variables (as parameters)
************************************************************************
DEFINE DATA LOCAL
1 #MYTEXT (A) DYNAMIC
END-DEFINE
*
#MYTEXT := '123456'      /* extended to 6 bytes, *LENGTH(#MYTEXT) = 6
*
CALLNAT 'DYNAMX05' USING #MYTEXT
*
WRITE *LENGTH(#MYTEXT)          /* *LENGTH(#MYTEXT) = 8
                                /* at least 10 bytes are
                                /* allocated (extended in DYNAMX05)
```

```
*
END
```

サブプログラム DYNAMX05：

```
** Example 'DYNAMX05': Dynamic variables (as parameters)
************************************************************************
DEFINE DATA PARAMETER
1 #MYPARM (A) DYNAMIC
END-DEFINE
*
WRITE *LENGTH(#MYPARM)                  /* *LENGTH(#MYPARM) = 6
#MYPARM := '1234567'                    /* *LENGTH(#MYPARM) = 7
#MYPARM := '12345678'                   /* *LENGTH(#MYPARM) = 8
EXPAND DYNAMIC VARIABLE #MYPARM TO 10   /* 10 bytes are allocated
*
WRITE *LENGTH(#MYPARM)                  /* *LENGTH(#MYPARM) = 8
*
END
```

3GL プログラムに対する CALL

CALL ステートメントで INTERFACE4 オプションを使用すると、ダイナミック変数およびラージ変数を有効に使用できます。このオプションを使用すると、異なるパラメータ構造を使用した3GL プログラムとのインターフェイスになります。

このインターフェイスを使用するには 3GL プログラムを少々変更する必要がありますが、従来の FINFO 構造体と比較して、以下のような大きな利点があります。

- 渡すパラメータの数は無制限です（従来の制限：40）。
- パラメータのデータサイズは無制限です（従来の制限：パラメータ当たり 64 KB）。
- 配列情報を含め、完全なパラメータ情報を 3GL プログラムに渡すことができます。 パラメータデータへの安全なアクセスを可能にする、エクスポートされたファンクションが用意されています（従来は Natural 内部でメモリを上書きしないよう注意する必要がありました）。

FINFO 構造体の詳細については、CALL INTERFACE4 ステートメントの説明を参照してください。

ダイナミックパラメータを使用して 3GL プログラムを呼び出す前に、必要なバッファサイズを必ず割り当てるようにしてください。 これは、EXPAND ステートメントを使用すると明示的に実行できます。

バッファを初期化する必要がある場合、MOVE ALL UNTIL ステートメントを使用することにより、ダイナミック変数を初期値および必要なサイズに設定できます。 Natural には、3GL プログラムでダイナミックパラメータに関する情報を取得し、パラメータデータを返すときに長さを変更できるようにする、一連のファンクションが用意されています。

例：

```
MOVE ALL ' ' TO  #MYDYNTEXT1 UNTIL 10000
  /* a buffer of length 10000 is allocated
  /* #MYDYNTEXT1 is initialized with blanks
  /* and *LENGTH(#MYDYNTEXT1) = 10000
CALL INTERFACE4  'MYPROG'  USING  #MYDYNTEXT1
WRITE *LENGTH(#MYDYNTEXT1)
  /* *LENGTH(#MYDYNTEXT1) may have changed in the 3GL program
```

詳細については、『ステートメント』ドキュメントに記載されている、CALL ステートメントの説明を参照してください。

ラージ変数およびダイナミック変数によるワークファイルへのアクセス

以下では次のトピックについて説明します。

- PORTABLE および UNFORMATTED
- ASCII、ASCII-COMPRESSED、および SAG
- TRANSFER および ENTIRE CONNECTION に対する特別な条件

PORTABLE および UNFORMATTED

2 つのワークファイルタイプ PORTABLE と UNFORMATTED を使用して、ラージおよびダイナミック変数をワークファイルに書き込んだりワークファイルから読み取ったりできます。 これらのタイプには、ダイナミック変数に対するサイズ制限がありません。 ただし、ラージ変数は最大フィールド／レコード長の 32766 バイトを超えることはできません。

ワークファイルタイプ PORTABLE の場合は、フィールド情報がワークファイル内に保存されます。 レコード内のフィールドサイズが現在のサイズと異なる場合は、ダイナミック変数は READ 中にサイズ変更されます。

ワークファイルタイプ UNFORMATTED は、例えば、データベースから読み込んだビデオを、他のユーティリティで再生可能なファイルに直接格納する場合などに使用できます。 WRITE WORK ステートメントでは、フィールドはそのバイト長でファイルに書き込まれます。 すべてのデータタイプ（DYNAMIC であるかどうかに関わらず）は同じように扱われます。 構造情報は挿入されません。 Natural ではバッファリングメカニズムを使用するので、データが完全に書き込まれるのは CLOSE WORK の後のみであることが予測できます。 これは、Natural の稼働中にファイルが別のユーティリティで処理される場合に特に重要です。

READ WORK ステートメントでは、固定長のフィールドはその全体の長さで読み込まれます。 ファイルの終わりに到達すると、現在のフィールドの残りは空白で埋められます。 次のフィールドは変更されません。 データタイプが DYNAMIC の場合、ファイルが 1073741824 バイトを超えない

限り、ファイルの残りの部分がすべて読み込まれます。 ファイルの終わりに到達すると、残りのフィールド（変数）は変更されないまま維持されます（通常の Natural の動作）。

ASCII、ASCII-COMPRESSED、および SAG

ワークファイルタイプ ASCII、ASCII-COMPRESSED、および SAG（バイナリ）では、ダイナミック変数は処理できず、エラーが生成されます。 これらのワークファイルタイプに対するラージ変数では、フィールド／レコードの最大長である 32766 バイトを超えない限り、問題は発生しません。

TRANSFER および ENTIRE CONNECTION に対する特別な条件

`READ WORK FILE` ステートメントとともに使用すると、ワークファイルタイプ TRANSFER はダイナミック変数を処理できます。 この場合、ダイナミック変数に対するサイズ制限はありません。 ワークファイルタイプ ENTIRE CONNECTION では、ダイナミック変数は処理できません。 ただし、これらは両方とも、最長フィールド／レコードである 1073741824 バイトのラージ変数を処理できます。

`WRITE WORK FILE` ステートメントとともに使用すると、ワークファイルタイプ TRANSFER は、フィールド／レコードの最大長が 32766 バイトのダイナミック変数を処理できます。 ワークファイルタイプ ENTIRE CONNECTION では、ダイナミック変数は処理できません。 ただし、これらは両方とも、最長フィールド／レコードである 1073741824 バイトのラージ変数を処理できます。

可変長の列に対する DDM の生成および編集

データタイプに応じて、対応するデータベースフォーマット A または フォーマット B が生成されます。 データベースのデータタイプが `VARCHAR` の場合、Natural の列の長さは DBMS で定義されているデータタイプの最大長に設定されます。 データタイプが非常に大きい場合、フィールド長の位置にキーワード `DYNAMIC` が生成されます。

すべての可変長の列に対し、`LINDICATOR` フィールド `L@<column-name>` が生成されます。 データベースのデータタイプが `VARCHAR` の場合、フォーマット／長さが I2 の `LINDICATOR` フィールドが生成されます。 大きなデータタイプ（下表参照）の場合、フォーマット／長さは I4 になります。

データベースにアクセスするときには、`LINDICATOR` を操作することにより、定義済みのバッファ長（または `*LENGTH`）に関係なく、読み込むフィールドの長さを取得したり、書き込むフィールドの長さを設定したりできます。 通常は、取得処理の後、対応するインジケータの長さに `*LENGTH` は設定されます。

DDM の例：

```
  T  L  Name                      F   Leng        S   D   Remark
     :
     1  L@PICTURE1                I   4                                   /*
length indicator
     1  PICTURE1                  B   DYNAMIC             IMAGE
     1  N@PICTURE1                I   2                                   /* NULL
indicator
     1  L@TEXT1                   I   4                                   /*
length indicator
     1  TEXT1                     A   DYNAMIC             TEXT
     1  N@TEXT1                   I   2                                   /* NULL
indicator
     1  L@DESCRIPTION             I   2                                   /*
length indicator
     1  DESCRIPTION               A   1000                VARCHAR(1000)
     :
     :
  ~~~~~~~~~~~~~~~~~~~~~~Extended Attributes~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~/*
concerning PICTURE1
   Header               :    ---
   Edit Mask            :    ---
   Remarks              :   IMAGE
```

生成されるフォーマットは可変長のフォーマットです。Naturalプログラマは、定義をDYNAMICから固定長に変更したり（拡張フィールド編集）、データタイプVARCHARに対応するDDMのフィールド定義をマルチプルバリューフィールド（従来の生成タイプ）にしたりできます。

例：

```
  T  L  Name                    F   Leng          S   D   Remark
     :
     1  L@PICTURE1              I   4                                     /*
length indicator
     1  PICTURE1                B   1000000000            IMAGE
     1  N@PICTURE1              I   2                                     /* NULL
indicator
     1  L@TEXT1                 I   4                                     /*
length indicator
     1  TEXT1                   A   5000                  TEXT
     1  N@TEXT1                 I   2                                     /* NULL
indicator
     1  L@DESCRIPTION           I   2                                     /*
length indicator
  M  1  DESCRIPTION             A   100                   VARCHAR(1000)
       :
       :
  ~~~~~~~~~~~~~~~~~~~~Extended Attributes~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~/*
concerning PICTURE1
   Header             :    ---
```

```
   Edit Mask           :    ---
   Remarks             :   IMAGE
```

データベースのラージオブジェクトへのアクセス

データベースのラージオブジェクト（CLOBまたはBLOB）にアクセスするには、対応する英数字、Unicode、またはバイナリの各ラージフィールドのDDMを使用する必要があります。フィールドが固定長で定義され、データベースのラージオブジェクトがこのフィールドに収まらない場合、ラージオブジェクトは切り捨てられます。 データベースオブジェクトの明確な長さがわからない場合、ダイナミックフィールドを使用します。 そのオブジェクトを保持するために必要な再割り当てが行われます。 切り捨ては行われません。

プログラム例：

```
DEFINE DATA LOCAL

1 person VIEW OF xyz-person
  2 nachname
  2 vorname_1
  2 L@PICTURE1                    /* I4 length indicator for PICTURE1
  2 PICTURE1                      /* defined as dynamic in the DDM
  2 TEXT1                         /* defined as non-dynamic in the DDM

END-DEFINE

SELECT * INTO VIEW person FROM xyz-person                    /* PICTURE1 will be
read completely
                            WHERE nachname = 'SMITH'         /* TEXT1 will be
truncated to fixed length 5000

   WRITE  'length of PICTURE1: '  L@PICTURE1               /* the L-INDICATOR will
contain the length
                                                        /* of PICTURE1 (=
*LENGTH(PICTURE1)
   /* do something with PICTURE1 and TEXT1

   L@PICTURE1  :=  100000
   INSERT INTO xyz-person  (*) VALUES (VIEW person)        /* only the first 100000
Bytes of PICTURE1
```

```
                                                    /* are inserted
END-SELECT
```

ビューのフォーマット／長さの定義が省略されている場合、DDM の定義が使用されます。 レポーティングモードでは、対応する DDM フィールドが DYNAMIC で定義されている場合、任意の長さを指定できます。 ダイナミックフィールドは、固定長のバッファにマップされます。 この逆は実行できません。

DDM のフォーマット／長さの定義	ビューのフォーマット／長さの定義	
(An)	-	有効
	(An)	有効
	(Am)	レポーティングモードでのみ有効
	(A) DYNAMIC	無効
(A) DYNAMIC	-	有効
	(A) DYNAMIC	有効
	(An)	レポーティングモードでのみ有効
	(Am / i : j)	レポーティングモードでのみ有効

フォーマット B の変数も同様です。

SQL ステートメント内の LINDICATOR 節のパラメータ

LINDICATOR フィールドを I2 フィールドとして定義する場合、対応する列の値の受け渡しには SQL データタイプ VARCHAR を使用します。 LINDICATOR ホスト変数を I4 として指定する場合、ラージオブジェクトデータタイプ（CLOB／BLOB）を使用します。

フィールドを DYNAMIC として定義すると、列は内部ループでは実際の長さまで読み取られます。 LINDICATOR フィールドと *LENGTH はこの長さに設定されます。 固定長フィールドの場合、定義した長さまで列が読み込まれます。いずれの場合も、フィールドは LINDICATOR フィールドで定義した値まで書き込まれます。

ダイナミック変数の使用によるパフォーマンスへの影響

ダイナミック変数を少量ずつ複数回にわたって拡張する場合（バイト単位など）、必要なストレージの（おおよその）上限がわかっているときは、拡張を繰り返すのではなく EXPAND ステートメントを使用します。 これにより、必要なストレージを調整するための余分なオーバーヘッドを回避できます。

ダイナミック変数が不要になった場合、特に *LENGTH の値が大きい変数の場合は、REDUCE ステートメントまたは RESIZE ステートメントを使用します。 これにより、Natural でストレージを解放または再利用できます。 したがって、全体的なパフォーマンスが向上します。

ダイナミック変数に割り当てられているメモリの量は、REDUCE DYNAMIC VARIABLE ステートメントを使用すると減らすことができます。変数に特定の長さを（再）割り当てするには、EXPAND ステートメントを使用できます（変数を初期化する場合は、MOVE ALL UNTIL ステートメントを使用します）。

例：

```
** Example 'DYNAMX06': Dynamic variables (allocated memory)
************************************************************************
DEFINE DATA LOCAL
1 #MYDYNTEXT1  (A)  DYNAMIC
1 #LEN         (I4)
END-DEFINE
*
#MYDYNTEXT1 := 'a'     /* used length is 1, value is 'a'
                       /* allocated size is still 1
WRITE *LENGTH(#MYDYNTEXT1)
*
EXPAND DYNAMIC VARIABLE  #MYDYNTEXT1 TO 100
                       /* used length is still 1, value is 'a'
                       /* allocated size is 100
*
CALLNAT 'DYNAMX05' USING #MYDYNTEXT1
WRITE *LENGTH(#MYDYNTEXT1)
                       /* used length and allocated size
                       /* may have changed in the subprogram
*
#LEN := *LENGTH(#MYDYNTEXT1)
REDUCE DYNAMIC VARIABLE  #MYDYNTEXT1  TO  #LEN
                       /* if allocated size is greater than used length,
                       /* the unused memory is released
*
REDUCE DYNAMIC VARIABLE  #MYDYNTEXT1  TO  0
WRITE *LENGTH(#MYDYNTEXT1)
                       /* free allocated memory for dynamic variable
END
```

ルール：

- ダイナミックオペランドは、適切な場所で使用します。
- メモリ使用量の上限がわかっている場合は、EXPAND ステートメントを使用します。
- ダイナミックオペランドが不要になった場合は、REDUCE ステートメントを使用します。

ダイナミック変数の出力

ダイナミック変数は、以下のような出力ステートメント内で使用できます。

ステートメント	注意点
DISPLAY	これらのステートメントでは、セッションパラメータの AL（出力に対する英数字長）またはEM（編集マスク）を使用して、ダイナミック変数の入出力フォーマットを設定する必要があります。
WRITE	
INPUT	
REINPUT	--
PRINT	PRINT ステートメントの出力はフォーマットが指定されていないため、AL および EM パラメータを使用して、PRINT ステートメント内のダイナミック変数の出力フォーマットを設定する必要はありません。 したがって、これらのパラメータは無視されます。

ダイナミック X-array

ダイナミックX-arrayの割り当てでは、最初にオカレンス数を指定し、後でそのオカレンス数を拡張できます。

例：

```
DEFINE DATA LOCAL
 1 #X-ARRAY(A/1:*)  DYNAMIC
END-DEFINE
*
EXPAND  ARRAY   #X-ARRAY TO (1:10)  /* Current boundaries (1:10)
#X-ARRAY(*) := 'ABC'
EXPAND  ARRAY   #X-ARRAY TO (1:20)  /* Current boundaries (1:20)
#X-ARRAY(11:20) := 'DEF'
```

21 ユーザー定義定数

定数は、Naturalプログラム全体で使用できます。このドキュメントでは、サポートされている定数のタイプとその使用方法について説明します。

このchapterでは、次のトピックについて説明します。

数値定数

以下では次のトピックについて説明します。

- 数値定数
- 数値定数の検証

数値定数

数値定数には、1～29桁の数字と小数点表記のための特殊文字（ピリオドまたはコンマ）を指定できます。

例：

```
1234   +1234   -1234

12.34  +12.34  -12.34
```

```
MOVE 3 TO #XYZ
COMPUTE #PRICE = 23.34
COMPUTE #XYZ = -103
COMPUTE #A = #B * 6074
```

Note: 内部的には、小数点なしの数値定数は整数フォーマット（フォーマットI）で表されます。一方、小数点付きの数値定数、およびフォーマットIに収まらないほど大きな値を持つ小数点なしの数値定数は、パック型フォーマット（フォーマットP）で表されます。

例：

数値定数		フォーマット	長さ
最小値	最大値		
	<= -2147483649	P	>=10
-2147483648	-32769	I	4
-32768	32767	I	2
32768	2147483647	I	4
>= 2147483648		P	>=10

数値定数の検証

INIT オプションを使用した COMPUTE、MOVE、DEFINE DATA の各ステートメント内で数値定数を使用すると、コンパイル時に Natural によって、対応するフィールドに対して定数値が適しているかどうかが確認されます。 これにより、コンパイル時にエラー条件が検出されるため、ランタイムエラーを回避できます。

英数字定数

以下では次のトピックについて説明します。

- 英数字定数
- 英数字定数内のアポストロフィ
- 英数字定数の連結

英数字定数

英数字定数には、1～1073741824 バイト（1 GB）の英数字を指定できます。

英数字定数は、次のいずれかで囲む必要があります。：アポストロフィ（'）

```
'text'
```

または引用符（"）

```
"text"
```

例：

```
MOVE 'ABC' TO #FIELDX
MOVE '% INCREASE' TO #TITLE
DISPLAY "LAST-NAME" NAME
```

Note: ユーザー定義変数に割り当てるために使用する 1 つの英数字定数は、複数のステートメント行に分割できません。

英数字定数内のアポストロフィ

アポストロフィで囲まれた英数字定数内で1つのアポストロフィを表すには、2つのアポストロフィまたは 1 つの引用符を使用する必要があります。

引用符で囲まれた英数字定数内で1つのアポストロフィを表すには、1つのアポストロフィを指定します。

例：

以下を出力するとします。

```
HE SAID, 'HELLO'
```

以下のいずれの表記も使用できます。

```
WRITE 'HE SAID, ''HELLO'''
WRITE 'HE SAID, "HELLO"'
WRITE "HE SAID, ""HELLO"""
WRITE "HE SAID, 'HELLO'"
```

Note: 上述したように引用符がアポストロフィに変換されない場合、これはプロファイルパラメータ TQ（Translate Quotation Marks）の設定が原因です。詳細については、Natural の管理者に問い合わせてください。

英数字定数の連結

ハイフンを使用すると、複数の英数字定数を 1 つの値として連結できます。

例：

```
MOVE 'XXXXXX' - 'YYYYYY' TO #FIELD
MOVE "ABC" - 'DEF' TO #FIELD
```

上記の方法で、英数字定数を **16** 進定数と連結することもできます。

Unicode 定数

以下では次のトピックについて説明します。

- Unicode テキスト定数
- Unicode テキスト定数内のアポストロフィ
- Unicode 16 進定数
- Unicode 定数の連結

Unicode テキスト定数

Unicodeテキスト定数は、先頭に文字"U"を指定し、その後の文字列を次のいずれかで囲む必要があります。：アポストロフィ（'）

```
U'text'
```

または引用符（"）

```
U"text"
```

例：

```
U'HELLO'
```

コンパイラは、このテキスト定数を Unicode フォーマット（UTF-16）で生成プログラムに格納します。

Unicode テキスト定数内のアポストロフィ

アポストロフィで囲まれたUnicodeテキスト定数内で1つのアポストロフィを表すには、2つのアポストロフィまたは1つの引用符を使用する必要があります。

引用符で囲まれたUnicodeテキスト定数内で1つのアポストロフィを表すには、1つのアポストロフィを指定します。

例：

以下を出力するとします。

```
HE SAID, 'HELLO'
```

以下のいずれの表記も使用できます。

```
WRITE U'HE SAID, ''HELLO'''
WRITE U'HE SAID, "HELLO"'
WRITE U"HE SAID, ""HELLO"""
WRITE U"HE SAID, 'HELLO'"
```

Note: 上述したように引用符がアポストロフィに変換されない場合、これはプロファイルパラメータ `TQ`（Translate Quotation Marks）の設定が原因です。詳細については、Naturalの管理者に問い合わせてください。

Unicode 16 進定数

以下の構文は、Unicode 文字または Unicode 文字列を 16 進表記で指定するために使用します。

```
UH'hhhh...'
```

h は、16 進数の桁（0～9、A～F）を表します。 UTF-16 Unicode 文字はダブルバイトであるため、指定する 16 進の文字数は 4 の倍数である必要があります。

例：

以下の例は、文字列 "45" を定義しています。

```
UH'00340035'
```

Unicode 定数の連結

Unicode テキスト定数（U）および Unicode 16 進定数（UH）は、連結できます。

有効な例：

```
MOVE U'XXXXXX' - UH'00340035' TO #FIELD
```

Unicode テキスト定数または Unicode 16 進定数は、コードページ英数字定数または H 定数とは連結できません。

無効な例：

```

MOVE U'ABC' - 'DEF' TO #FIELD
MOVE UH'00340035' - H'414243' TO #FIELD
```

その他の有効な例：

```

DEFINE DATA LOCAL
1 #U10 (U10)              /* Unicode variable with 10 (UTF-16) characters, total
byte length = 20
1 #UD  (U)  DYNAMIC        /* Unicode variable with dynamic length
END-DEFINE
*
#U10 := U'ABC'            /* Constant is created as X'004100420043' in the object,
the UTF-16 representation for string 'ABC'.

#U10 := UH'004100420043'   /* Constant supplied in hexadecimal format only,
corresponds to U'ABC'

#U10 := U'A'-UH'0042'-U'C' /* Constant supplied in mixed formats, corresponds to
```

```
U'ABC'.
END
```

日付／時刻の定数

以下では次のトピックについて説明します。

- 日付定数
- 時刻定数
- 拡張時刻定数

日付定数

日付定数は、フォーマット D の変数とともに使用します。

日付定数は、以下の形式で定義します。

D'*yyyy-mm-dd***'**	国際式
D'*dd.mm.yyyy***'**	ドイツ式
D'*dd/mm/yyyy***'**	ヨーロッパ式
D'*mm/dd/yyyy***'**	米国式

上記の *dd* は日、*mm* は月、*yyyy* は年を表します。

例：

```
DEFINE DATA LOCAL
  1 #DATE (D)
END-DEFINE
...
MOVE D'2004-03-08' TO #DATE
...
```

デフォルトの日付形式は、プロファイルパラメータ DTFORM（日付形式）を使用して、Natural 管理者が制御します。

時刻定数

時刻定数は、フォーマット T の変数とともに使用します。

時刻定数は、以下の形式で定義します。

```
T'hh:ii:ss'
```

上記の *hh* は時間、*ii* は分、*ss* は秒を表します。

例：

```
DEFINE DATA LOCAL
  1 #TIME (T)
END-DEFINE
...
MOVE T'11:33:00' TO #TIME
...
```

拡張時刻定数

日付情報は時刻情報のサブセットであるため、時刻変数（フォーマットT）には、日付および時刻の情報を格納できます。ただし、「標準の」時刻定数（接頭辞"T"）では、時刻変数の時刻情報のみを処理できます。

```
T'hh:ii:ss'
```

拡張時刻定数（接頭辞 "E"）を使用すると、日付情報を含む、時刻変数のすべての内容を処理できます。

```
E'yyyy-mm-dd hh:ii:ss'
```

この点を除き、拡張時刻定数と時刻変数の使用方法は、標準の時刻定数の場合と同じです。

Note: 拡張時刻定数に指定する必要のある日付情報の形式は、プロファイルパラメータ DTFORM の設定によって決まります。 上記の拡張時刻定数は、DTFORM=I（国際式）を想定しています。

16 進の定数

以下では次のトピックについて説明します。

16 進の定数

16 進定数では、標準のキーボード文字で入力できない値を指定できます。

16 進定数には、1～1073741824 バイト（1 GB）の英数字を指定できます。

接頭文字 "H" で、16 進定数を示します。 定数自身は、16 進数を表す文字（0～9、A～F）で構成される文字列をアポストロフィで囲む必要があります。 1 バイトのデータを表すには、2 つの 16 進文字が必要です。

16 進文字表現は、コンピュータが ASCII 文字セットを使用しているか、EBCDIC 文字セットを使用しているかによって異なります。 したがって、16 進定数を別のコンピュータに転送する場合、文字変換が必要になることがあります。

ASCII の例：

```
H'313233'    (equivalent to the alphanumeric constant '123')
H'414243'    (equivalent to the alphanumeric constant 'ABC')
```

EBCDIC の例：

```
H'F1F2F3'    (equivalent to the alphanumeric constant '123')
H'C1C2C3'    (equivalent to the alphanumeric constant 'ABC')
```

16進定数を別のフィールドに転送する場合は、英数字値（フォーマット A）として扱われます。

フォーマット A、U、B 以外で定義されているフィールドに、英数字値（フォーマット A）をデータ転送することはできません。 したがって、対応する変数がフォーマット A、U、B ではない場合に `DEFINE DATA` ステートメントで 16 進定数を初期値として使用すると、構文エラー NAT0094 で拒否されます。

例：

```
DEFINE DATA LOCAL
1 #I(I2) INIT <H'000F'>      /* causes a NAT0094 syntax error
END-DEFINE
```

16 進定数の連結

定数の間にハイフンを使用すると、16 進定数を連結できます。

ASCII の例：

```
H'414243' - H'444546' (equivalent to 'ABCDEF')
```

EBCDIC の例：

```
H'C1C2C3' - H'C4C5C6' (equivalent to 'ABCDEF')
```

上記の方法で、16 進定数を英数字定数と連結することもできます。

論理定数

論理定数 "TRUE" および "FALSE" は、フォーマット L で定義されたフィールドに論理値を割り当てるために使用できます。

例：

```
DEFINE DATA LOCAL
  1 #FLAG (L)
END-DEFINE
...
MOVE TRUE TO #FLAG
...
IF #FLAG ...
  statement ...
  MOVE FALSE TO #FLAG
```

```
END-IF
 ...
```

浮動小数点定数

浮動小数点定数は、フォーマット F で定義された変数で使用できます。

例：

```
DEFINE DATA LOCAL
  1 #FLT1 (F4)
END-DEFINE
...
COMPUTE #FLT1 = -5.34E+2
...
```

属性定数

属性定数は、フォーマットCで定義された変数（制御変数）で使用できます。属性定数は、カッコで囲む必要があります。

次の属性を使用できます。

属性	説明
AD=D	デフォルト
AD=B	点滅
AD=I	高輝度
AD=N	非表示
AD=V	反転
AD=U	下線付き
AD=C	イタリック
AD=Y	ダイナミック属性
AD=P	保護
CD=BL	青
CD=GR	緑
CD=NE	デフォルト色
CD=PI	ピンク
CD=RE	赤

属性	説明
CD=TU	空色
CD=YE	黄色

セッションパラメータ AD および CD の説明も参照してください。

例：

```
DEFINE DATA LOCAL
  1 #ATTR (C)
  1 #FIELD (A10)
END-DEFINE
...
MOVE (AD=I CD=BL) TO #ATTR
...
INPUT #FIELD (CV=#ATTR)
...
```

ハンドル定数

ハンドル定数NULL-HANDLEは、GUIハンドルおよびオブジェクトハンドルで使用できます。

GUI ハンドルの詳細については、「*ダイアログエレメントを定義する方法*」を参照してください。

オブジェクトハンドルの詳細については、「***NaturalX***」を参照してください。

名前付き定数の定義

同じ定数値を何度も使用する必要がある場合、以下のように名前付き定数を定義してメンテナンスの労力を減らすことができます。

- DEFINE DATA ステートメント内でフィールドを定義します。
- フィールドに定数値を割り当てます。
- プログラムで定数値ではなくフィールド名を使用します。

これにより、値を変更する必要があるときには、プログラム内の関連するすべての場所を変更するのではなく、DEFINE DATA ステートメントを1回変更するだけですべての値を変更できます。

DEFINE DATA ステートメントのフィールド定義の後に、キーワード CONSTANT を付けた山カッコ（< と >）内に定数値を指定します。

- 値が英数字の場合、アポストロフィで囲む必要があります。
- 値が Unicode テキストフォーマットの場合、先頭に文字 "U" を指定し、その後の文字列をアポストロフィで囲む必要があります。
- 値が Unicode 16 進フォーマットの場合、先頭に文字 "UH" を指定し、その後の文字列をアポストロフィで囲む必要があります。

例：

```
DEFINE DATA LOCAL
  1 #FIELDA (N3) CONSTANT <100>
  1 #FIELDB (A5) CONSTANT <'ABCDE'>
  1 #FIELDC (U5) CONSTANT <U'ABCDE'>
  1 #FIELDD (U5) CONSTANT <UH'00410042004300440045'>
END-DEFINE
...
```

プログラム実行中は、名前付き定数の値を変更できません。

22 初期値（および RESET ステートメント）

このchapterでは、ユーザー定義変数のデフォルトの初期値、ユーザー定義変数への初期値の割り当て方法、およびフィールド値をデフォルトの初期値または DEFINE DATA ステートメントで定義した初期値に戻すための RESET ステートメントの使用方法について説明します。

このchapterでは、次のトピックについて説明します。

ユーザー定義変数／配列のデフォルトの初期値

フィールドに初期値を指定しない場合、そのフォーマットに応じたデフォルトの初期値でフィールドは初期化されます。

フォーマット	デフォルトの初期値
B、F、I、N、P	0
A、U	空白
L	F(ALSE)
D	D' '
T	T'00:00:00'
C	(AD=D)
GUI ハンドル	NULL-HANDLE
オブジェクトハンドル	NULL-HANDLE

ユーザー定義変数／配列への初期値の割り当て

DEFINE DATA ステートメントでは、ユーザー定義変数に初期値を割り当てることができます。初期値が英数字の場合、アポストロフィで囲む必要があります。

- 変更可能な初期値の割り当て
- 変更不可能な初期値の割り当て
- 初期値としての Natural システム変数の割り当て

- 英数字変数の初期値としての文字列の割り当て

変更可能な初期値の割り当て

変更可能な初期値を変数／配列に割り当てる場合、DEFINE DATA ステートメントの変数定義の後に、キーワード INIT とともに山カッコ（< と >）で囲んだ初期値を指定します。 割り当てられた値は、変数／配列が参照されるたびに使用されます。 割り当てられた値は、プログラムの実行中に変更できます。

例：

```
DEFINE DATA LOCAL
1 #FIELDA (N3) INIT <100>
1 #FIELDB (A20) INIT <'ABC'>
END-DEFINE
...
```

変更不可能な初期値の割り当て

変数／配列を名前付き定数として扱う場合、DEFINE DATA ステートメントの変数定義の後に、キーワード CONSTANT とともに山カッコ（< と >）で囲んだ初期値を指定します。 割り当てられた定数値は、変数／配列が参照されるたびに使用されます。 割り当てられた値は、プログラムの実行中には変更できません。

例：

```
DEFINE DATA LOCAL
1 #FIELDA (N3) CONST <100>
1 #FIELDB (A20) CONST <'ABC'>
END-DEFINE
...
```

初期値としての Natural システム変数の割り当て

フィールドの初期値として、**Natural** システム変数の値を指定することもできます。

例：

以下の例では、システム変数 *DATX が初期値を指定するために使用されています。

```
DEFINE DATA LOCAL
1 #MYDATE (D) INIT <*DATX>
END-DEFINE
...
```

英数字変数の初期値としての文字列の割り当て

初期値として、特定の1文字または文字列で、変数全体または変数の一部を埋めることができます（英数字変数にのみ有効）。

- フィールド全体を埋める場合
  FULL LENGTH *<character(s)>* オプションを使用して、フィールド全体を特定の文字（列）で埋めます。

  以下の例では、フィールド全体をアスタリスク（*）で埋めています。

  ```
  DEFINE DATA LOCAL
  1 #FIELD (A25) INIT FULL LENGTH <'*'>
  END-DEFINE
  ...
  ```

- フィールドの最初の*n* 文字を埋める場合
  LENGTH *n <character(s)>* オプションを使用して、フィールドの先頭 *n* 文字を特定の文字（列）で埋めます。

  以下の例では、フィールドの先頭 4 文字を感嘆符（!）で埋めています。

  ```
  DEFINE DATA LOCAL
  1 #FIELD (A25) INIT LENGTH 4 <'!'>
  END-DEFINE
  ...
  ```

ユーザー定義変数の初期値へのリセット

RESET ステートメントは、フィールドの値をリセットするために使用します。 以下の 2 つのオプションを使用できます。

- デフォルトの初期値へのリセット
- DEFINE DATA で定義した初期値へのリセット

Notes:

1. DEFINE DATA ステートメントで CONSTANT 節を指定して宣言したフィールドは、内容を変更できないので、RESET ステートメントで参照できません。
2. レポーティングモードでは、プログラムに DEFINE DATA LOCAL ステートメントが含まれていなければ、RESET ステートメントを使用して変数を定義することもできます。

デフォルトの初期値へのリセット

RESET（INITIAL なし）は、指定された各フィールドの内容をフォーマットに依存したデフォルトの初期値に設定します。

例：

```
DEFINE DATA LOCAL
1 #FIELDA (N3)  INIT <100>
1 #FIELDB (A20) INIT <'ABC'>
1 #FIELDC (I4)  INIT <5>
END-DEFINE
...
...
RESET #FIELDA                    /* resets field value to default initial value
...
```

DEFINE DATA で定義した初期値へのリセット

RESET INITIAL は、指定された各フィールドを DEFINE DATA ステートメントのフィールド定義に従った初期値に設定します。

DEFINE DATA ステートメントで INIT 節を指定せずに宣言したフィールドに対して、RESET INITIAL は RESET（INITIAL なし）と同じ効果を持ちます。

例：

```
DEFINE DATA LOCAL
1 #FIELDA (N3)  INIT <100>
1 #FIELDB (A20) INIT <'ABC'>
1 #FIELDC (I4)  INIT <5>
END-DEFINE
...
RESET INITIAL #FIELDA #FIELDB #FIELDC /* resets field values to initial values as
defined in DEFINE DATA
...
```

23 フィールドの再定義

再定義は、フィールドのフォーマットを変更したり、単一フィールドを複数のセグメントに分割したりするために使用します。

このchapterでは、次のトピックについて説明します。

DEFINE DATA ステートメントの REDEFINE オプションの使用

DEFINE DATA ステートメントの REDEFINE オプションは、単一のフィールド（ユーザー定義変数またはデータベースフィールド）を1つ以上の新しいフィールドとして再定義するために使用できます。 グループも再定義できます。

Important: ダイナミック変数は再定義には使用できません。

REDEFINE オプションは、フォーマットに関係なく、フィールドのバイト位置を左から右に再定義します。 バイト位置は、元のフィールドと再定義されたフィールド（複数可）の間で一致している必要があります。

再定義は元のフィールド定義の直後に指定する必要があります。

例 **1**：

以下の例では、データベースフィールド BIRTH を、3つの新しいユーザー定義変数として再定義しています。

```
DEFINE DATA LOCAL
01 EMPLOY-VIEW VIEW OF STAFFDDM
  02 NAME
  02 BIRTH
  02 REDEFINE BIRTH
     03 #BIRTH-YEAR  (N4)
     03 #BIRTH-MONTH (N2)
     03 #BIRTH-DAY   (N2)
END-DEFINE
   ...
```

例 **2**：

以下の例では、グループ #VAR2（フォーマット N および P の 2 つのユーザー定義変数で構成）を、フォーマット A の 1 つの変数として再定義しています。

```
DEFINE DATA LOCAL
01 #VAR1 (A15)
01 #VAR2
  02 #VAR2A (N4.1)
  02 #VAR2B (P6.2)
01 REDEFINE #VAR2
  02 #VAR2RD (A10)
END-DEFINE
   ...
```

FILLER *nX* 表記を使用して、再定義するフィールドに *n* バイトの充填バイト（使用しないセグメント）を定義できます。 末尾の充填バイトの定義は任意です。

例 **3**：

以下の例では、ユーザー定義変数 #FIELD を 3 つの新しいユーザー定義変数（それぞれフォーマット／長さが A2）として再定義しています。 FILLER 表記によって、元のフィールドの 3〜4、および 7〜10 バイト目を使用しないことが示されています。

```
DEFINE DATA LOCAL
1 #FIELD (A12)
1 REDEFINE #FIELD
  2 #RFIELD1 (A2)
  2 FILLER 2X
  2 #RFIELD2 (A2)
  2 FILLER 4X
  2 #RFIELD3 (A2)
END-DEFINE
...
```

再定義の使用方法を示すプログラム例

以下のプログラムは、再定義の使用方法を示しています。

```
** Example 'DDATAX01': DEFINE DATA
************************************************************************
DEFINE DATA LOCAL
01 VIEWEMP VIEW OF EMPLOYEES
  02 NAME
  02 FIRST-NAME
  02 SALARY   (1:1)
*
01 #PAY       (N9)
01 REDEFINE #PAY
  02 FILLER   3X
  02 #USD     (N3)
  02 #OOO     (N3)
END-DEFINE
*
READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'
  MOVE SALARY (1) TO #PAY
  DISPLAY NAME FIRST-NAME #PAY #USD #OOO
END-READ
END
```

プログラム DDATAX01 の出力：

#PAY およびその再定義でできたフィールドがどのように表示されるかに注意してください。

```
Page      1                                              04-11-11  14:15:54

        NAME              FIRST-NAME          #PAY    #USD #OOO
-------------------- -------------------- ---------- ---- ----

JONES                VIRGINIA                46000   46    0
JONES                MARSHA                  50000   50    0
JONES                ROBERT                  31000   31    0
```

24 配列

Naturalは配列の処理をサポートしています。配列は、複数次元のテーブル、つまり、単一の名前で識別される2つ以上の論理的に関連する要素です。配列は、複数次元の単一データ要素、または連続する構造体や個別の要素を含む階層的なデータ構造で構成できます。

このchapterでは、次のトピックについて説明します。

配列の定義

Natural では、1次元、2次元、または3次元の配列を使用できます。配列は、独立した変数、より大きなデータ構造の一部、またはデータベースビューの一部です。

Important: ダイナミック変数は配列定義には使用できません。

手順 24.1. 1 次元配列を定義するには

- フォーマットと長さの後に、スラッシュと「インデックス表記」、つまり配列のオカレンス数を指定します。

  例えば、以下の1次元配列には、3つのオカレンス（各オカレンスのフォーマット／長さはA10）があります。

```
DEFINE DATA LOCAL
1 #ARRAY (A10/1:3)
END-DEFINE
...
```

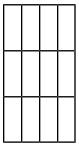
手順 24.2. 2 次元配列を定義するには

- 両方の次元に対するインデックス表記を指定します。

```
DEFINE DATA LOCAL
1 #ARRAY (A10/1:3,1:4)
```

```
END-DEFINE
...
```

2 次元配列はテーブルとして表すことができます。 上記の例で定義されている配列は、3 つの 「行」 および 4 つの 「列」 から成るテーブルです。



配列の初期値

配列の 1 つ以上のオカレンスに初期値を割り当てるには、以下の例のように、「通常の」変数に対する初期値定義と同様に INIT 指定を使用します。

1 次元配列への初期値の割り当て

以下の例は、初期値が 1 次元配列にどのように割り当てられるかを説明しています。

- 初期値を 1 つのオカレンスに割り当てる場合、次を指定します。

```
1 #ARRAY (A1/1:3) INIT (2) <'A'>
```

"A" は、2 番目のオカレンスに割り当てられます。

- 同じ初期値をすべてのオカレンスに割り当てるには、以下のように指定します。

```
1 #ARRAY (A1/1:3) INIT ALL <'A'>
```

"A" は、すべてのオカレンスに割り当てられます。 または、次を指定できます。

```
1 #ARRAY (A1/1:3) INIT (*) <'A'>
```

- 同じ初期値を複数オカレンスの範囲に割り当てるには、以下のように指定します。

```
1 #ARRAY (A1/1:3) INIT (2:3) <'A'>
```

"A" は、2〜3 番目のオカレンスに割り当てられます。

- 異なる初期値を全オカレンスに割り当てるには、以下のように指定します。

```
1 #ARRAY (A1/1:3) INIT <'A','B','C'>
```

"A" は最初のオカレンス、"B" は 2 番目のオカレンス、"C" は 3 番目のオカレンスにそれぞれ割り当てられます。

- 異なる初期値をいくつかのオカレンス（全オカレンスではなく）に割り当てるには、以下のように指定します。

```
1 #ARRAY (A1/1:3) INIT (1) <'A'> (3) <'C'>
```

"A" は最初のオカレンス、"C" は 3 番目のオカレンスに割り当てられますが、2 番目のオカレンスには値は割り当てられません。

または、次を指定できます。

```
1 #ARRAY (A1/1:3) INIT <'A',,'C'>
```

- オカレンス数より少ない数の初期値が指定されると、残ったオカレンスは空のままになります。

```
1 #ARRAY (A1/1:3) INIT <'A','B'>
```

"A" は最初のオカレンス、"B" は 2 番目のオカレンスに割り当てられますが、3 番目のオカレンスには値は割り当てられません。

2 次元配列への初期値の割り当て

このセクションでは、2次元配列に初期値を割り当てる方法について説明します。以下のトピックについて説明します。

- 異なる値の割り当て

前提条件

このセクションの例では、3 オカレンスの第 1 次元（「行」）と 4 オカレンスの第 2 次元（「列」）を持つ 2 次元配列を使用するものとします。

```
1 #ARRAY (A1/1:3,1:4)
```

縦方向：第 **1** 次元（**1:3**）、横方向：第 **2** 次元（**1:4**）

(1,1)	(1,2)	(1,3)	(1,4)
(2,1)	(2,2)	(2,3)	(2,4)
(3,1)	(3,2)	(3,3)	(3,4)

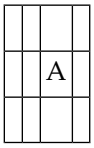
以下の最初の例では、*同じ*初期値を2次元配列のオカレンスにどのように割り当てるのかを説明します。2 番目の例では、*異なる*初期値をどのように割り当てるのかを説明します。

例では、"*" と "V" の表記の使用に特に注意してください。 これらの表記は両方とも、該当する次元のすべてのオカレンスを参照します。"*" は、その次元のすべてのオカレンスを*同じ*値で初期化することを示します。一方、"V" は、その次元のすべてのオカレンスを*異なる*値で初期化することを示します。

同じ値の割り当て

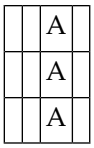
- 初期値を 1 つのオカレンスに割り当てる場合、次を指定します。

```
1 #ARRAY (A1/1:3,1:4) INIT (2,3) <'A'>
```



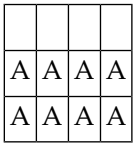
- 同じ初期値を第 2 次元の 1 オカレンス（第 1 次元の全オカレンス）に割り当てるには、以下のように指定します。

```
1 #ARRAY (A1/1:3,1:4) INIT (*,3) <'A'>
```



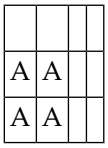
■ 同じ初期値を第1次元のオカレンス範囲（第2次元の全オカレンス）に割り当てるには、以下のように指定します。

```
1 #ARRAY (A1/1:3,1:4) INIT (2:3,*) <'A'>
```



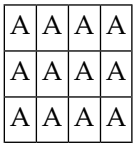
■ 同じ初期値を各次元のオカレンス範囲に割り当てるには、以下のように指定します。

```
1 #ARRAY (A1/1:3,1:4) INIT (2:3,1:2) <'A'>
```



■ 同じ初期値を全オカレンス（両方の次元）に割り当てるには、以下のように指定します。

```
1 #ARRAY (A1/1:3,1:4) INIT ALL <'A'>
```



または、次を指定できます。

```
1 #ARRAY (A1/1:3,1:4) INIT (*,*) <'A'>
```

異なる値の割り当て

■ `1 #ARRAY (A1/1:3,1:4) INIT (V,2) <'A','B','C'>`

	A		
	B		
	C		

■ `1 #ARRAY (A1/1:3,1:4) INIT (V,2:3) <'A','B','C'>`

	A	A	
	B	B	
	C	C	

■ `1 #ARRAY (A1/1:3,1:4) INIT (V,*) <'A','B','C'>`

A	A	A	A
B	B	B	B
C	C	C	C

■ `1 #ARRAY (A1/1:3,1:4) INIT (V,*) <'A',,'C'>`

A	A	A	A
C	C	C	C

■ `1 #ARRAY (A1/1:3,1:4) INIT (V,*) <'A','B'>`

A	A	A	A
B	B	B	B

■ `1 #ARRAY (A1/1:3,1:4) INIT (V,1) <'A','B','C'> (V,3) <'D','E','F'>`

A		D	
B		E	
C		F	

```
1 #ARRAY (A1/1:3,1:4) INIT (3,V) <'A','B','C','D'>
```

A	B	C	D

```
1 #ARRAY (A1/1:3,1:4) INIT (*,V) <'A','B','C','D'>
```

A	B	C	D
A	B	C	D
A	B	C	D

```
1 #ARRAY (A1/1:3,1:4) INIT (2,1) <'A'> (*,2) <'B'> (3,3) <'C'> (3,4) <'D'>
```

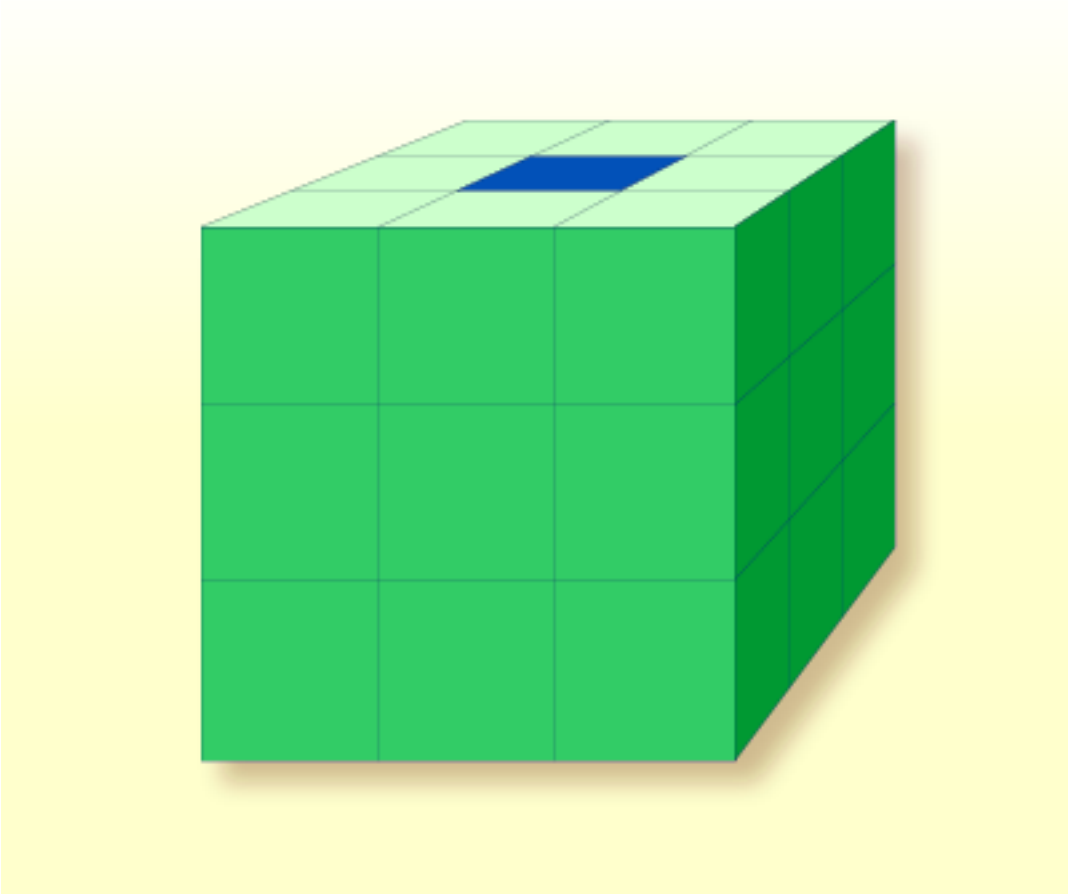
	B		
A	B		
	B	C	D

```
1 #ARRAY (A1/1:3,1:4) INIT (2,1) <'A'> (V,2) <'B',C',D'> (3,3) <'E'> (3,4) <'F'>
```

	B		
A	C		
	D	E	F

3 次元配列

3 次元配列は以下のように表すことができます。



上記の配列は、以下のように定義します（同時に、行 1、列 2、および面 2 の強調表示されたフィールドに初期値を割り当てます）。

```
DEFINE DATA LOCAL
1 #ARRAY2
  2 #ROW (1:4)
    3 #COLUMN (1:3)
      4 #PLANE (1:3)
        5 #FIELD2 (P3) INIT (1,2,2) <100>
```

```
END-DEFINE
...
```

データエリアエディタでローカルデータエリアとして定義すると、同じ配列は以下のように表示されます。

```
I T L Name                            F Leng Index/Init/EM/Name/Comment
- - - --------------------------------- - ---- ---------------------------------
    1 #ARRAY2
    2 #ROW                                   (1:4)
    3 #COLUMN                                (1:3)
    4 #PLANE                                 (1:3)
I   5 #FIELD2                           P    3
```

大きいデータ構造の一部としての配列

複数次元の配列を使用すると、COBOLまたはPL1の構造体に類似したデータ構造体を定義できます。

例：

```
DEFINE DATA LOCAL
1 #AREA
  2 #FIELD1 (A10)
  2 #GROUP1 (1:10)
    3 #FIELD2 (P2)
    3 #FIELD3 (N1/1:4)
END-DEFINE
...
```

この例では、データエリア #AREA の全体のサイズは以下のとおりです。

10 + (10 * (2 + (1 * 4))) バイト = 70 バイト

#FIELD1 は 10 バイトの長さの英数字です。 #GROUP1 は #AREA 内のサブエリアの名前で、2 つのフィールドから成り、10 オカレンスを持っています。 #FIELD2 は、パック型数値で長さは 2 です。 #FIELD3 は、4 オカレンスを持つ、#GROUP1 の 2 番目のフィールドで長さ 1 の数値です。

#FIELD3 の特定のオカレンスを参照するには、2 つのインデックスを使用する必要があります。最初のインデックスで #GROUP1 のオカレンスを指定し、2 番目のインデックスで #FIELD3 の特定のオカレンスを指定する必要があります。 例えば、同じプログラムにおいて、後から ADD ステートメントで #FIELD3 を参照するには、以下のように指定します。

```
ADD 2 TO #FIELD3 (3,2)
```

データベース配列

Adabas は、マルチプルバリューフィールドおよびピリオディックグループの形で、データベース内の配列構造をサポートします。 これらについては、「*データベース配列*」を参照してください。

以下の例は、マルチプルバリューフィールドを含むビューの DEFINE DATA での定義を示しています。

```
DEFINE DATA LOCAL
1 EMPLOYEES-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 ADDRESS-LINE (1:10) /* <--- MULTIPLE-VALUE FIELD
END-DEFINE
...
```

同じビューが、ローカルデータエリアでは以下のように表示されます。

```
I  T  L Name                            F Leng Index/Init/EM/Name/Comment
-  -  - --------------------------------- - ---- ---------------------------------
   V  1 EMPLOYEES-VIEW                             EMPLOYEES
     2 NAME                               A   20
   M 2 ADDRESS-LINE                       A   20 (1:10) /* MU-FIELD
```

インデックス表記での演算式の使用

配列のオカレンス範囲を表すために簡単な演算式をインデックスとして使用できます。

例：

MA (I:I+5)	値 I から I + 5 までの範囲のフィールド MA の値が参照されます。
MA (I+2:J-3)	値 I + 2 から J - 3 までの範囲のフィールド MA の値が参照されます。

インデックスの演算に使用できる演算子は、"+" および "-" のみです。

配列演算のサポート

配列演算のサポートには、配列レベル、行／列レベル、および個々の要素レベルでの演算が含まれています。

配列演算では、1 つまたは 2 つのオペランドと受け取りフィールドの 3 番目の変数（任意）を使用した簡単な演算式のみを使用できます。

インデックスの範囲を定義する式には、演算子 "+" および "-" のみを使用できます。

配列演算の例

以下の例では、次のフィールド定義を想定しています。

```
DEFINE DATA LOCAL
01 #A (N5/1:10,1:10)
01 #B (N5/1:10,1:10)
01 #C (N5)
END-DEFINE
...
```

1. `ADD #A(*,*) TO #B(*,*)`

   結果オペランドである配列 #B には、配列 #A と配列 #B の元の値を要素ごとに加算した結果が格納されます。

2. `ADD 4 TO #A(*,2)`

   配列 #A の 2 番目の列は、その元の値に 4 を加えた値で置き換えられます。

3. `ADD 2 TO #A(2,*)`

   配列 #A の 2 番目の行は、その元の値に 2 を加えた値で置き換えられます。

4. `ADD #A(2,*) TO #B(4,*)`

   配列 #A の 2 番目の行の値は、配列 #B の 4 番目の行に加算されます。

5. `ADD #A(2,*) TO #B(*,2)`

   これは不正な演算であるため、構文エラーが発生します。 行は行に、列は列にのみ加算できます。

6. `ADD #A(2,*) TO #C`

   配列 #A の 2 番目の行のすべての値は、スカラ値 #C に加算されます。

7. `ADD #A(2,5:7) TO #C`

   配列 #A の 2 番目の行の 5、6、7 番目の列の値がスカラ値 #C に加算されます。

25 X-array

通常の配列のフィールドを定義するときは、インデックスの範囲、つまり次元ごとのオカレンス数を正確に指定する必要があります。 ランタイムには配列のすべてのフィールドがデフォルトで存在するため、追加の割り当て操作を行うことなく、定義した各オカレンスにアクセスできます。 サイズのレイアウトは変更できません。したがって、フィールドのオカレンスを追加することも削除することもできません。

一方、必要なオカレンス数が開発時にわからないため、ランタイムに配列のフィールド数を柔軟に増減したい場合は、X-array（eXtensible array：拡張可能な配列）と呼ばれているものを使用します。

X-arrayでは、ランタイムにサイズ変更できるため、メモリをより効率よく管理できます。例えば、大量の配列オカレンスを短期間だけ使用してから、アプリケーションが配列を使用しなくなった時点でメモリを削減することができます。

このchapterでは、次のトピックについて説明します。

定義

X-array とは、コンパイル時にオカレンス数を定義しない配列のことです。 X-array は、DEFINE DATA ステートメント内で、少なくとも 1 つの次元のインデックス範囲に、アスタリスク（*）を使用して定義します。インデックス定義内のアスタリスク文字（*）は、プログラムの実行時に特定の値を割り当てることができる、変更可能なインデックス範囲を表します。 上限または下限のいずれかのみを可変として定義できます。両方を定義することはできません。

X-array は、（固定の）配列を定義できる場所にならどこにでも、つまり、任意のレベルで、またはインデックス付きのグループとして定義できます。ただし、データベースビューのMU-/PE-フィールドへのアクセスには使用できません。 複数次元の配列では、定数と変更可能な境界を組み合わせて使用できます。

例：

```
DEFINE DATA LOCAL
1 #X-ARR1 (A5/1:*)          /* lower bound is fixed at 1, upper bound is variable
1 #X-ARR2 (A5/*)            /* shortcut for (A5/1:*)
1 #X-ARR3 (A5/*:100)        /* lower bound is variable, upper bound is fixed at 100
1 #X-ARR4 (A5/1:10,1:*)     /* 1st dimension has a fixed index range with (1:10)
```

```
END-DEFINE                    /* 2nd dimension has fixed lower bound 1 and variable
upper bound
```

X-array のストレージ管理

X-arrayのオカレンス数は、アクセスする前に明示的に割り当てる必要があります。次元のオカレンス数を増減するには、EXPAND、RESIZE、REDUCE の各ステートメントを使用します。

ただし、X-array の次元数（1、2、または 3 次元）は変更できません。

例：

```
DEFINE DATA LOCAL
1 #X-ARR(I4/10:*)
END-DEFINE
EXPAND ARRAY #X-ARR TO (10:10000)
/* #X-ARR(10) to #X-ARR(10000) are accessible
WRITE *LBOUND(#X-ARR)                    /* is 10
   *UBOUND(#X-ARR)                       /* is 10000
   *OCCURRENCE(#X-ARR)                   /* is 9991
#X-ARR(*) := 4711                        /* same as #X-ARR(10:10000) := 4711
/* resize array from current lower bound=10 to upper bound =1000
RESIZE ARRAY #X-ARR TO (*:1000)
/* #X-ARR(10) to #X-ARR(1000) are accessible
/* #X-ARR(1001) to #X-ARR(10000) are released
WRITE *LBOUND(#X-ARR)                    /* is 10
   *UBOUND(#X-ARR)                       /* is 1000
   *OCCURRENCE(#X-ARR)                   /* is 991
/* release all occurrences
REDUCE ARRAY #X-ARR TO  0
WRITE *OCCURRENCE(#X-ARR)                /* is 0
```

X-Group 配列のストレージ管理

X-group配列のオカレンス数を増減する場合、独立した次元と従属した次元を区別する必要があります。

直接指定されている（継承していない）X-group 配列または X-array の次元は、*独立*しています。

直接指定されていない（上位の配列から継承している）X-group 配列または X-array の次元は、*従属*しています。

X-array の独立した次元のみ、EXPAND、RESIZE、REDUCE の各ステートメントで変更できます。次元の変更は、その次元を独立した次元として所有している X-group 配列の名前を使用して行う必要があります。

独立した次元と従属した次元の例：

```
DEFINE  DATA LOCAL
1 #X-GROUP-ARR1(1:*)                  /* (1:*)
  2 #X-ARR1    (I4)                   /* (1:*)
  2 #X-ARR2    (I4/2:*)               /* (1:*,2:*)
  2 #X-GROUP-ARR2                     /* (1:*)
    3 #X-ARR3  (I4)                   /* (1:*)
    3 #X-ARR4  (I4/3:*)               /* (1:*,3:*)
    3 #X-ARR5  (I4/4:*, 5:*)          /* (1:*,4:*,5:*)
END-DEFINE
```

以下の表は、上記のプログラム内の各次元が独立しているのか従属しているのかを示しています。

名前	従属した次元	独立した次元
#X-GROUP-ARR1		(1:*)
#X-ARR1	(1:*)	
#X-ARR2	(1:*)	(2:*)
#X-GROUP-ARR2	(1:*)	
#X-ARR3	(1:*)	
#X-ARR4	(1:*)	(3:*)
#X-ARR5	(1:*)	(4:*,5:*)

従属した次元に使用できるインデックス表記は、単一のアスタリスク（*）、アスタリスクで定義した範囲（*:*）、または定義済みのインデックス範囲のみです。

これは、従属した次元の境界は変更できないことを示しています。

従属した次元のオカレンス数は、対応するグループ配列を操作することによってのみ変更できます。

```
EXPAND ARRAY #X-GROUP-ARR1 TO (1:11)       /* #X-ARR1(1:11) are allocated
                                           /* #X-ARR3(1:11) are allocated
EXPAND ARRAY #X-ARR2 TO (*:*, 2:12)        /* #X-ARR2(1:11, 2:12) are allocated
EXPAND ARRAY #X-ARR2 TO (1:*, 2:12)        /* same as before
EXPAND ARRAY #X-ARR2 TO (*  , 2:12)        /* same as before
```

```
EXPAND ARRAY #X-ARR4 TO (*:*, 3:13)        /* #X-ARR4(1:11, 3:13) are allocated
EXPAND ARRAY #X-ARR5 TO (*:*, 4:14, 5:15)  /* #X-ARR5(1:11, 4:14, 5:15) are allocated
```

EXPAND ステートメントは、任意の順序で記述できます。

従属した次元しか配列にないため、以下のような EXPAND ステートメントは使用できません。

```
EXPAND ARRAY #X-ARR1 TO ...
EXPAND ARRAY #X-GROUP-ARR2 TO ...
EXPAND ARRAY #X-ARR3 TO ...
```

X-array の参照

X-array のオカレンス数は、アクセスする前に EXPAND ステートメントまたは RESIZE ステートメントを使用して、明示的に割り当てる必要があります。 値を Tamino から取得する場合、ステートメント READ、FIND、GET ではオカレンスが暗黙的に割り当てられます。

一般的なルールとして、存在しない X-array のオカレンスにアクセスしようとするとランタイムエラーが発生します。 ただし、ステートメントの中には、全体範囲表記（#X-ARR(*) など）を使用して X-array のすべてのオカレンスを参照する場合は、実体を持たない X-array フィールドにアクセスしてもエラーにならないものがあります。 これは、以下に対して適用されます。

- CALL ステートメント内で使用されているパラメータ
- CALLNAT、PERFORM、SEND EVENT、OPEN DIALOG の各ステートメント内で使用されている、OPTIONAL として定義されているパラメータ
- COMPRESS ステートメント内で使用されているソースフィールド
- PRINT ステートメント内で指定されている出力フィールド
- RESET ステートメント内で参照されているフィールド

これらのステートメントの 1 つを使用して実体を持たない X-array のオカレンスを個別に参照すると、対応するエラーメッセージが発行されます。

例：

```
DEFINE DATA LOCAL
1 #X-ARR (A10/1:*) /* X-array only defined, but not allocated
END-DEFINE
RESET #X-ARR(*)    /* no error, because complete field referenced with (*)
RESET #X-ARR(1:3)  /* runtime error, because individual occurrences (1:3) are
```

```
referenced
END
```

配列参照のアスタリスク（*）表記は、次元のすべての範囲を表します。配列がX-arrayの場合、アスタリスクは、現在割り当てられている下限および上限のインデックス範囲を表します。この下限および上限は、*LBOUND および *UBOUND によって確認できます。

X-array を使用したパラメータ引き渡し

X-array をパラメータとして使用すると、以下の整合性チェックで定数の配列のように扱われます。

- フォーマット
- 長さ
- 次元
- オカレンス数

また、X-array パラメータは、ステートメント RESIZE、REDUCE、または EXPAND を使用することにより、オカレンス数を変更することもできます。X-arrayパラメータのサイズ変更が可能かどうかは、以下の 3 つの要素によって決まります。

- 使用するパラメータ引き渡しのタイプ（参照渡しまたは値渡し）
- 呼び出し元または X-array パラメータの定義
- 引き渡す X-array の範囲のタイプ（全体または一部）

以下の表は、X-array パラメータに EXPAND、RESIZE、REDUCE の各ステートメントを適用できる条件を示しています。

値渡しの例

呼び出し元	パラメータ		
	スタティック	変数（1:V）	X-array
スタティック	×	×	○
X-array の一部。例：CALLNAT...#XA(1:5)	×	×	○
X-array 全体。例：CALLNAT...#XA(*)	×	×	○

参照渡し／値渡し（結果）

呼び出し元	パラメータ			
	スタティック	変数（1:V）	固定の下限を持つ X-array。例： `DEFINE DATA PARAMETER 1 #PX (A10/1:*)`	固定の上限を持つ X-array。例： `DEFINE DATA PARAMETER 1 #PX (A10/*:1)`
スタティック	×	×	×	×
X-array の一部。例： `CALLNAT...#XA(1:5)`	×	×	×	×

固定の下限を持つ X-array の全体。例：
 	
`DEFINE DATA LOCAL`
`1 #XA(A10/1:*)`
`...`
`CALLNAT...#XA(*)`
	×	×	○	×

固定の上限を持つ X-array の全体。例：
 	
`DEFINE DATA LOCAL`
`1 #XA(A10/*:1)`
`...`
`CALLNAT...#XA(*)`
	×	×	×	○

X-group 配列を使用したパラメータ引き渡し

X-group配列の宣言は、グループの各要素が同じ上限および下限を持つことを暗黙的に意味します。 したがって、X-group配列フィールドの従属した次元のオカレンス数は、`RESIZE`、`REDUCE`、`EXPAND` の各ステートメントで X-group 配列のグループ名を指定したときにのみ変更できます（前述の「*X-Group 配列のストレージ管理*」を参照）。

X-group 配列のメンバは、パラメータデータエリアに定義されている X-group 配列にパラメータとして引き渡すことができます。 呼び出す側と呼び出される側のグループ構造は必ずしも同

一である必要はありません。 呼び出した側の X-group 配列に矛盾がない限り、呼び出された側は RESIZE、REDUCE および EXPAND を実行できます。

パラメータとして引き渡す **X-group** 配列の要素の例：

プログラム：

```
DEFINE DATA LOCAL
1 #X-GROUP-ARR1(1:*)                    /* (1:*)
   2 #X-ARR1 (I4)                       /* (1:*)
   2 #X-ARR2 (I4)                       /* (1:*)
1 #X-GROUP-ARR2(1:*)                    /* (1:*)
   2 #X-ARR3 (I4)                       /* (1:*)
   2 #X-ARR4 (I4)                       /* (1:*)
END-DEFINE
...
CALLNAT ... #X-ARR1(*) #X-ARR4(*)
...
END
```

サブプログラム：

```
DEFINE DATA PARAMETER
1 #X-GROUP-ARR(1:*)                     /* (1:*)
   2 #X-PAR1 (I4)                       /* (1:*)
   2 #X-PAR2 (I4)                       /* (1:*)
END-DEFINE
...
RESIZE ARRAY #X-GROUP-ARR to (1:5)
...
END
```

サブプログラムの RESIZE ステートメントは実行できません。 プログラムの X-group 配列で定義されているフィールドのオカレンス数と矛盾しています。

ダイナミック変数の X-array

ダイナミック変数の X-array の割り当てでは、最初に EXPAND ステートメントを使用してオカレンス数を指定し、後でそのオカレンス数に値を割り当てることによって変更できます。

例：

```
DEFINE DATA LOCAL
 1 #X-ARRAY(A/1:*)  DYNAMIC
END-DEFINE
EXPAND  ARRAY   #X-ARRAY TO (1:10)
  /* allocate #X-ARRAY(1) to #X-ARRAY(10) with zero length.
  /* *LENGTH(#X-ARRAY(1:10)) is zero
#X-ARRAY(*) := 'abc'
  /* #X-ARRAY(1:10) contains 'abc',
  /* *LENGTH(#X-ARRAY(1:10)) is 3
EXPAND  ARRAY   #X-ARRAY TO (1:20)
  /* allocate #X-ARRAY(11) to #X-ARRAY(20) with zero length
  /* *LENGTH(#X-ARRAY(11:20)) is zero
#X-ARRAY(11:20) := 'def'
  /* #X-ARRAY(11:20) contains 'def'
  /* *LENGTH(#X-ARRAY(11:20)) is 3
```

配列の上限および下限

システム変数*LBOUNDおよび*UBOUNDには、指定した次元（1、2、または3次元。複数指定可）に対する配列の、その時点での下限および上限が設定されます。

X-array にオカレンスが割り当てられていない場合、*LBOUND または *UBOUND は未定義のため、アクセスするとランタイムエラーが発生します。 ランタイムエラーを回避するには、*LBOUND または *UBOUND を評価する前に *OCCURRENCE を使用して、オカレンス数がゼロかどうかを確認します。

例：

```
IF *OCCURRENCE (#A) NE 0  AND  *UBOUND(#A) < 100 THEN ...
```

26 ユーザー定義関数

このchapterでは、次のトピックについて説明します。

ユーザー定義関数について

関数を使用すると、サブプログラムと同様に、データを受信したり、データを変更したり、結果を呼び出し側モジュールに渡したりすることができます。 サブプログラムを上回る関数の利点として、追加の一時変数を必要とすることなく、直接ステートメントおよび式でファンクションコールを使用できることが挙げられます。

通常、関数に渡されたパラメータに応じて関数内で結果が作成され、その結果が呼び出し側のオブジェクトに戻されます。 呼び出し側のモジュールに別の値を戻す場合は、パラメータを使用してこれを行います。「サブプログラム」を参照してください。

ファンクションコードが完全に実行された後、コントロールは呼び出し側のオブジェクトに戻され、プログラムはファンクションコールの後に続くステートメントを続行します。

詳細については、以下も参照してください。

- Natural ファンクションオブジェクトタイプ
- ファンクションコール
- Natural ステートメント `DEFINE FUNCTION`、`DEFINE PROTOTYPE`

ファンクションコールとサブプログラムコールの違い

以下の2つの例は、ファンクションコールとサブプログラムコールの使用法の違いを示しています。

ファンクションコールの使用例：

以下の例は、ファンクションコールを使用するプログラムオブジェクト、`DEFINE FUNCTION` ステートメントで作成したファンクション定義を含むファンクションオブジェクト、および`DEFINE PROTOTYPE` ステートメントで作成したコピーコードオブジェクトで構成されています。

プログラムオブジェクト：

```
/* Excerpt from a Natural program using a function call
INCLUDE C#ADD
WRITE #ADD(< 2,3 >) /* function call; no temporary variable necessary
END
```

ファンクションオブジェクト：

```
/* Natural function definition
DEFINE FUNCTION #ADD
  RETURNS (I4) BY VALUE
  DEFINE DATA PARAMETER
  1 #SUMMAND1 (I4) BY VALUE
  1 #SUMMAND2 (I4) BY VALUE
  END-DEFINE

  #ADD := #SUMMAND1 + #SUMMAND2
END-FUNCTION
END
```

コピーコードオブジェクト（"C#ADD" など）：

```
/* Natural copycode containing prototype
DEFINE PROTOTYPE #ADD
  RETURNS (I4)
  DEFINE DATA PARAMETER
  1 #SUMMAND1 (I4) BY VALUE
  1 #SUMMAND2 (I4) BY VALUE
  END-DEFINE
END-PROTOTYPE
```

サブプログラムを使用して同じ機能を実現する場合、一時変数を使用する必要があります。

サブプログラムの使用例：

以下の例は、サブプログラムオブジェクトを呼び出すプログラムオブジェクトで構成され、一時変数が使用されています。

プログラムオブジェクト：

```
/* Natural program using a subprogram
DEFINE DATA LOCAL
1 #RESULT (I4) INIT <0>           /* temporary variable
END-DEFINE

CALLNAT 'N#ADD' USING #RESULT 2 3 /* result is stored into #RESULT
WRITE #RESULT                     /* print out the result of the subprogram
END
```

サブプログラムオブジェクト（"N#ADD" など）：

```
/* Natural program using a subprogram
DEFINE DATA PARAMETER
1 #RETURN   (I4) BY VALUE RESULT
1 #SUMMAND1 (I4) BY VALUE
1 #SUMMAND2 (I4) BY VALUE
END-DEFINE

#RETURN := #SUMMAND1 + #SUMMAND2
END
```

ファンクション定義（DEFINE FUNCTION）

ファンクション定義には、ファンクションが呼び出されたときに実行されるNaturalコードが含まれています。サブプログラムと同様に、Naturalオブジェクトを作成する必要があります。この場合は、ファンクション定義を含む「ファンクション」タイプのオブジェクトを作成します。ファンクション定義は、Natural ステートメント `DEFINE FUNCTION` を使用して作成します。

ファンクションコール自体は、実行可能コードを含むどのオブジェクトタイプでも構いません。

プロトタイプ定義（DEFINE PROTOTYPE）

ファンクションコールをコンパイルするためには、戻り値の *format-length/array-definition* に関する情報が Natural に必要です。 その後、この情報はプロトタイプ定義のコンパイラから使用できるようになります。 この定義は、Natural ステートメント `DEFINE PROTOTYPE` を使用して作成します。 戻されるパラメータの定義を含めることもできます。この定義は、その後コンパイル時にチェックされます。

Natural では、ランタイム時に初めて 「呼び出し側」 のオブジェクトと 「呼び出された」 オブジェクト間が接続されるため、コンパイル時にどのファンクションタイプの戻り値で処理してい

るのかがコンピュータで認識されることはありません。これは、ファンクションを含むオブジェクトが必ずしもコンパイル時に存在する必要はないためです。このため、コンパイル時に *format-length/array-definition* を生成プログラムに組み込めるよう、プロトタイプ定義が作成されます。

プロトタイプ定義には決して実行可能コードが含まれないことに注意してください。プロトタイプ定義には、ファンクションコールに関する情報、つまり戻り値または戻されるパラメータの *format/length/array-definition* のみが含まれます。

記号および変数ファンクションコール

変数ファンクションコールを定義するには、常に `DEFINE PROTOTYPE VARIABLE` ステートメントを使用する必要があります。使用しなかった場合、ファンクションコールは暗黙的な記号ファンクションコールであると想定されます。

このトピックの詳細については、「*ファンクションコール*」を参照してください。

自動または暗黙的なプロトタイプ定義（APT）

明示的プロトタイプ定義（EPT）も `PT` 節も存在しない場合、生成プログラムでプロトタイプ定義の検索が行われます。詳細については、下記の「*可能なプロトタイプ定義の組み合わせ*」を参照してください。

プロトタイプキャスト（PT 節）

特定のファンクションの該当プロトタイプを見つけるために、Naturalによってファンクション名を持つプロトタイプが検索されます。それ以外の場合、ファンクションコールは記号ファンクションコールであると想定されます。この場合、ファンクションコールでキーワード `PT=` を使用して、ファンクション「署名」を定義する必要があります。

戻り値の中間結果（IR 節）

この節を使用すると、明示的または暗示的なプロトタイプ定義を使用しないで、ファンクションコールの戻り値のフォーマットおよび長さを指定することができます。つまり、中間結果を明示的に指定できます。 詳細については、「*ファンクションコール*」の *intermediate-result-definition* を参照してください。

可能なプロトタイプ定義の組み合わせ

以下の表では、DEFINE PROTOTYPE ステートメントとファンクションコールで使用可能な節、またはそのいずれかを使用した場合に可能なさまざまな構文の組み合わせに基づいて、プロトタイプ定義の影響を説明しています。 以下の組み合わせは、属するファンクションコールにのみ有効なファンクションプロトタイプ部分を定義するために使用できます。

- 明示的な **DEFINE PROTOTYPE** 定義（**EPT**）
  記号または変数ファンクションコール、パラメータ定義、戻り値定義を決定できます。
- プロトタイプキャスト（**PT** 節）
  パラメータ定義、戻り値定義を決定できます。
- 戻り値の中間結果（**IR** 節）
  戻り値定義を決定できます。

ケース	DEFINE PROTOTYPE での明示的プロトタイプ定義（EPT）	ファンクションコールの PT 節（PT）	ファンクションコールの IR 節（IR）		GP からのプロトタイプ定義の自動読み込み（APT）	プロトタイプの動作
1	x	x	x		-	SV（EPT）、PS（PT）、R（IR）
2	-	x	x		-	S、PS（PT）、R（IR）
3	x	-	x		-	SV（EPT）、PS（EPT）、R（IR）
4	-	-	x		x	S、PS（APT）、R（IR）
5	x	x	-		-	SV（EPT）、PS（PT）、R（PT）
6	-	x	-		-	S、PS（PT）、R（IR）
7	x	-	-		-	SV（EPT）、PS（EPT）、R（EPT）
8	-	-	-		x	S、PS（APT）、R（APT）

上記の意味は次に示すとおりです。

EPT	明示的な DEFINE PROTOTYPE ステートメント。
PT	プロトタイプキャスト（PT 節）。
IR	戻り値の中間結果（IR 節）。
APT	外部 GP 経由の自動プロトタイプ定義。
S	記号ファンクションコール。
V	変数ファンクションコール。
SV（EPT）	明示的プロトタイプ定義によって、記号ファンクションコールが実行されるか、変数ファンクションコールが実行されるかが決定されます。
R（IR）	ファンクションコールの IR 節によって戻り変数（R）が定義されます。
R（PT）	ファンクションコールの PT 節によって戻り変数（R）が定義されます。
R（EPT）	明示的な DEFINE PROTOTYPE ステートメントによって戻り変数（R）が定義されます。
PS（PT）	ファンクションコールの PT 節によって、パラメータ署名（PS）（戻り値定義なしのパラメータ定義）が定義されます。
PS（EPT）	明示的な DEFINE PROTOTYPE ステートメントによって、パラメータ署名（PS）（戻り値定義なしのパラメータ定義）が定義されます。
PS（APT）	生成プログラム（GP）からプロトタイプ定義を読み込むことよって、パラメータ署名（PS）が自動的に定義されます。

例えば、上記の表のケース 1 の動作は以下のとおりです。

明示的な DEFINE PROTOTYPE ステートメント（EPT）が使用され、ファンクションコールで PT および IR 節が定義された場合、どのよう動作になるでしょうか。

EPT 定義によって、記号ファンクションコールが実行されるか、変数ファンクションコールが実行されるかが決定されます。 DEFINE PROTOTYPE VARIABLE が事前に定義されている場合、変数ファンクションコールが想定されます。 パラメータ署名（戻り値定義なしの、すべてのパラメータのフォーマットおよび長さの定義）が PT 節によって定義され、戻り値のフォーマットおよび長さがファンクションコールの IR 節によって定義されます。 この場合、自動プロトタイプ定義（APT）は起動されません。

結論として、上記のケースから以下の一般的ルールが導き出されます。

- 変数ファンクションコールの場合、常に、コールの明示的プロトタイプ定義（EPT）が存在している必要があります。
- PT 節では、記号ファンクションコールであるか変数ファンクションコールであるかは決定されません。
- PT 節の定義によって、パラメータおよび戻り値の EPT 定義が上書きされます。
- IR 節の定義によって、戻り値定義が上書きされます。
- EPT も PT 節も存在しない場合、生成プログラムでプロトタイプ定義の検索が行われます（自動プロトタイプ定義）。

再帰的なファンクションコール

ファンクションを再帰的に呼び出す場合、ファンクションプロトタイプがファンクション定義に含まれているか、または INCLUDE ファイルで挿入される必要があります。

例：

ファンクションオブジェクト：

```
/* Function definition for calculation of the math. factorial
DEFINE FUNCTION #FACT
  RETURNS (I4) BY VALUE
  DEFINE DATA PARAMETER
  1 #PARA (I4) BY VALUE
  LOCAL
  1 #TEMP (I4)
  END-DEFINE

  /* Prototype definition is necessary
  INCLUDE C#FACT

  /* Program code
  IF #PARA=0
    #FACT := 1
  ELSE
    #TEMP := #PARA - 1
    #FACT := #PARA * #FACT(< #TEMP >)
  END-IF

END-FUNCTION
END
```

コピーコードオブジェクト（"C#FACT" など）：

```
/* Prototype definition is necessary
DEFINE PROTOTYPE #FACT
  RETURNS (I4)
  DEFINE DATA PARAMETER
  1 #PARA (I4) BY VALUE
```

```
  END-DEFINE
END-PROTOTYPE
```

プログラムオブジェクト：

```
/* Prototype definition
INCLUDE C#FACT

/* function call
WRITE #FACT(<12>)
END
```

ステートメントおよび式でのファンクションの動作

オペランドの代わりに、ステートメントまたは式で直接ファンクションコールを使用することができます。 ただし、使用できるのは、オペランドが修正不可能な場合のみです。

すべてのファンクションコールは、コンパイル時に分析される構文の順序に従って実行されます。 ファンクションコールの結果は内部一時変数に保存され、ステートメントまたは式に渡されます。

この一定の順序により、例えばステートメントの出力に意図しない影響を与えることなく、ファンクション内で標準出力を実行することができます。

例：

プログラム：

```
/* Natural program using a function call
INCLUDE CPRINT
PRINT 'before' #PRINT(<>) 'after'
END
```

ファンクションオブジェクト：

```
/* Natural function definition
/* function returns integer value 10
DEFINE FUNCTION #PRINT
  RETURNS (I4)
  WRITE '#PRINT'
  #PRINT := 10
```

```
END-FUNCTION
END
```

コピーコード（"CPRINT" など）：

```
DEFINE PROTOTYPE #PRINT END-PROTOTYPE
```

標準出力に送られる結果は以下のとおりです。

```
#PRINT
before       10 after
```

ステートメントとしてのファンクションの使用

ファンクションは、ステートメントおよび式とは関係なく、ステートメントとして呼び出すこともできます。 この場合、戻り値（定義されていると仮定）は考慮されません。

ただし、オプションのオペランドリストの後に独立したファンクションを宣言する場合、オペランドリストの後にセミコロンを付けて、ファンクションコールがオペランドリストの一部ではないことを明確にする必要があります。

例：

プログラムオブジェクト：

```
/* Natural program using a function call
DEFINE DATA LOCAL
1 #A (I4) INIT <1>
1 #B (I4) INIT <2>
END-DEFINE

INCLUDE CPROTO

WRITE #A #B
#PRINT_ADD(< 2,3 >) /* function call belongs to operand list just in front of it

WRITE '******'

WRITE #A #B;        /* semicolon separates operand list and function call
```

```
#PRINT_ADD(< 2,3 >) /* function call doesn't belong to the operand list
END
```

ファンクションオブジェクト：

```
/* Natural function definition
DEFINE FUNCTION #PRINT_ADD
  RETURNS (I4) BY VALUE
  DEFINE DATA PARAMETER
  1 #SUMMAND1 (I4) BY VALUE
  1 #SUMMAND2 (I4) BY VALUE
  END-DEFINE

  #PRINT_ADD := #SUMMAND1 + #SUMMAND2
  PRINT '#PRINT_ADD =' #PRINT_ADD
END-FUNCTION
END
```

コピーコードオブジェクト（"CPROTO" など）：

```
/* Natural copycode containing prototype
DEFINE PROTOTYPE #PRINT_ADD
  RETURNS (I4)
  DEFINE DATA PARAMETER
  1 #SUMMAND1 (I4) BY VALUE
  1 #SUMMAND2 (I4) BY VALUE
  END-DEFINE
END-PROTOTYPE
```

27 データベース内のデータへのアクセス

ここでは、Naturalを使用してデータベースのデータにアクセスするときのさまざまな面について説明します。

次のトピックについて説明します。

- **Natural** およびデータベースへのアクセス
- **Adabas** データベースのデータへのアクセス
- **SQL** データベースのデータへのアクセス
- **Tamino** データベースのデータへのアクセス

以下の項目も参照してください。

- Natural からのデータベースアクセス処理に影響を与える Natural プロファイルパラメータの概要については、「*プロファイルパラメータの概要*」の「*データベース管理*」を参照してください。

28 Natural およびデータベースへのアクセス

このchapterでは、異なるタイプのデータベース管理システムに対して Natural で提供される機能の概要について説明します。

次のトピックについて説明します。

Natural でサポートされるデータベース管理システム

Natural では、次のタイプのデータベース管理システム（DBMS）に対して特定のデータベースインターフェイスが提供されています。

- ネスト構造のリレーショナル DBMS（Adabas）
- SQL タイプの DBMS（Oracle、Sybase、Informix、MS SQL Server）
- XML タイプの DBMS（Tamino）

以下では次のトピックについて説明します。

Adabas

Natural は、統合された Adabas インターフェイスを使用することによって、ローカルマシンまたはリモートコンピュータ上の Adabas データベースにアクセスすることができます。 リモートアクセスの場合は、Entire Net-Work などのルーティングおよび通信用の追加のソフトウェアが必要です。いずれの場合も、Adabas データベースを実行しているホストマシンのタイプは、Natural ユーザーに透過的です。

Tamino

Natural for Tamino は、ローカルマシンまたはリモートホスト上の Tamino データベースサーバーに対し、ネイティブ HTTP プロトコルを使用したアクセス機能を提供します。 Tamino データベースには、Adabas データベースまたは SQL データベースでデータにアクセスするときと同じ方法でアクセスすることができます。

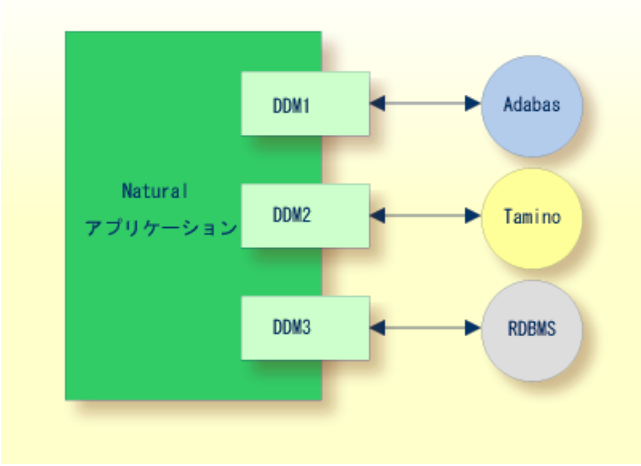
SQL データベース

Natural は、Entire Access を経由して SQL データベースシステムにアクセスします。Entire Access は、汎用的なインターフェイスおよびルーティングソフトウェアであり、Oracle、MS SQL Server、標準化された ODBC 接続などさまざまな SQL データベース管理システムをサポートします。 SQL データベース管理システムおよびサポートされるプラットフォームの詳細については、Entire Access のドキュメントを参照してください。 Natural コンフィグレーションの各側面に関する詳細については、Natural および Entire Access のドキュメントに記載されています。

データ定義モジュールを使用したアクセス

異なるデータベース管理システムへのアクセスを便利で透過的なものにするために、Naturalでは「データ定義モジュール」（DDM）という特殊なオブジェクトが使用されます。 このDDMによって、Naturalのデータ構造と、使用するデータベースシステムのデータ構造との間の接続が確立されます。該当するデータベース構造には、SQL データベースのテーブル、Adabas データベースのファイル、または Tamino データベースの doctype などがあります。 したがって、Natural アプリケーションからアクセスするデータベースの実際の構造は DDM によって隠されます。 DDM は、Natural DDM エディタを使用して作成します。

Natural は、特定のデータベースシステムの特定のデータ構造を表す DDM への参照を使用して、単一アプリケーション内から複数のデータベースタイプ（Adabas、Tamino、RDBMS）にアクセスすることができます。 次の図は、異なるタイプのデータベースに接続する 1 つのアプリケーションを示しています。



Natural のデータ操作言語

Naturalには、サポートするすべてのデータベースシステムに対して FIND、READ、STORE、DELETE などの同じ言語ステートメントを使用してNaturalアプリケーションからアクセスできるようにするデータ操作言語（DML）が組み込まれています。 アクセスするデータベースのタイプがわからなくても、Natural アプリケーションでこれらのステートメントを使用することができます。

Natural は、データベースシステムの実際のタイプをコンフィグレーションファイルから特定し、DML ステートメントをデータベース固有のコマンドに変換します。つまり、Adabas に対してはダイレクトコマンド、SQL データベースに対しては SQL ステートメント文字列とホスト変数構造、および Tamino データベースに対しては XQuery 要求を生成します。

一部の Natural DML ステートメントでは、すべてのデータベースタイプにはサポートされていない機能が提供されるため、このような機能の使用は特定データベースシステムのみに制限されます。 ステートメントドキュメントで、該当するデータベース固有の考慮事項を参照してください。

Natural の特殊な SQL ステートメント

「通常の」 Natural DML ステートメントに加えて、Natural では SQL データベースシステムと連携した、より限定して使用するための一連の SQL ステートメントが提供されます。『ステートメント』ドキュメントの「*SQL ステートメントの概要*」を参照してください。

ストアドプロシージャを使用するためのフレキシブル SQL や各種機能も SQL コマンドのセットに含まれています。 これらのステートメントは、SQL データベースへのアクセスにのみ使用可能です。Adabas をはじめとする SQL 以外のデータベースには有効ではありません。

29 Adabas データベースのデータへのアクセス

このchapterでは、Naturalを使用してAdabasデータベースのデータにアクセスするときのさまざまな面について説明します。

次のトピックについて説明します。

データ定義モジュール - DDM

Naturalがデータベースファイルにアクセスできるようにするには、物理データベースファイルの論理定義が必要です。このような論理ファイル定義は、データ定義モジュール（DDM）と呼ばれます。

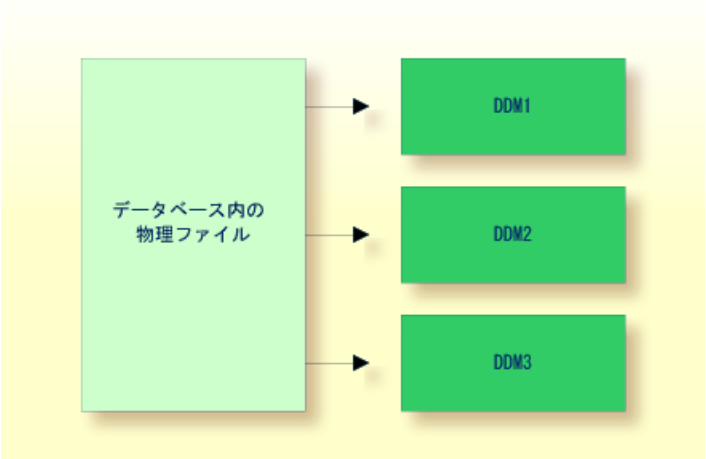
このsectionでは、次のトピックについて説明します。

- データ定義モジュールの使用
- DDM の管理
- DDM のリスト／表示

データ定義モジュールの使用

データ定義モジュールには、ファイルの個々のフィールドについての情報が含まれています。この情報は、Natural プログラムでこれらのフィールドを使用するために必要です。 DDM は、物理データベースファイルの論理ビューを構成しています。

データベースの各物理ファイルに、1 つ以上の DDM を定義できます。 各 DDM には、1 つ以上のデータビューを定義できます。ステートメントドキュメントの DEFINE DATA の「*ビューの定義*」を参照してください。



DDM は、Natural 管理者が Predict（Predict を使用できない場合は、対応する Natural 機能）を使用して定義します。

DDM の管理

各データベースフィールドについて、DDMにはデータベース内部のフィールド名の他に、Naturalプログラムで使用されるフィールドの名前である「外部」フィールド名も含まれています。また、フィールドのフォーマットと長さ、およびフィールドが DISPLAY または WRITE ステートメントで出力されるときに使われる各種の指定（列見出し、編集マスクなど）もDDMに定義されます。

DDMに定義されるフィールド属性については、『エディタ』ドキュメントの「*DDM エディタ*」セクションの「*DDM の編集 - フィールド属性定義*」を参照してください。

DDM のリスト／表示

希望する DDM の名前がわからない場合は、システムコマンド LIST VIEW を使用して、現在のライブラリで使用できる既存の DDM すべてのリストを取得できます。 このリストから、表示する DDM を選択できます。

LIST VIEW	すべてのビューのリストを表示します（DDM）。
LIST VIEW *view-name*	1つのビュー名を指定すると、指定したビューが表示されます。 *view-name* にアスタリスク表記を使用して、すべてのビューのリストを表示したり（*）、特定範囲のビューを表示したり（A* など）することができます。

データベース配列

Adabas は、マルチプルバリューフィールドおよびピリオディックグループの形で、データベース内の配列構造をサポートします。

このsectionでは、次のトピックについて説明します。

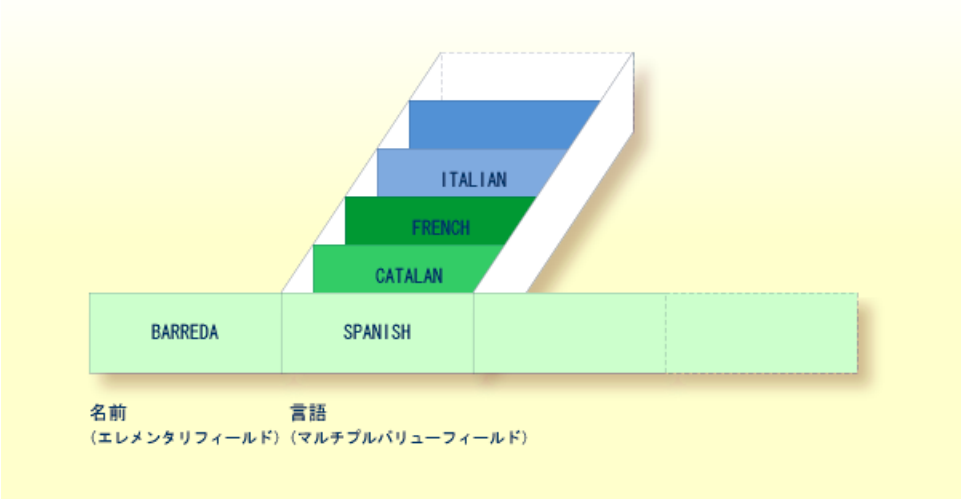
- マルチプルバリューフィールド
- ピリオディックグループ
- マルチプルバリューフィールドとピリオディックグループの参照
- ピリオディックグループ内のマルチプルバリューフィールド
- ピリオディックグループ内のマルチプルバリューフィールドの参照

- データベース配列の内部カウントの参照

マルチプルバリューフィールド

マルチプルバリューフィールドは、任意のレコード内に複数の値を持つことができるフィールドです。値の数は65534以下ですが、Adabasバージョンおよび FDT の定義に応じて異なります。

例：



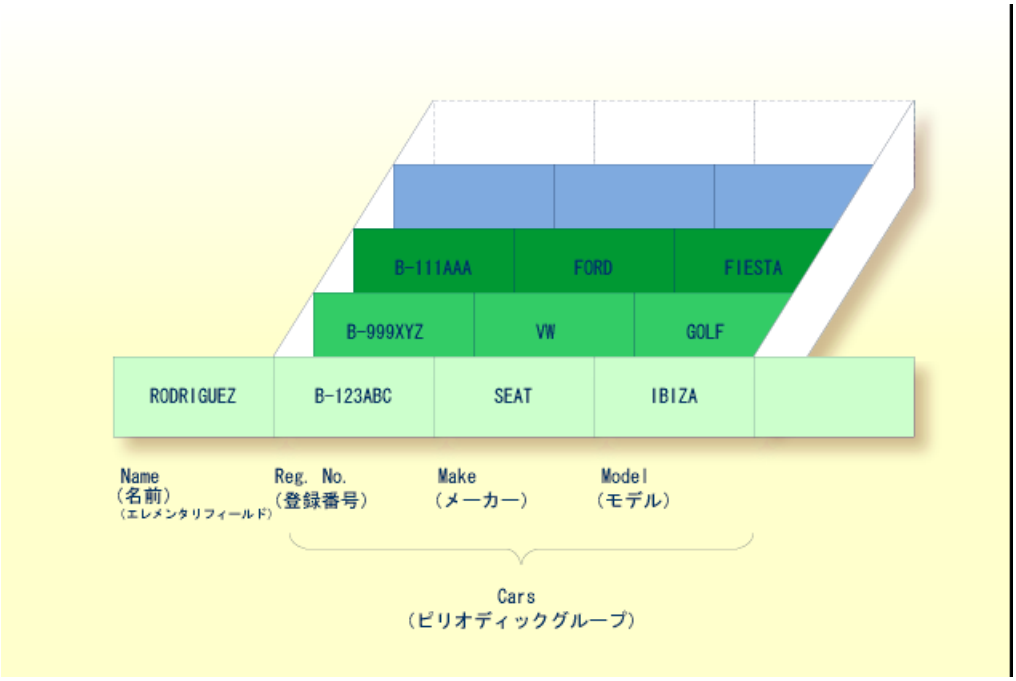
上記の図を EMPLOYEES ファイルのレコードと仮定すると、最初のフィールド（Name）はエレメンタリフィールドであり、1つの値、つまり従業員の名前のみを含めることができます。これに対して2つ目のフィールド（Languages）には、従業員が話す言語が含まれています。従業員は複数の言語を話せる可能性があるので、マルチプルバリューフィールドになっています。

ピリオディックグループ

ピリオディックグループは、任意のレコード内で複数のオカレンスを持つことができるエレメンタリフィールドまたはマルチプルバリューフィールドのグループです。オカレンスの数は65534以下ですが、Adabas バージョンおよび FDT の定義に応じて異なります。

マルチプルバリューフィールドに含まれる各値は、一般に 「オカレンス」 と呼ばれます。オカレンスの数は、フィールドに含まれる値の数であり、特定のオカレンスは特定の値を表します。同様に、ピリオディックグループでは、オカレンスは複数の値の集まりを表します。

例：



上の図が車両ファイル内のレコードであると仮定すると、最初のフィールド（Name）は人物の名前を含むエレメンタリフィールドです。「Cars」は、その人物が所有する自動車を含むピリオディックグループです。 ピリオディックグループは、各自動車の登録番号、メーカー、およびモデルの 3 つのフィールドで構成されています。 Cars の各オカレンスには 1 台の車に関する値が含まれます。

マルチプルバリューフィールドとピリオディックグループの参照

マルチプルバリューフィールドまたはピリオディックグループの1つ以上のオカレンスを参照するには、フィールド名の後に 「インデックス表記」 を指定します。

例：

次の例では、上述の例のマルチプルバリューフィールド LANGUAGES とピリオディックグループ CARS を使用します。

マルチプルバリューフィールド LANGUAGES の各値は、次のように参照できます。

LANGUAGES (1)	最初の値（"SPANISH"）を参照します。
LANGUAGES (X)	変数 X の値によって、参照する値が決まります。
LANGUAGES (1:3)	最初の 3 つの値（"SPANISH"、"CATALAN"、および "FRENCH"）を参照します。
LANGUAGES (6:10)	6 番目から 10 番目の値を参照します。
LANGUAGES (X:Y)	変数 X と変数 Y の値によって、参照する値が決まります。

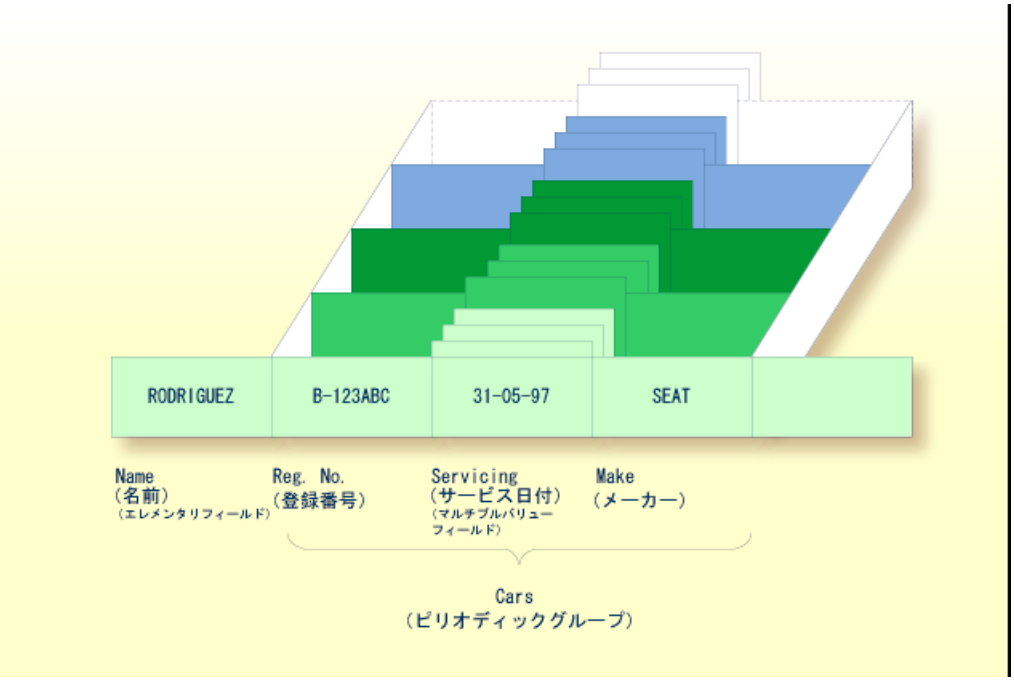
ピリオディックグループ CARS のさまざまなオカレンスも同様の方法で参照できます。

CARS (1)	最初のオカレンス（"B-123ABC/SEAT/IBIZA"）を参照します。
CARS (X)	変数 X の値によって、参照するオカレンスが決まります。
CARS (1:2)	最初の 2 つのオカレンス（"B-123ABC/SEAT/IBIZA" と "B-999XYZ/VW/GOLF"）を参照します。
CARS (4:7)	4 番目から 7 番目のオカレンスを参照します。
CARS (X:Y)	変数 X と変数 Y の値によって、参照するオカレンスが決まります。

ピリオディックグループ内のマルチプルバリューフィールド

Adabas 配列には、2 次元まで、つまり 1 つのピリオディックグループ内に 1 つのマルチプルバリューフィールドを含めることができます。

例：



上の図が車両ファイル内のレコードであると仮定すると、最初のフィールド（Name）は人物の名前を含むエレメンタリフィールドです。「Cars」は、その人物が所有する自動車を含むピリオディックグループです。 ピリオディックグループは、各自動車の登録番号、サービス日付、およびメーカーの3つのフィールドで構成されています。ピリオディックグループCars内のフィールドServicingはマルチプルバリューフィールドであり、各車の異なるサービス日付が含まれています。

ピリオディックグループ内のマルチプルバリューフィールドの参照

ピリオディックグループ内のマルチプルバリューフィールドの1つ以上のオカレンスを参照するには、フィールド名の後に 「インデックス表記」 を指定します。

例：

次の例では、上述の例のピリオディックグループ CARS 内のマルチプルバリューフィールド SERVICING を使用します。 マルチプルバリューフィールドの各値は、次のように参照できます。

SERVICING (1,1)	CARS の最初のオカレンスにある SERVICING の最初の値（"31-05-97"）を参照します。
SERVICING (1:5,1)	CARS の最初の 5 つのオカレンスにある SERVICING の最初の値を参照します。
SERVICING (1:5,1:10)	CARS の最初の 5 つのオカレンスにある SERVICING の最初の 10 個の値を参照します。

データベース配列の内部カウントの参照

レコードに値またはオカレンスがいくつ存在するかが不明なマルチプルバリューフィールドやピリオディックグループの参照が必要になることがあります。Adabasでは、各マルチプルバリューフィールドの値の数、および各ピリオディックグループのオカレンスの数の内部カウントが保持されています。このカウントは、READステートメントでフィールド名の直前に"C*"を指定することによって読み込むことができます。

カウントは、フォーマット／長さ N3 で返されます。詳細については、「*データベース配列の内部カウントの参照*」を参照してください。

例：

C*LANGUAGES	マルチプルバリューフィールド LANGUAGES の値の数を返します。
C*CARS	ピリオディックグループ CARS のオカレンスの数を返します。
C*SERVICING(1)	ピリオディックグループの最初のオカレンスにあるマルチプルバリューフィールド SERVICINGの値の数を返します。SERVICINGはピリオディックグループ内のマルチプルバリューフィールドであると仮定しています。

DEFINE DATA ビュー

Naturalプログラムでデータベースフィールドを使用できるようにするには、ビューでフィールドを指定する必要があります。

このsectionでは、次のトピックについて説明します。

- **データベースビューの使用**

データベースビューの使用

Naturalプログラムでデータベースフィールドを使用できるようにするには、ビューでフィールドを指定する必要があります。

ビューには次の内容を指定します。

- フィールドの取得元となるデータ定義モジュール（DDM）の名前。
- データベースフィールド自体の名前。つまり、データベース内部のショートネームではなく、ロングネーム。

データベースビューの定義

データベースビューは、次のいずれかの場所に定義します。

- プログラムの DEFINE DATA ステートメント内部
- プログラム外部のローカルデータエリア（LDA）またはグローバルデータエリア（GDA）。DEFINE DATA ステートメントを使用して、該当するデータエリアを参照します。詳細については、「フィールドの定義」セクションを参照してください。

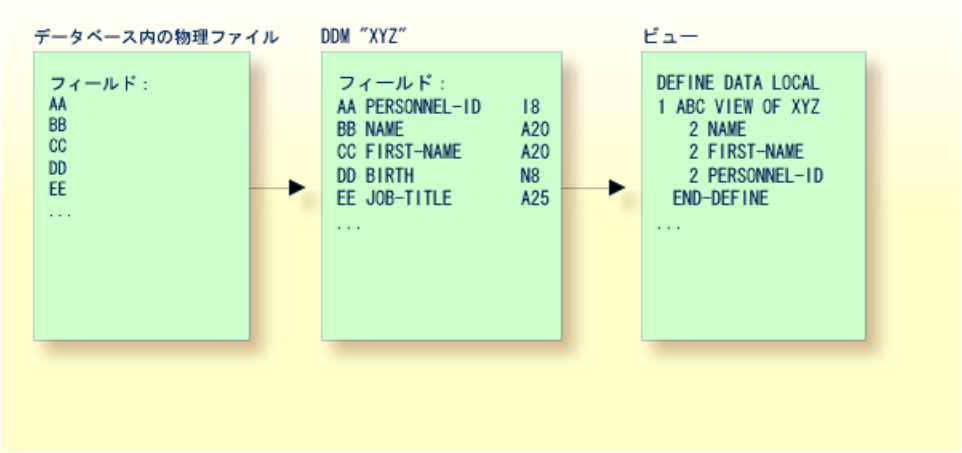
レベル 1 では、ビュー名を次のように指定します。

```
1 view-name VIEW OF ddm-name
```

view-name はビューに選択した名前、*ddm-name* はビューに指定されたフィールドの取得元となる DDM の名前です。

レベル 2 では、DDM のデータベースフィールドの名前を指定します。

次の図に示すように、ビューの名前は "ABC" で、DDM "XYZ" から取得したフィールド NAME、FIRST-NAME、および PERSONNEL-ID で構成されています。



データベースフィールドのフォーマットと長さは、基礎となるDDMですでに定義されているため、ビューに指定する必要はありません。

コンフィグレーションユーティリティのDBMS割り当てテーブル（『*コンフィグレーションユーティリティ*』ドキュメントの「*データベース管理システムの割り当て*」を参照）に指定されているタイプ ADA2 のデータベースでは、次のことが適用されます。

- Adabas LA/LB オプションである大きな英数字（LA）またはラージオブジェクト（LOB）のフィールドを使用する場合は、A20 や U20 などの固定フォーマット／長さ、および (A)DYNAMIC や U(DYNAMIC) などのダイナミックフォーマット／長さの両方を使用して、ビュー定義内にフィールドを指定できます。
- LA フィールドまたは LOB フィールドに関係のある長さインジケータフィールド L@... も、ビュー内に指定できます。

ビューは DDM 全体を含めたり、そのサブセットのみを含めたりすることができます。 ビューのフィールドの順番を、基礎となる DDM と同じにする必要はありません。

ビュー名は、アクセスするデータベースを決定するためにデータベースアクセスステートメントで使用されます。「データベースアクセスのステートメント」を参照してください。

データベースアクセスのステートメント

データベースからデータを読み込むには、次のステートメントを使用できます。

READ	指定した順番でデータベースからレコードの範囲を選択します。
FIND	指定した検索条件に一致するレコードをデータベースから選択します。
HISTOGRAM	1つのデータベースフィールドの値のみを読み込みます。または、指定した検索条件に一致するレコードの数を決定します。

READ ステートメント

次のトピックについて説明します。

- READ ステートメントの使用
- READ ステートメントの基本構文
- READ ステートメントの例
- 読み込むレコード数の制限
- STARTING／ENDING 節
- WHERE 節
- READ ステートメントのその他の例

READ ステートメントの使用

READ ステートメントは、データベースからレコードを読み取るために使用します。 レコードがデータベースから読まれる順番は次のとおりです。

- データベースに物理的に保存されている順番（READ IN PHYSICAL SEQUENCE）
- Adabas 内部シーケンス番号の順番（READ BY ISN）
- ディスクリプタフィールドの値の順番（READ IN LOGICAL SEQUENCE）

このドキュメントでは、READ IN LOGICAL SEQUENCE のみを取り上げます。これは最も頻繁に使用される形の READ ステートメントです。

他の 2 つのオプションの詳細については、『ステートメント』ドキュメントの READ ステートメントの説明を参照してください。

READ ステートメントの基本構文

READ ステートメントの基本構文は次のとおりです。

```
READ view IN LOGICAL SEQUENCE BY descriptor
```

または、以下のように短くすることができます。

```
READ view LOGICAL BY descriptor
```

各項目の意味を次に示します。

view	DEFINE DATA ステートメントで定義されるビューの名前（「*DEFINE DATA ビュー*」を参照）。
descriptor	そのビューで定義されるデータベースフィールドの名前。このフィールドの値によって、データベースから読み込まれるレコードの順番が決まります。

ディスクリプタを指定する場合は、キーワード LOGICAL を指定する必要はありません。

```
READ view BY descriptor
```

ディスクリプタを指定しない場合は、**DDM** の "デフォルト順" にデフォルトディスクリプタとして定義されたフィールドの値の順番でレコードが読み込まれます。 ただし、ディスクリプタを指定しない場合は、次のようにキーワード LOGICAL を指定する必要があります。

```
READ view LOGICAL
```

READ ステートメントの例

```
** Example 'READX01': READ
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 PERSONNEL-ID
  2 JOB-TITLE
END-DEFINE
*
READ (6) MYVIEW BY NAME
  DISPLAY NAME PERSONNEL-ID JOB-TITLE
```

```
END-READ
END
```

プログラム READX01 の出力：

上記の例の READ ステートメントでは、EMPLOYEES ファイルのレコードが各従業員の姓のアルファベット順に読み込まれます。

プログラムによって次のような出力が作成され、各従業員の情報がそれぞれの姓のアルファベット順に表示されます。

```
Page      1                                                 04-11-11  14:15:54

        NAME         PERSONNEL          CURRENT
                        ID             POSITION
-------------------- --------- -------------------------

ABELLAN              60008339  MAQUINISTA
ACHIESON             30000231  DATA BASE ADMINISTRATOR
ADAM                 50005800  CHEF DE SERVICE
ADKINSON             20008800  PROGRAMMER
ADKINSON             20009800  DBA
ADKINSON             2001100
```

従業員を生年月日順にリストするレポートを作成するためにレコードを読み込む場合の適切な READ ステートメントは次のようになります。

```
READ MYVIEW BY BIRTH
```

指定できるのは、基礎となる **DDM** で「ディスクリプタ」 として定義されているフィールドのみです。サブディスクリプタ、スーパーディスクリプタ、ハイパーディスクリプタ、フォネティックディスクリプタ、またはノンディスクリプタの場合もあります。

読み込むレコード数の制限

上記のプログラム例で示されているように、キーワード READ の後にカッコで囲んだ数字を次のように指定することによって、読み込まれるレコード数を制限できます。

```
READ (6) MYVIEW BY NAME
```

上記の例では、READ ステートメントで 6 件を超えるレコードは読み込まれなくなります。

リミット表記がない場合、上記の READ ステートメントによって、EMPLOYEES ファイルのすべてのレコードが姓の順に A から Z まで読み込まれます。

STARTING／ENDING 節

READ ステートメントでは、ディスクリプタフィールドの*値*に基づいてレコードの選択を限定することもできます。 BY 節または WITH 節で EQUAL TO/STARTING FROM オプションを設定することによって、読み込みを開始する値を指定できます。 THRU/ENDING AT オプションを追加して、読み込みを終了する値を論理順で指定することもできます。

例えば、"TRAINEE" で開始して "Z" まで継続する職種の順番で従業員をリストするには、次のいずれかのステートメントを使用します。

```
READ MYVIEW WITH JOB-TITLE = 'TRAINEE'
READ MYVIEW WITH JOB-TITLE STARTING from 'TRAINEE'
READ MYVIEW BY JOB-TITLE = 'TRAINEE'
READ MYVIEW BY JOB-TITLE STARTING from 'TRAINEE'
```

等号（=）または STARTING FROM オプションの右側の値はアポストロフィで囲む必要があります。 値が数値の場合は、このテキスト表記は不要です。

BY オプションを使用するときは、WITH オプションを使用できません。この逆も同様です。

読み込む一連のレコードは、THRU 節または ENDING AT 節で終了制限を追加することによって、より厳密に指定できます。

職種が "TRAINEE" のレコードだけを読み込むには、次のように指定します。

```
READ MYVIEW BY JOB-TITLE STARTING from 'TRAINEE' THRU 'TRAINEE'
READ MYVIEW WITH JOB-TITLE EQUAL TO 'TRAINEE'
                           ENDING AT 'TRAINEE'
```

職種が "A" または "B" で始まるレコードだけを読み込むには、次のように指定します。

```
READ MYVIEW BY JOB-TITLE = 'A' THRU 'C'
READ MYVIEW WITH JOB-TITLE STARTING from 'A' ENDING AT 'C'
```

値は、THRU/ENDING AT の後に指定された値まで、その値を含めて読み込まれます。 上記の 2 つの例では、職種が "A" または "B" で始まる全レコードが読み込まれます。職種 "C" があれば、これも読み込まれますが、次に高い値である "CA" は読み込まれません。

WHERE 節

WHERE 節を使用して、読み込むレコードをさらに限定することができます。

例えば、米国通貨で給与が支払われ、職種が "TRAINEE" で始まる従業員のみを必要とする場合は、次のように指定します。

```
READ MYVIEW WITH JOB-TITLE = 'TRAINEE'
            WHERE CURR-CODE = 'USD'
```

WHERE 節は、次のように BY 節とともに使用することもできます。

```
READ MYVIEW BY NAME
            WHERE SALARY = 20000
```

WHERE 節は、次の 2 つの点で BY/WITH 節と異なります。

- WHERE 節に指定するフィールドがディスクリプタである必要はありません。
- WHERE オプションに続く式は論理条件です。

WHERE 節では次の論理演算子が有効です。

EQUAL	EQ	=
NOT EQUAL TO	NE	¬=
LESS THAN	LT	<
LESS THAN OR EQUAL TO	LE	<=
GREATER THAN	GT	>
GREATER THAN OR EQUAL TO	GE	>=

次のプログラムは、STARTING FROM、ENDING AT、および WHERE の各節の使い方を説明するものです。

```
** Example 'READX02': READ  (with STARTING, ENDING and WHERE clause)
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
  2 INCOME     (1:2)
    3 CURR-CODE
    3 SALARY
    3 BONUS    (1:1)
```

```
END-DEFINE
*
READ (3) MYVIEW WITH  JOB-TITLE
STARTING FROM 'TRAINEE' ENDING AT 'TRAINEE'
                 WHERE CURR-CODE (*) = 'USD'
  DISPLAY NOTITLE NAME / JOB-TITLE 5X INCOME (1:2)
  SKIP 1
END-READ
END
```

プログラム READX02 の出力：

```
          NAME                              INCOME
         CURRENT
        POSITION              CURRENCY    ANNUAL     BONUS
                                CODE      SALARY
-------------------------     --------- ---------- ----------

SENKO                         USD            23000          0
TRAINEE                       USD            21800          0

BANGART                       USD            25000          0
TRAINEE                       USD            23000          0

LINCOLN                       USD            24000          0
TRAINEE                       USD            22000          0
```

READ ステートメントのその他の例

次の例のプログラムを参照してください。

■ *READX03 - READ ステートメント*

FIND ステートメント

次のトピックについて説明します。

- FIND ステートメントの使用
- FIND ステートメントの基本構文
- 処理するレコード数の制限
- WHERE 節
- WHERE 節が含まれる FIND ステートメントの例
- IF NO RECORDS FOUND 条件

FIND ステートメントの使用

FIND ステートメントは、指定した検索条件に一致するレコードをデータベースから選択するために使用します。

FIND ステートメントの基本構文

FIND ステートメントの基本構文は次のとおりです。

```
FIND RECORDS IN view WITH field = value
```

または、以下のように短くすることができます。

```
FIND view WITH field = value
```

各項目の意味を次に示します。

view	DEFINE DATA ステートメントで定義されるビューの名前（「*DEFINE DATA ビュー*」を参照）。
field	そのビューで定義されるデータベースフィールドの名前。

*field*に指定できるのは、基礎となる **DDM** で「ディスクリプタ」として定義されているフィールドのみです。サブディスクリプタ、スーパーディスクリプタ、ハイパーディスクリプタ、またはフォネティックディスクリプタの場合もあります。

完全な構文については、FIND ステートメントのドキュメントを参照してください。

処理するレコード数の制限

上述した READ ステートメントの場合と同様に、キーワード FIND の後にカッコで囲んだ数字を指定することによって、処理するレコード数を制限できます。

```
FIND (6) RECORDS IN MYVIEW WITH NAME = 'CLEGG'
```

上記の例では、検索条件に一致する最初の 6 つのレコードだけが処理されます。

リミット表記がない場合は、検索条件に一致するすべてのレコードが処理されます。

Note: FIND ステートメントに WHERE 節（下記参照）が含まれている場合、WHERE 節の結果として拒否されるレコードは制限に対して*カウントされません*。

WHERE 節

FIND ステートメントの WHERE 節を使用すると、WITH 節で選択したレコードが読み込まれた後、このレコードの処理が実行される*前*に評価される追加の選択条件を指定できます。

WHERE 節が含まれる FIND ステートメントの例

```
** Example 'FINDX01': FIND (with WHERE)
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 JOB-TITLE
  2 CITY
END-DEFINE
*
FIND MYVIEW WITH CITY = 'PARIS'
            WHERE JOB-TITLE = 'INGENIEUR COMMERCIAL'
  DISPLAY NOTITLE CITY JOB-TITLE PERSONNEL-ID NAME
END-FIND
END
```

Note: この例では、WITH 節と WHERE 節の*両方*の条件に一致するレコードだけが DISPLAY ステートメントで処理されます。

プログラム FINDX01 の出力：

```
        CITY                  CURRENT           PERSONNEL         NAME
                             POSITION              ID
-------------------- ------------------------- --------- --------------------

PARIS                INGENIEUR COMMERCIAL      50007300  CAHN
PARIS                INGENIEUR COMMERCIAL      50006500  MAZUY
PARIS                INGENIEUR COMMERCIAL      50004700  FAURIE
PARIS                INGENIEUR COMMERCIAL      50004400  VALLY
PARIS                INGENIEUR COMMERCIAL      50002800  BRETON
```

```
PARIS                INGENIEUR COMMERCIAL      50001000  GIGLEUX
PARIS                INGENIEUR COMMERCIAL      50000400  KORAB-BRZOZOWSKI
```

IF NO RECORDS FOUND 条件

WITH 節と WHERE 節に指定した検索条件に一致するレコードが見つからない場合、FIND 処理ループ内のステートメントは実行されません。上記の例では、DISPLAY ステートメントが実行されないため、従業員データは表示されません。

ただし、FIND ステートメントでは IF NO RECORDS FOUND 節も提供されます。これにより、検索条件に一致するレコードがない場合に実行する処理を指定できます。

例：

```
** Example 'FINDX02': FIND (with IF NO RECORDS FOUND)
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
END-DEFINE
*
FIND MYVIEW WITH NAME = 'BLACKSMITH'
  IF NO RECORDS FOUND
    WRITE 'NO PERSON FOUND.'
  END-NOREC
  DISPLAY NAME FIRST-NAME
END-FIND
END
```

上記のプログラムでは、NAME フィールドに "BLACKSMITH" の値があるすべてのレコードが選択されます。選択された各レコードについて、姓（NAME）と名前（FIRST-NAME）が表示されます。NAME = 'BLACKSMITH' のレコードがファイルに見つからない場合は、IF NO RECORDS FOUND 節内の WRITE ステートメントが実行されます。

プログラム FINDX02 の出力：

```
Page      1                                              04-11-11  14:15:54

        NAME              FIRST-NAME
-------------------- --------------------
```

```
NO PERSON FOUND.
```

FIND ステートメントのその他の例

次の例のプログラムを参照してください。

- *FINDX07 - FIND（複数節を含む）*
- *FINDX08 - FIND（LIMIT を含む）*
- *FINDX09 - FIND（*NUMBER、*COUNTER、*ISN を使用）*
- *FINDX10 - FIND（READ との組み合わせ）*
- *FINDX11 - FIND NUMBER（*NUMBER を含む）*

HISTOGRAM ステートメント

次のトピックについて説明します。

- HISTOGRAM ステートメントの使用
- HISTOGRAM ステートメントの基本構文
- 読み込む値の数の制限
- STARTING／ENDING 節
- WHERE 節
- HISTOGRAM ステートメントの例

HISTOGRAM ステートメントの使用

HISTOGRAM ステートメントは、1 つのデータベースフィールドの値だけを読み込むか、または指定した検索条件に一致するレコード数を決定するために使用します。

HISTOGRAM ステートメントでは、HISTOGRAM ステートメントに指定されたもの以外のデータベースフィールドへのアクセスは提供されません。

HISTOGRAM ステートメントの基本構文

HISTOGRAM ステートメントの基本構文は次のとおりです。

```
HISTOGRAM VALUE IN view FOR field
```

または、以下のように短くすることができます。

```
HISTOGRAM view FOR field
```

各項目の意味を次に示します。

view	DEFINE DATA ステートメントで定義されるビューの名前（「*DEFINE DATA ビュー*」を参照）。
field	そのビューで定義されるデータベースフィールドの名前。

完全な構文については、HISTOGRAM ステートメントのドキュメントを参照してください。

読み込む値の数の制限

READ ステートメントの場合と同様に、キーワード HISTOGRAM の後にカッコで囲んだ数字を指定することによって、読み込まれるレコード数を制限できます。

```
HISTOGRAM (6) MYVIEW FOR NAME
```

上記の例では、フィールド NAME の最初の 6 つの値だけが読み込まれます。

リミット表記がない場合は、すべての値が読み込まれます。

STARTING／ENDING 節

READ ステートメントと同様に、HISTOGRAM ステートメントでも、開始値と終了値を指定して読み込む値の範囲を絞り込むために、STARTING FROM 節と ENDING AT（またはTHRU）節が提供されます。

例：

```
HISTOGRAM MYVIEW FOR NAME STARTING from 'BOUCHARD'
```

```
HISTOGRAM MYVIEW FOR NAME STARTING from 'BOUCHARD' ENDING AT 'LANIER'
HISTOGRAM MYVIEW FOR NAME from 'BLOOM' THRU 'ROESER'
```

WHERE 節

HISTOGRAMステートメントでも、値が読み込まれた後、その値の処理が実行される*前*に評価される追加の選択条件を指定できる WHERE 節が提供されます。 WHERE 節に指定するフィールドは、HISTOGRAM ステートメントの主節のフィールドと同じである必要があります。

HISTOGRAM ステートメントの例

```
** Example 'HISTOX01': HISTOGRAM
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 CITY
END-DEFINE
*
LIMIT 8
HISTOGRAM MYVIEW CITY STARTING FROM 'M'
  DISPLAY NOTITLE CITY 'NUMBER OF/PERSONS' *NUMBER *COUNTER
END-HISTOGRAM
END
```

上記のプログラムでは、システム変数 *NUMBER と *COUNTER も HISTOGRAM ステートメントで評価され、DISPLAY ステートメントで出力されます。 *NUMBER には最後に読み込まれた値が含まれるデータベースレコードの数が入り、*COUNTER には読み込まれた値の合計数が入ります。

プログラム HISTOX01 の出力：

```
        CITY          NUMBER OF     CNT
                       PERSONS
-------------------- ----------- -----------

MADISON                        3           1
MADRID                        41           2
MAILLY LE CAMP                 1           3
MAMERS                         1           4
MANSFIELD                      4           5
MARSEILLE                      2           6
```

```
MATLOCK                        1          7
MELBOURNE                      2          8
```

MULTI-FETCH 節

このsectionは、Adabas データベースのマルチフェッチレコード検索機能について説明します。

このsectionで説明するマルチフェッチ機能は、コンフィグレーションユーティリティの DBMS 割り当てテーブルで定義できる ADA というタイプのデータベースに対してのみサポートされます。『*コンフィグレーションユーティリティ*』ドキュメントの「*データベース管理システムの割り当て*」を参照してください。 MULTI-FETCH 節はデータベースタイプ ADA2 ではサポートされません。

次のトピックについて説明します。

- マルチフェッチ機能の目的
- サポートされるステートメント
- マルチフェッチ使用時の考慮事項

マルチフェッチ機能の目的

標準モードのNaturalは、単一のデータベースコールでは複数のレコードを読み込みまず、フェッチごとに 1 つのレコードを取得するモードで常に稼働します。 このような稼働は堅実で安定していますが、大量のデータベースレコードの処理には時間がかかる場合があります。 これらのプログラムのパフォーマンスを向上させるために、マルチフェッチ処理を使用できます。

デフォルトでは、Natural はシングルフェッチを使用して Adabas データベースからデータを検索します。 このデフォルトは Natural のプロファイルパラメータ MFSET を使用して設定できます。

ON（マルチフェッチ）および OFF（シングルフェッチ）の値でデフォルトの動作を定義します。 MFSET が NEVER に設定されると、Natural は常にシングルフェッチモードを使用し、ステートメントレベルでのすべての設定を無視します。

デフォルトの処理モードは、ステートメントレベルで変更することもできます。

サポートされるステートメント

マルチフェッチ処理は、データベース更新を伴わない次のステートメントに対してサポートされます。

- FIND
- READ
- HISTOGRAM

構文の詳細については、FIND、READ、または HISTOGRAM の各ステートメントの MULTI-FETCH 節に関する説明を参照してください。

マルチフェッチ使用時の考慮事項

同じ Adabas ファイルを参照するネストされたデータベースループで、内側のループの 1 つに UPDATE ステートメントが含まれる場合、Natural では更新後の値で外側のループの処理が続行されます。 これは、マルチフェッチモードでは、外側のループのシーケンス制御に使用されるディスクリプタの値が内側のデータベースループによって更新される場合に、外側の論理 READ ループの位置を変更する必要があることを示します。 このように処理することによって現在のディスクリプタに矛盾が発生した場合は、エラーが返されます。 このような状況を避けるために、外側のデータベースループのマルチフェッチを無効にすることをお勧めします。

一般には、マルチフェッチモードによって、Adabasデータベースにアクセスするときのパフォーマンスが改善されます。 ただし、一部のケースで、特にデータベース更新が伴う場合は、パフォーマンスの向上にはシングルフェッチを使用する方が有利なときがあります。

データベース処理ループ

このsectionでは、FIND、READ、または HISTOGRAM の各ステートメントの結果としてデータベースから選択されたデータの処理に必要な処理ループについて説明します。

次のトピックについて説明します。

- データベース処理ループの作成
- 処理ループの階層
- 同じファイルにアクセスする FIND ループのネストの例

- READ および FIND ステートメントのネストのその他の例

データベース処理ループの作成

Natural では、FIND、READ、または HISTOGRAM ステートメントの結果としてデータベースから選択されたデータの処理に必要な処理ループが自動的に作成されます。

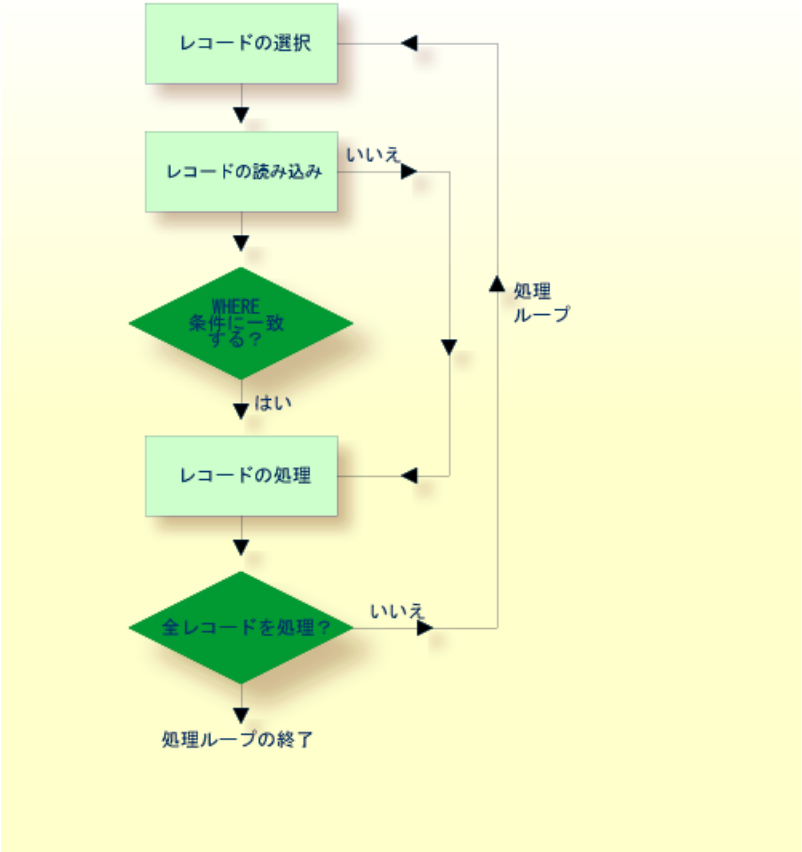
例：

次の例では、FIND ループを使用して、NAME フィールドに "ADKINSON" という値が含まれるすべてのレコードを EMPLOYEES ファイルから選択し、選択したレコードを処理します。 この例では、選択された各レコードの特定のフィールドを表示する処理が含まれます。

```
** Example 'FINDX03': FIND
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
END-DEFINE
*
FIND MYVIEW WITH NAME = 'ADKINSON'
  DISPLAY NAME FIRST-NAME CITY
END-FIND
END
```

WITH 節に加えて、FIND ステートメントに WHERE 節が含まれる場合は、WITH 節の結果として選択されたレコードのうち、さらに、WHERE 条件に一致するものだけが処理されます。

次の図は、データベース処理ループのフローロジックを示しています。



処理ループの階層

FIND および（または）READ ステートメントを複数使用することによって、次の例に示すように、処理ループの階層が作成されます。

処理ループの階層の例

```
** Example 'FINDX04': FIND  (two FIND statements nested)
************************************************************************
DEFINE DATA LOCAL
1 PERSONVIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
1 AUTOVIEW VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
  2 MODEL
END-DEFINE
*
EMP. FIND PERSONVIEW WITH NAME = 'ADKINSON'
  VEH. FIND AUTOVIEW WITH PERSONNEL-ID = PERSONNEL-ID (EMP.)
```

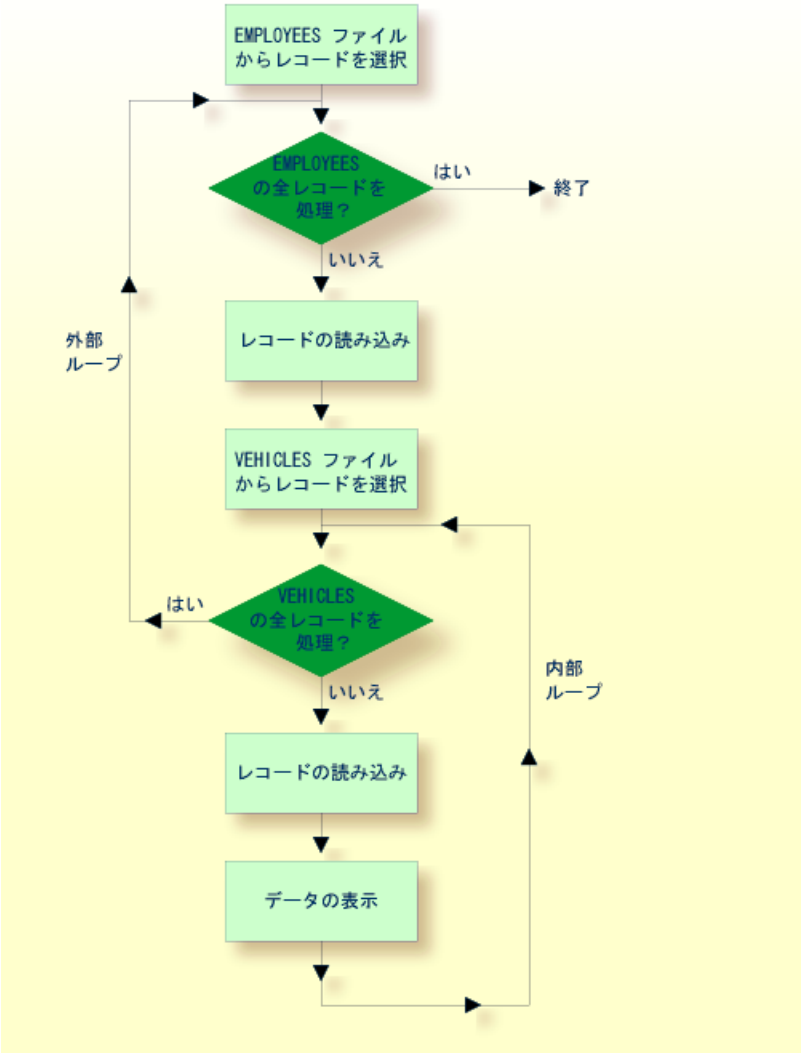
```
    DISPLAY NAME MAKE MODEL
  END-FIND
END-FIND
END
```

上記のプログラムでは、"ADKINSON" という名前のすべての従業員が EMPLOYEES ファイルから選択されます。 選択された各レコード（従業員）は、その後、次のように処理されます。

1. VEHICLES ファイルから車を選択するために、1 つ目の FIND ステートメントで EMPLOYEES ファイルから選択されたレコードの PERSONNEL-ID を選択条件に使用して、2 つ目の FIND ステートメントが実行されます。
2. 選択された各従業員の NAME が表示されます。この情報は EMPLOYEES ファイルから取得されます。その従業員が所有する各車の MAKE と MODEL も表示されます。この情報は VEHICLES ファイルから取得されます。

2 つ目の FIND ステートメントでは、次の図に示すように、1 つ目の FIND ステートメントの外部処理ループ内に内部処理ループが作成されます。

この図は、上述のプログラム例における処理ループの階層のフローロジックを示しています。



同じファイルにアクセスする FIND ループのネストの例

階層の両方のレベルで同じファイルが使用される処理ループの階層を構成することもできます。

```
** Example 'FINDX05': FIND (two FIND statements on same file nested)
************************************************************************
DEFINE DATA LOCAL
1 PERSONVIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
1 #NAME (A40)
END-DEFINE
*
WRITE TITLE LEFT JUSTIFIED
```

```
  'PEOPLE IN SAME CITY AS:' #NAME / 'CITY:' CITY SKIP 1
*
FIND PERSONVIEW WITH NAME = 'JONES'
                WHERE FIRST-NAME = 'LAUREL'
  COMPRESS NAME FIRST-NAME INTO #NAME
  /*
  FIND PERSONVIEW WITH CITY = CITY
    DISPLAY NAME FIRST-NAME CITY
  END-FIND
END-FIND
END
```

上記のプログラムでは、まず姓が "JONES" で名前が "LAUREL" の従業員が EMPLOYEES ファイルからすべて選択されます。 次に、同じ都市に住んでいるすべての従業員が EMPLOYEES ファイルから選択され、そのリストが作成されます。 DISPLAY ステートメントで表示されるすべてのフィールド値は、2 つ目の FIND ステートメントから取得されます。

プログラム FINDX05 の出力：

```
PEOPLE IN SAME CITY AS: JONES LAUREL
CITY: BALTIMORE

        NAME              FIRST-NAME              CITY
-------------------- -------------------- --------------------

JENSON               MARTHA               BALTIMORE
LAWLER               EDDIE                BALTIMORE
FORREST              CLARA                BALTIMORE
ALEXANDER            GIL                  BALTIMORE
NEEDHAM              SUNNY                BALTIMORE
ZINN                 CARLOS               BALTIMORE
JONES                LAUREL               BALTIMORE
```

READ および FIND ステートメントのネストのその他の例

次の例のプログラムを参照してください。

- *READX04 - READ ステートメント（FIND およびシステム変数 *NUMBER と *COUNTER との組み合わせ）*
- *LIMITX01 - LIMIT ステートメント（READ、FIND ループ処理）*

データベース更新 - トランザクション処理

このsectionでは、トランザクションに基づいてNaturalでデータベース更新処理が実行される方法について説明します。

次のトピックについて説明します。

- 論理トランザクション
- レコードホールドロジック
- トランザクションのバックアウト
- トランザクションの再スタート
- トランザクションデータを使用したトランザクション再スタートの例

論理トランザクション

Naturalでは、トランザクションに基づいてデータベース更新処理が実行されます。つまり、すべてのデータベース更新要求は論理トランザクション単位で処理されます。論理トランザクションは、データベースに含まれている情報が論理的に一貫性を持っていることを確実にするために、完全に実行されなければならない最小の業務ユニットです。業務ユニットの定義はユーザーが行います。

論理トランザクションは、1つ以上のデータベースファイルに関連する1つ以上の更新ステートメント（DELETE、STORE、UPDATE）で構成することができます。 また、論理トランザクションは、複数の Natural プログラムにまたがることもできます。

論理トランザクションは、レコードが「ホールド」状態におかれたときに開始します。Naturalでは、レコードが更新のために読み込まれるとき、例えば、FINDループにUPDATEステートメントやDELETEステートメントが含まれる場合に、この処理が自動的に行われます。

論理トランザクションの終了は、プログラムの END TRANSACTION ステートメントによって決まります。 このステートメントは、トランザクション内のすべての更新が正常に適用されたことを保証し、トランザクション中に 「ホールド」 状態におかれていたすべてのレコードを解放します。

例：

```
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
END-DEFINE
FIND MYVIEW WITH NAME = 'SMITH'
   DELETE
   END TRANSACTION
```

```
END-FIND
END
```

選択された各レコードは 「ホールド」 状態に置かれ、削除され、その後は END TRANSACTION ステートメントが実行されるときに 「ホールド」 状態から解放されます。

Note: Natural 管理者が設定する Natural プロファイルパラメータ ETEOP は、各 Natural プログラムの終了時に Natural で END TRANSACTION ステートメントを生成するかどうかを決定します。 詳細については、Natural 管理者に確認してください。

STORE ステートメントの例

次のプログラム例では、EMPLOYEES ファイルに新しいレコードを追加します。

```
** Example 'STOREX01': STORE  (Add new records to EMPLOYEES file)
*
** CAUTION: Executing this example will modify the database records!
************************************************************************
DEFINE DATA LOCAL
1 EMPLOYEE-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID(A8)
  2 NAME        (A20)
  2 FIRST-NAME  (A20)
  2 MIDDLE-I    (A1)
  2 SALARY      (P9/2)
  2 MAR-STAT    (A1)
  2 BIRTH       (D)
  2 CITY        (A20)
  2 COUNTRY     (A3)
*
1 #PERSONNEL-ID (A8)
1 #NAME         (A20)
1 #FIRST-NAME   (A20)
1 #INITIAL      (A1)
1 #MAR-STAT     (A1)
1 #SALARY       (N9)
1 #BIRTH        (A8)
1 #CITY         (A20)
1 #COUNTRY      (A3)
1 #CONF         (A1)   INIT <'Y'>
END-DEFINE
*
REPEAT
  INPUT 'ENTER A PERSONNEL ID AND NAME (OR ''END'' TO END)' //
        'PERSONNEL-ID : ' #PERSONNEL-ID    //
        'NAME         : ' #NAME  /
        'FIRST-NAME   : ' #FIRST-NAME
  /*********************************************************************
  /*  validate entered data
```

```
/***********************************************************************
IF #PERSONNEL-ID = 'END' OR #NAME = 'END'
  STOP
END-IF
IF #NAME = ' '
  REINPUT WITH TEXT 'ENTER A LAST-NAME'
          MARK 2 AND SOUND ALARM
END-IF
IF #FIRST-NAME = ' '
  REINPUT WITH TEXT 'ENTER A FIRST-NAME'
          MARK 3 AND SOUND ALARM
END-IF
/***********************************************************************
/*  ensure person is not already on file
/***********************************************************************
FIP2. FIND NUMBER EMPLOYEE-VIEW WITH PERSONNEL-ID =  #PERSONNEL-ID
/*
IF *NUMBER (FIP2.) > 0
  REINPUT 'PERSON WITH SAME PERSONNEL-ID ALREADY EXISTS'
           MARK 1 AND SOUND ALARM
END-IF
/***********************************************************************
/*  get further information
/***********************************************************************
INPUT
  'ENTER EMPLOYEE DATA'                              ////
  'PERSONNEL-ID             :' #PERSONNEL-ID (AD=IO) /
  'NAME                     :' #NAME         (AD=IO) /
  'FIRST-NAME               :' #FIRST-NAME   (AD=IO) ///
  'INITIAL                  :' #INITIAL              /
  'ANNUAL SALARY            :' #SALARY               /
  'MARITAL STATUS           :' #MAR-STAT             /
  'DATE OF BIRTH (YYYYMMDD) :' #BIRTH                /
  'CITY                     :' #CITY                 /
  'COUNTRY (3 CHARS)        :' #COUNTRY              //
  'ADD THIS RECORD (Y/N)    :' #CONF         (AD=M)
/***********************************************************************
/*  ENSURE REQUIRED FIELDS CONTAIN VALID DATA
/***********************************************************************
IF #SALARY < 10000
  REINPUT TEXT 'ENTER A PROPER ANNUAL SALARY' MARK 2
END-IF
IF NOT (#MAR-STAT = 'S'  OR = 'M' OR = 'D' OR = 'W')
  REINPUT TEXT 'ENTER VALID MARITAL STATUS  S=SINGLE ' -
               'M=MARRIED D=DIVORCED W=WIDOWED' MARK 3
END-IF
IF NOT(#BIRTH = MASK(YYYYMMDD) AND #BIRTH = MASK(1582-2699))
  REINPUT TEXT 'ENTER CORRECT DATE' MARK 4
END-IF
IF #CITY  = ' '
  REINPUT TEXT 'ENTER A CITY NAME' MARK 5
END-IF
```

```
  IF #COUNTRY = ' '
    REINPUT TEXT 'ENTER A COUNTRY CODE' MARK 6
  END-IF
  IF NOT (#CONF = 'N' OR= 'Y')
    REINPUT TEXT 'ENTER Y (YES) OR N (NO)' MARK 7
  END-IF
  IF #CONF = 'N'
    ESCAPE TOP
  END-IF
  /*********************************************************************
  /*  add the record with STORE
  /*********************************************************************
  MOVE #PERSONNEL-ID TO EMPLOYEE-VIEW.PERSONNEL-ID
  MOVE #NAME         TO EMPLOYEE-VIEW.NAME
  MOVE #FIRST-NAME   TO EMPLOYEE-VIEW.FIRST-NAME
  MOVE #INITIAL      TO EMPLOYEE-VIEW.MIDDLE-I
  MOVE #SALARY       TO EMPLOYEE-VIEW.SALARY (1)
  MOVE #MAR-STAT     TO EMPLOYEE-VIEW.MAR-STAT
  MOVE EDITED #BIRTH TO EMPLOYEE-VIEW.BIRTH (EM=YYYYMMDD)
  MOVE #CITY         TO EMPLOYEE-VIEW.CITY
  MOVE #COUNTRY      TO EMPLOYEE-VIEW.COUNTRY
  /*
  STP3. STORE RECORD IN FILE EMPLOYEE-VIEW
  /*
  /*********************************************************************
  /*  mark end of logical transaction
  /*********************************************************************
  END OF TRANSACTION
  RESET INITIAL #CONF
END-REPEAT
END
```

プログラム STOREX01 の出力：

```

ENTER A PERSONNEL ID AND NAME (OR 'END' TO END)

PERSONNEL ID :

```

```
NAME         :
FIRST NAME   :
```

レコードホールドロジック

Natural を Adabas で使用する場合、更新するレコードは、END TRANSACTION または BACKOUT TRANSACTION ステートメントが発行されるまで、あるいはトランザクションタイムリミットを超えるまで、「ホールド」 状態におかれます。

1人のユーザーに対してレコードが「ホールド」状態におかれると、他のユーザーはそのレコードを更新できません。 同じレコードを更新しようとする他のユーザーは、最初のユーザーがそのトランザクションを終了またはバックアウトしてレコードが 「ホールド」 から解放されるまで 「待ち」 状態におかれます。

ユーザーが待ち状態におかれることを防ぐために、セッションパラメータ WH（ホールドレコードの待機）を使用できます。『パラメータリファレンス』を参照してください。

プログラムで更新ロジックを使用するときには、以下の点を考慮する必要があります。

- レコードをホールド状態における最大時間は、Adabas のトランザクションタイムリミット（Adabas パラメータ TT）によって決まります。このタイムリミットを超えると、エラーメッセージが表示され、最後の END TRANSACTION 以降に行われたすべてのデータベース更新が取り消されます。
- ホールドされるレコード件数およびトランザクションタイムリミットは、トランザクションのサイズ、つまり、プログラムにおける END TRANSACTION ステートメントの配置に影響されます。 どこで END TRANSACTION を発行するかを決めるときには、再スタート機能を考慮する必要があります。例えば、処理中のレコードの大多数が更新*されない*場合は、レコードの「ホールド」状態の制御には GET ステートメントが効率的です。これにより、複数の END TRANSACTION ステートメントの発行が回避され、ホールドされる ISN の数が少なくなります。 大きなファイルを処理するときは、GET ステートメントでは追加の Adabas コールが必要となることに注意してください。 GET ステートメントの例を次に示します。

ホールドロジックの例

```
** Example 'GETX01': GET (put single record in hold with UPDATE stmt)
**
** CAUTION: Executing this example will modify the database records!
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 SALARY (1)
END-DEFINE
*
RD. READ EMPLOY-VIEW BY NAME
  DISPLAY EMPLOY-VIEW
  IF SALARY (1) > 1500000
```

```
    /*
    GE. GET EMPLOY-VIEW *ISN (RD.)
    /*
    WRITE '=' (50) 'RECORD IN HOLD:' *ISN(RD.)
    COMPUTE SALARY (1) = SALARY (1) * 1.15
    UPDATE (GE.)
    END TRANSACTION
  END-IF
END-READ
END
```

トランザクションのバックアウト

アクティブな論理トランザクション中、つまり、END TRANSACTION ステートメントが発行される前に、BACKOUT TRANSACTION ステートメントを使用してトランザクションをキャンセルすることができます。 このステートメントを実行すると、これまでに適用されたすべての更新（追加または削除されたすべてのレコードを含む）が削除され、トランザクションによってホールドされていたすべてのレコードが解放されます。

トランザクションの再スタート

END TRANSACTION ステートメントを使用して、トランザクション関連情報を保存することもできます。 トランザクション処理が異常終了する場合は、GET TRANSACTION DATA ステートメントでこの情報を読み取り、トランザクションを再スタートするときにどこで処理を再開するかを確認できます。

トランザクションデータを使用したトランザクション再スタートの例

次のプログラムは、EMPLOYEES ファイルと VEHICLES ファイルを更新します。 再スタート処理の後で、正常に処理された最後の EMPLOYEES レコードがユーザーに通知されます。 ユーザーは、その EMPLOYEES レコードから処理を再開できます。 再スタート処理の前に正常に更新された最後の VEHICLES レコードを、トランザクションの再スタートメッセージに含めるように設定することもできます。

```
** Example 'GETTRX01': GET TRANSACTION
*
** CAUTION: Executing this example will modify the database records!
************************************************************************
DEFINE DATA LOCAL
01 PERSON VIEW OF EMPLOYEES
  02 PERSONNEL-ID      (A8)
  02 NAME              (A20)
  02 FIRST-NAME        (A20)
  02 MIDDLE-I          (A1)
  02 CITY              (A20)
01 AUTO VIEW OF VEHICLES
  02 PERSONNEL-ID      (A8)
```

```
  02 MAKE              (A20)
  02 MODEL             (A20)
*
01 ET-DATA
  02 #APPL-ID          (A8) INIT <' '>
  02 #USER-ID          (A8)
  02 #PROGRAM          (A8)
  02 #DATE             (A10)
  02 #TIME             (A8)
  02 #PERSONNEL-NUMBER (A8)
END-DEFINE
*
GET TRANSACTION DATA #APPL-ID #USER-ID #PROGRAM
                     #DATE    #TIME    #PERSONNEL-NUMBER
*
IF  #APPL-ID NOT = 'NORMAL'     /* if last execution ended abnormally
AND #APPL-ID NOT = ' '
  INPUT (AD=OIL)
    //  20T '*** LAST SUCCESSFUL TRANSACTION ***' (I)
    /   20T '***********************************'
    /// 25T      'APPLICATION:' #APPL-ID
    /   32T             'USER:' #USER-ID
    /   29T          'PROGRAM:' #PROGRAM
    /   24T     'COMPLETED ON:' #DATE 'AT' #TIME
    /   20T 'PERSONNEL NUMBER:' #PERSONNEL-NUMBER
END-IF
*
REPEAT
  /*
  INPUT (AD=MIL) // 20T 'ENTER PERSONNEL NUMBER:' #PERSONNEL-NUMBER
  /*
  IF #PERSONNEL-NUMBER = '99999999'
    ESCAPE BOTTOM
  END-IF
  /*
  FIND1. FIND PERSON WITH PERSONNEL-ID = #PERSONNEL-NUMBER
    IF NO RECORDS FOUND
      REINPUT 'SPECIFIED NUMBER DOES NOT EXIST; ENTER ANOTHER ONE.'
    END-NOREC
    FIND2. FIND AUTO WITH PERSONNEL-ID = #PERSONNEL-NUMBER
      IF NO RECORDS FOUND
        WRITE 'PERSON DOES NOT OWN ANY CARS'
        ESCAPE BOTTOM
      END-NOREC
      IF *COUNTER (FIND2.) = 1     /* first pass through the loop
        INPUT (AD=M)
          /   20T 'EMPLOYEES/AUTOMOBILE DETAILS' (I)
          /   20T '----------------------------'
          /// 20T 'NUMBER:' PERSONNEL-ID (AD=O)
          /   22T   'NAME:' NAME ' ' FIRST-NAME ' ' MIDDLE-I
          /   22T   'CITY:' CITY
          /   22T   'MAKE:' MAKE
```

```
          /   21T  'MODEL:' MODEL
        UPDATE (FIND1.)            /* update the EMPLOYEES file
      ELSE                         /* subsequent passes through the loop
        INPUT NO ERASE (AD=M IP=OFF) //////// 28T MAKE / 28T MODEL
      END-IF
      /*
      UPDATE (FIND2.)              /* update the VEHICLES file
      /*
      MOVE *APPLIC-ID TO #APPL-ID
      MOVE *INIT-USER TO #USER-ID
      MOVE *PROGRAM   TO #PROGRAM
      MOVE *DAT4E     TO #DATE
      MOVE *TIME      TO #TIME
      /*
      END TRANSACTION #APPL-ID #USER-ID #PROGRAM
                      #DATE    #TIME    #PERSONNEL-NUMBER
      /*
    END-FIND                       /* for VEHICLES    (FIND2.)
  END-FIND                         /* for EMPLOYEES   (FIND1.)
END-REPEAT                         /* for REPEAT
*
STOP                               /* Simulate abnormal transaction end
END TRANSACTION 'NORMAL  '
END
```

ACCEPT/REJECT を使用したレコードの選択

このsectionでは、ユーザー指定の論理条件に基づいてレコードを選択するために使用する ACCEPT ステートメントおよび REJECT ステートメントについて説明します。

次のトピックについて説明します。

- ACCEPT および REJECT とともに使用可能なステートメント
- ACCEPT ステートメントの例
- ACCEPT/REJECT ステートメントの論理条件基準
- AND 演算子を指定した ACCEPT ステートメントの例
- OR 演算子を指定した REJECT ステートメントの例

ACCEPT および REJECT とともに使用可能なステートメント

ACCEPT ステートメントおよび REJECT ステートメントは、次のデータベースアクセスステートメントと連携して使用できます。

- READ
- FIND
- HISTOGRAM

ACCEPT ステートメントの例

```
** Example 'ACCEPX01': ACCEPT IF
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
  2 CURR-CODE (1:1)
  2 SALARY    (1:1)
END-DEFINE
*
READ (20) MYVIEW BY NAME WHERE CURR-CODE (1) = 'USD'
ACCEPT IF SALARY (1) >= 40000
  DISPLAY NAME JOB-TITLE SALARY (1)
END-READ
END
```

出力

```
  Page      1                                                 04-11-11  11:11:11

         NAME                  CURRENT            ANNUAL
                              POSITION            SALARY
  -------------------- ------------------------- ----------

  ADKINSON             DBA                          46700
  ADKINSON             MANAGER                      47000
  ADKINSON             MANAGER                      47000
  AFANASSIEV           DBA                          42800
  ALEXANDER            DIRECTOR                     48000
  ANDERSON             MANAGER                      50000
```

```
  ATHERTON            ANALYST                        43000
  ATHERTON            MANAGER                        40000
```

ACCEPT/REJECT ステートメントの論理条件基準

ACCEPT ステートメントおよび REJECT ステートメントを使用すると、READ ステートメントの WITH 節および WHERE 節で指定した論理条件に加えて、論理条件を指定できます。

ACCEPT/REJECT ステートメントの IF 節の論理条件基準は、レコードが選択されて読み込まれた*後*で評価されます。

論理条件演算子には次のものがあります。詳細については、「*論理条件基準*」を参照してください。

EQUAL	EQ	:=
NOT EQUAL TO	NE	¬=
LESS THAN	LT	<
LESS EQUAL	LE	<=
GREATER THAN	GT	>
GREATER EQUAL	GE	>=

ACCEPT/REJECT ステートメントの論理条件基準は、ブール演算子 AND、OR、および NOT で結合することもできます。 さらに、論理グループを示すためにカッコを使用することもできます。次の例を参照してください。

AND 演算子を指定した ACCEPT ステートメントの例

次のプログラムは、ACCEPT ステートメントでのブール演算子 AND の使用を示しています。

```
** Example 'ACCEPX02': ACCEPT IF ... AND ...
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
  2 CURR-CODE (1:1)
  2 SALARY    (1:1)
END-DEFINE
*
READ (20) MYVIEW BY NAME WHERE CURR-CODE (1) = 'USD'
ACCEPT IF  SALARY (1) >= 40000
         AND SALARY (1) <= 45000
  DISPLAY NAME JOB-TITLE SALARY (1)
```

```
END-READ
END
```

プログラム ACCEPX02 の出力：

```
Page      1                                                  04-12-14  12:22:01

        NAME                  CURRENT            ANNUAL
                             POSITION            SALARY
-------------------- ------------------------ ----------

AFANASSIEV           DBA                           42800
ATHERTON             ANALYST                       43000
ATHERTON             MANAGER                       40000
```

OR 演算子を指定した REJECT ステートメントの例

次のプログラムは、REJECT ステートメントでブール演算子 OR を使用したものです。論理演算子が逆になっているため、前の例の ACCEPT ステートメントと同じ出力が生成されます。

```
** Example 'ACCEPX03': REJECT IF ... OR ...
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
  2 CURR-CODE (1:1)
  2 SALARY    (1:1)
END-DEFINE
*
READ (20) MYVIEW BY NAME WHERE CURR-CODE (1) = 'USD'
REJECT IF SALARY (1) < 40000
         OR SALARY (1) > 45000
  DISPLAY NAME JOB-TITLE SALARY (1)
END-READ
END
```

プログラム ACCEPX03 の出力：

```
Page      1                                                  04-12-14  12:26:27

        NAME                  CURRENT            ANNUAL
                             POSITION            SALARY
-------------------- ------------------------ ----------

AFANASSIEV           DBA                           42800
```

```
ATHERTON            ANALYST                         43000
ATHERTON            MANAGER                         40000
```

ACCEPT/REJECT ステートメントのその他の例

次の例のプログラムを参照してください。

- ***ACCEPX04 - ACCEPT IF ... LESS THAN ...***
- ***ACCEPX05 - ACCEPT IF ... AND ...***
- ***ACCEPX06 - REJECT IF ... OR ...***

AT START/END OF DATA ステートメント

このsectionでは、AT START OF DATA ステートメントおよび AT END OF DATA ステートメントの使用について説明します。

次のトピックについて説明します。

- AT START OF DATA ステートメント
- AT END OF DATA ステートメント
- AT START OF DATA/AT END OF DATA ステートメントの例
- AT START OF DATA/AT END OF DATA のその他の例

AT START OF DATA ステートメント

AT START OF DATA ステートメントは、データベース処理ループで一連のレコードの最初のレコードが読み込まれた後に実行する処理を指定するために使用します。

AT START OF DATA ステートメントは、処理ループ内に指定する必要があります。

AT START OF DATA 処理で出力が生成される場合は、*最初のフィールド値の前*に出力されます。デフォルトでは、この出力は左揃えでページに表示されます。

AT END OF DATA ステートメント

AT END OF DATA ステートメントは、データベース処理ループのすべてのレコードが処理された後に実行する処理を指定するために使用します。

AT END OF DATA ステートメントは、処理ループ内に指定する必要があります。

AT END OF DATA 処理で出力が生成される場合は、*最後のフィールド値の後*に出力されます。 デフォルトでは、この出力は左揃えでページに表示されます。

AT START OF DATA/AT END OF DATA ステートメントの例

次のプログラム例は、AT START OF DATA ステートメントおよび AT END OF DATA ステートメントの使用を示しています。

時刻を表示するために、Natural のシステム変数 *TIME が AT START OF DATA ステートメントに組み込まれています。

最後に選択された従業員の名前を表示するために、Natural のシステム関数 OLD が AT END OF DATA ステートメントに組み込まれています。

```
** Example 'ATSTAX01': AT START OF DATA
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 JOB-TITLE
  2 INCOME  (1:1)
    3 CURR-CODE
    3 SALARY
    3 BONUS (1:1)
END-DEFINE
*
WRITE TITLE 'XYZ EMPLOYEE ANNUAL SALARY AND BONUS REPORT' /
READ (3) MYVIEW BY CITY STARTING FROM 'E'
  DISPLAY GIVE SYSTEM FUNCTIONS
          NAME (AL=15) JOB-TITLE (AL=15) INCOME (1)
  /*
 AT START OF DATA
    WRITE 'RUN TIME:' *TIME /
  END-START
  AT END OF DATA
    WRITE / 'LAST PERSON SELECTED:' OLD (NAME) /
  END-ENDDATA
END-READ
*
AT END OF PAGE
  WRITE / 'AVERAGE SALARY:' AVER (SALARY(1))
END-ENDPAGE
END
```

プログラムが次の出力を生成します。

```
                  XYZ EMPLOYEE ANNUAL SALARY AND BONUS REPORT

     NAME            CURRENT                  INCOME
                     POSITION
                                  CURRENCY   ANNUAL     BONUS
                                    CODE     SALARY
--------------- --------------- -------- ---------- ----------

RUN TIME: 12:43:19.1

DUYVERMAN       PROGRAMMER      USD           34000          0
PRATT           SALES PERSON    USD           38000       9000
MARKUSH         TRAINEE         USD           22000          0

LAST PERSON SELECTED: MARKUSH


AVERAGE SALARY:      31333
```

AT START OF DATA/AT END OF DATA のその他の例

次の例のプログラムを参照してください。

- *ATENDX01 - AT END OF DATA*
- *ATSTAX02 - AT START OF DATA*
- *WRITEX09 - WRITE（AT END OF DATA との組み合わせ）*

Unicode データ

Natural を使用すると、Adabas データベースのワイド文字フィールド（フォーマット W）にアクセスできます。

次のトピックについて説明します。

- データ定義モジュール
- アクセスコンフィグレーション

- 制限事項

データ定義モジュール

Adabas のワイド文字フィールド（W）は、Natural フォーマット U（Unicode）にマッピングされます。

フォーマット U の Natural フィールドの長さの定義は、フォーマット W の Adabas フィールドのサイズの半分に一致します。例えば、長さ 200 の Adabas のワイド文字フィールドは、Natural の "(U100)" にマッピングされます。

アクセスコンフィグレーション

Natural は Adabas からデータを受け取り、共通のエンコードとして UTF-16 を使用してデータを Adabas に送ります。

このエンコードは `OPRB` パラメータで指定され、オープン要求で Adabas に送られます。これはワイド文字フィールドに使用され、Adabas ユーザーセッション全体を通して適用されます。

制限事項

可変長のワイド文字フィールド（W）はサポートされていません。

照合ディスクリプタはサポートされていません。

Adabas および Unicode のサポートの詳細については、該当する Adabas 製品のドキュメントを参照してください。

30 SQL データベースのデータへのアクセス

このchapterでは、Natural で Entire Access を経由して SQL データベースを使用する方法について説明します。 インストールおよび設定の詳細については、『*データベース管理システムインターフェイス*』ドキュメントの「*Natural と Entire Access*」、および別冊の Entire Access ドキュメントを参照してください。

このchapterでは、次のトピックについて説明します。

Note: 原則として、『*Adabas データベースのデータへのアクセス*』ドキュメントに記載されている機能とその例は、Natural がサポートする SQL データベースにも該当します。 相違点がある場合は、個別のデータベースアクセスステートメントに関するドキュメントの「*データベース固有の考慮事項*」（『*ステートメント*』ドキュメントを参照）または各 Natural パラメータに関するドキュメント（『*パラメータリファレンス*』を参照）に詳細が記載されています。 また、Natural では SQL データベースにアクセスするための特別な一連のステートメントが提供されます。

Natural DDMs の生成

Entire Access は、Natural SQL ステートメントおよびほとんどの Natural DML ステートメントをサポートするアプリケーションプログラミングインターフェイス（API）です。

Natural DML ステートメントおよび Natural SQL ステートメントは、同じ Natural プログラムで使用することができます。 コンパイル時に、*NATCONF.CFG* 内に定義されているデータソースの DDM が DBMS タイプ "SQL" で DML ステートメントによって参照されると、Natural はこの DML ステートメントを SQL ステートメントに変換します。

Natural は DML ステートメントおよび SQL ステートメントを Entire Access へのコールに変換します。 Entire Access はこれらの要求を、ターゲット RDBMS で必要とされるデータフォーマットと SQL 方言に変換し、データベースドライバに渡します。

Natural プロファイルパラメータの設定

ETEOP パラメータ

このパラメータは Natural 管理者のみが設定可能です。

Natural プロファイルパラメータ `ETEOP` は、Natural セッション中のトランザクション処理を制御します。 例えば、1 つの論理トランザクションを 2 つ以上の Natural プログラムにわたって処理する場合などに必要になります。 この場合、Natural プログラムの終了時に、Natural で `END TRANSACTION` コマンドを発行しない、つまり 「コミット」 しないでください。

ETEOP パラメータを設定すると、次のように処理されます。

ON	Natural セッションが ET 状態でない場合は、Natural プログラムの終了時に Natural から END TRANSACTION ステートメントが発行されます。つまり、自動的に 「コミット」 されます。
OFF	Natural プログラムの終了時に、Natural から END TRANSACTION コマンドは発行されません（つまり、「コミット」 されません）。したがって、このように設定すると、単一の論理トランザクションを複数の Natural プログラムにわたって処理することができます。 これはデフォルトです。

Note: ETEOP パラメータは、Natural バージョン 6.1 以降に適用されます。 Natural の以前のバージョンでは、Natural プロファイルパラメータ OPRB を ETEOP の代わりに使用する必要があります。ETEOP=ON が OPRB=OFF に、ETEOP=OFF が ORPB=NOOPEN に相当します。

Natural DML ステートメント

次の表は、Natural が行う DML ステートメントから SQL ステートメントへの変換を示しています。

DML ステートメント	SQL ステートメント
BACKOUT TRANSACTION	ROLLBACK
DELETE	DELETE WHERE CURRENT OF *cursor-name*
END TRANSACTION	COMMIT
EQUAL ... OR	IN (...)
EQUAL ... THRU ...	BETWEEN ... AND ...
FIND ALL	SELECT
FIND NUMBER	SELECT COUNT (*)
HISTOGRAM	SELECT COUNT (*)
READ LOGICAL	SELECT ... ORDER BY
READ PHYSICAL	SELECT ... ORDER BY
SORTED BY ... [DESCENDING]	ORDER BY ... [DESCENDING]
STORE	INSERT
UPDATE	UPDATE WHERE CURRENT OF *cursor-name*
WITH	WHERE

Note: ブール演算子および関係演算子は、DML ステートメントと SQL ステートメントで同様に機能します。

Entire Access では、次に示す DML ステートメントおよびオプションはサポートされません。

- CIPHER
- COUPLED
- FIND FIRST, FIND UNIQUE, FIND ... RETAIN AS
- GET, GET SAME, GET TRANSACTION DATA, GET RECORD
- PASSWORD
- READ BY ISN
- STORE USING/GIVING NUMBER

BACKOUT TRANSACTION

Natural は、BACKOUT TRANSACTION ステートメントを SQL ROLLBACK コマンドに変換します。 このステートメントは、最後のリカバリユニットが完了した後に行われたすべてのデータベース更新を元に戻します。 リカバリユニットはセッションの最初、あるいは最後の END TRANSACTION（COMMIT）ステートメントまたは BACKOUT TRANSACTION（ROLLBACK）ステートメントの後に開始できます。

Note: 作業の論理ユニットが終了すると、すべてのカーソルが閉じるため、BACKOUT TRANSACTION ステートメントをデータベースループの内部に配置しないでください。ループの外側、またはネストされたループの一番外側のループの後に配置してください。

DELETE

DELETE ステートメントは、先行する FIND、READ、または SELECT の各ステートメントによって読み込まれた行をデータベーステーブルから削除します。 SQL ステートメントの DELETE WHERE CURRENT OF *cursor-name* に対応します。つまり、最後に読み込まれた行だけを削除できます。

例：

```
FIND EMPLOYEES WITH NAME = 'SMITH'
     AND FIRST_NAME = 'ROGER'
DELETE
```

Natural は、上述した Natural ステートメントを次の SQL ステートメントに変換し、カーソル名（例えば、"CURSOR1"）を割り当てます。 SELECT ステートメントと DELETE ステートメントは同じカーソルを参照します。

```
SELECT FROM EMPLOYEES
 WHERE NAME = 'SMITH' AND FIRST_NAME = 'ROGER'
DELETE FROM EMPLOYEES
 WHERE CURRENT OF CURSOR1
```

Natural は、FIND ステートメントを SQL SELECT ステートメントに変換する方法で、DELETE ステートメントを SQL DELETE ステートメントに変換します。 詳細については、後述の FIND ステートメントの説明を参照してください。

Note: FIND SORTED BY ステートメントまたは READ LOGICAL ステートメントで読み込まれた行を削除することはできません。 詳細については、後述の FIND ステートメントおよび READ ステートメントの説明を参照してください。

END TRANSACTION

Natural は、END TRANSACTION ステートメントを SQL COMMIT コマンドに変換します。 END TRANSACTION ステートメントは、論理トランザクションの終わりを示し、データベースに対してすべての更新をコミットし、トランザクション中にロックされていたデータを解放します。

Notes:

1. 作業の論理ユニットが終了すると、すべてのカーソルが閉じるため、END TRANSACTION ステートメントをデータベースループの内部に配置しないでください。ループの外側、またはネストされたループの一番外側のループの後に配置してください。
2. Entire Access とともに使用するときは、END TRANSACTION ステートメントでトランザクション（ET）データを保存することはできません。
3. Entire Access では、Natural プログラムの終了時に COMMIT は自動的に発行されません。

FIND

Natural は、FIND ステートメントを SQL SELECT ステートメントに変換します。 SELECT ステートメントは OPEN CURSOR コマンドによって実行され、その後に FETCH コマンドが続きます。 FETCH コマンドは、すべてのレコードが読み込まれるか、またはプログラムが FIND 処理ループを終了するまで繰り返し実行されます。 CLOSE CURSOR コマンドは、SELECT 処理を終了します。

例：

Natural ステートメント：

```
FIND EMPLOYEES WITH NAME = 'BLACKMORE'
   AND AGE EQ 20 THRU 40
OBTAIN PERSONNEL_ID NAME AGE
```

同等の SQL ステートメント：

```
SELECT PERSONNEL_ID, NAME, AGE
  FROM EMPLOYEES
    WHERE NAME = 'BLACKMORE'
      AND AGE BETWEEN 20 AND 40
```

検索条件の作成には、ディスクリプタとして指定されているテーブルの任意の列（フィールド）を使用できます。

Natural は FIND ステートメントの WITH 節を SQL SELECT ステートメントの WHERE 節に変換します。 Natural は、WITH 節を使用して行が選択された後で、FIND ステートメントの WHERE 節を評価します。 ビューフィールドは、ディスクリプタとして指定されている場合にのみ、WITH 節で使用できます。

Natural は、FIND NUMBER ステートメントを COUNT(*) 節が含まれる SQL SELECT ステートメントに変換します。 特定の検索条件についてレコードが存在するかどうかを決定する場合は、IF NO RECORDS FOUND 節よりも FIND NUMBER ステートメントの方がパフォーマンスが優れています。

Note: SORTED BY 節が含まれる FIND ステートメントで読み込まれた行を更新または削除することはできません。 Natural は、FIND ステートメントの SORTED BY 節を SQL SELECT ステートメントの ORDER BY 節に変換します。これにより、読み取り専用の結果テーブルが作成されます。

HISTOGRAM

Natural は、HISTOGRAM ステートメントを SQL SELECT ステートメントに変換します。 HISTOGRAM ステートメントによって、テーブル内の行のうち特定の列に同じ値を持つ行の数が返されます。 行の数は、Natural システム変数 *NUMBER で返されます。

例：

Natural ステートメント：

```
HISTOGRAM EMPLOYEES FOR AGE
OBTAIN AGE
```

同等の SQL ステートメント：

```
SELECT AGE, COUNT(*) FROM EMPLOYEES
  GROUP BY AGE
  ORDER BY AGE
```

READ

Natural は、READ ステートメントを SQL SELECT ステートメントに変換します。 READ PHYSICAL ステートメントと READ LOGICAL ステートメントの両方を使用できます。

READ LOGICAL ステートメントで読み込まれた行（例 1）を更新または削除することはできません。 Natural は、READ LOGICAL ステートメントを SQL SELECT ステートメントの ORDER BY 節に変換します。これにより、読み取り専用の結果テーブルが作成されます。

READ PHYSICAL ステートメント（例 2）は更新または削除できます。 Natural はこのステートメントを ORDER BY 節のない SELECT ステートメントに変換します。

例 **1**：

Natural ステートメント：

```
READ PERSONNEL BY NAME
OBTAIN NAME FIRSTNAME DATEOFBIRTH
```

同等の SQL ステートメント：

```
SELECT NAME, FIRSTNAME, DATEOFBIRTH FROM PERSONNEL
  WHERE NAME >= ' '
    ORDER BY NAME
```

例 **2**：

Natural ステートメント：

```
READ PERSONNEL PHYSICAL
OBTAIN NAME
```

同等の SQL ステートメント：

```
SELECT NAME FROM PERSONNEL
```

READ ステートメントに WHERE 節が含まれる場合、Natural は検索条件に従って行が選択された後で WHERE 節を評価します。

STORE

STORE ステートメントはデータベーステーブルに行を追加します。 SQL INSERT ステートメントに対応します。

例：

Natural ステートメント：

```
STORE RECORD IN EMPLOYEES
  WITH PERSONNEL_ID = '2112'
       NAME         = 'LIFESON'
       FIRST_NAME   = 'ALEX'
```

同等の SQL ステートメント：

```
INSERT INTO EMPLOYEES (PERSONNEL_ID, NAME, FIRST_NAME)
  VALUES ('2112', 'LIFESON', 'ALEX')
```

UPDATE

DML UPDATE ステートメントは、先行する FIND、READ、または SELECT の各ステートメントによって読み込まれたテーブルの行を更新します。 Natural は、DML UPDATE ステートメントを SQL UPDATE WHERE CURRENT OF *cursor-name* ステートメント（位置決め UPDATE ステートメント）に変換します。つまり、読み込まれた最後の行のみを更新できます。 ネストされたループでは、ネストされた各ループの最後の行を更新できます。

UPDATE の FIND/READ との使用

DML UPDATE ステートメントを Natural FIND ステートメントの後で使用すると、Natural は FIND ステートメントを FOR UPDATE OF 節のある SQL SELECT ステートメントに変換し、DML UPDATE ステートメントを UPDATE WHERE CURRENT OF *cursor-name* ステートメントに変換します。

例：

```
FIND EMPLOYEES WITH SALARY < 5000
  ASSIGN SALARY = 6000
  UPDATE
```

Natural は、上述した Natural ステートメントを次の SQL ステートメントに変換し、カーソル名（例えば、"CURSOR1"）を割り当てます。 SELECT ステートメントと UPDATE ステートメントは同じカーソルを参照します。

```
SELECT SALARY FROM EMPLOYEES WHERE SALARY < 5000
  FOR UPDATE OF SALARY
UPDATE EMPLOYEES SET SALARY = 6000
  WHERE CURRENT OF CURSOR1
```

FIND SORTED BY ステートメントまたは READ LOGICAL ステートメントで読み込まれた行を更新することはできません。 詳細については、上述の FIND ステートメントおよび READ ステートメントの説明を参照してください。

END TRANSACTION ステートメントまたは BACKOUT TRANSACTION ステートメントは、UPDATE ステートメントによってロックされたデータを解放します。

UPDATE の SELECT との使用

DML UPDATE ステートメントを SELECT ステートメントの後で使用できるのは、次の形のみです。

```
SELECT *
  INTO VIEW view-name
```

Natural は、上記以外の形での SELECT ステートメントと DML UPDATE ステートメントの使用をすべて拒否します。 Natural は、DML UPDATE ステートメントを非カーソルまたは「検索済み」SQL UPDATE ステートメントに変換します。つまり、Natural ビュー全体のみが更新可能であり、個々の列は更新できません。

また、DML UPDATE ステートメントを SELECT ステートメントの後で使用できるのは Natural ストラクチャードモードの場合のみです。構文は次のようになります。

```
UPDATE [RECORD] [IN] [STATEMENT] [(r)]
```

例：

```
DEFINE DATA LOCAL
01 PERS VIEW OF SQL-PERSONNEL
  02 NAME
  02 AGE
END-DEFINE
SELECT *
  INTO VIEW PERS
  FROM SQL-PERSONNEL
  WHERE NAME LIKE 'S%'
  OBTAIN NAME
    IF NAME = 'SMITH'
      ADD 1 TO AGE
    UPDATE
    END-IF
END-SELECT
```

他の点については、DML UPDATE ステートメントが SELECT ステートメントとともに機能する方法は、Natural FIND ステートメントとともに機能する場合と同じです。上述の「*UPDATE の FIND/READ との使用*」を参照してください。

Natural SQL ステートメント

Natural プログラミング言語内で使用できる SQL ステートメントには、一般セットと拡張セットという 2 つの異なるステートメントのセットがあります。 このプラットフォームでは、拡張セットのみが Natural でサポートされています。

一般セットは、Natural でサポートされる SQL 対応の各データベースシステムで操作できます。基本的には標準の SQL 構文定義に対応しています。 Natural SQL ステートメントの一般セットの詳細については、『ステートメント』ドキュメントの「*一般セットと拡張セット*」を参照してください。

このセクションでは、Natural SQL ステートメントの一般セットを Entire Access で使用するときの考慮事項と制限事項について説明します。

- DELETE
- INSERT
- PROCESS SQL
- SELECT

- UPDATE

DELETE

Natural SQL DELETE ステートメントは、カーソルを使わずにテーブルの行を削除します。

Natural は DML DELETE ステートメントを位置決め DELETE ステートメント、つまり SQL DELETE WHERE CURRENT OF *cursor-name* ステートメントに変換しますが、Natural SQL DELETE ステートメントは非カーソルまたは検索済みの DELETE ステートメントです。 検索済み DELETE ステートメントとは、どの SELECT ステートメントとも無関係のスタンドアロンステートメントです。

INSERT

INSERT ステートメントはテーブルに行を追加します。Natural STORE ステートメントに対応します。

PROCESS SQL

PROCESS SQL ステートメントは、*ddm-name* で特定されるデータベースに対して SQL ステートメントを *statement-string* の形で発行します。

Note: PROCESS SQL ステートメントを使用してデータベースループを実行することはできません。

パラメータ

Natural では INDICATOR 節と LINDICATOR 節がサポートされます。 代わりに、*statement-string* にパラメータを含めることもできます。 構文項目 *parameter* は、構文的には次のように定義されます。

```
[ :U ]
[ :G ]  :host-variable
```

host-variable は、SQL ステートメント内で参照される Natural プログラム変数です。

SET SQLOPTION*option*=*value*

Entire Access では、SET SQLOPTION *option*=*value* を *statement-string* のように指定することもできます。 この方法は、SQL データベースにアクセスするためのさまざまなオプションの指定に使用できます。オプションは、PROCESS SQL ステートメントによって参照されるデータベースにのみ適用されます。

サポートされるオプションは、以下のとおりです。

- DATEFORMAT
- DBPROCESS（Sybase のみ）
- TIMEOUT（Sybase のみ）
- TRANSACTION（Sybase のみ）

DATEFORMAT

このオプションは、SQL DATE および DATETIME 情報をタイプ A の Natural フィールドに取り込むために使用するフォーマットを指定します。 タイプ D または T の Natural フィールドが使用されている場合、このオプションは使用されなくなりました。 次に示す Natural の日付および時刻の編集マスクのサブセットを使用できます。

YYYY	年（4 桁）
YY	年（2 桁）
MM	月
DD	日
HH	時
II	分
SS	秒

日付フォーマットに空白が含まれる場合は、フォーマットをアポストロフィで囲む必要があります。

例：

ISO 日付フォーマットを使用する場合は、次のように指定します。

```
PROCESS SQL sql-ddm << SET SQLOPTION DATEFORMAT = YYYY-MM-DD >>
```

日付および時刻のコンポーネントをISOフォーマットで取得するには、次のように指定します。

```
PROCESS SQL sql-ddm << SET SQLOPTION DATEFORMAT = 'YYYY-MM-DD HH:II:SS'    >>
```

DATEFORMAT は、データベースからデータが取得される場合にのみ評価されます。 データがデータベースに渡される場合は、データベースシステムによって変換されます。 したがって、DATEFORMAT で指定されるフォーマットは、基礎となるデータベースで有効な日付フォーマットである必要があります。

Natural フィールドに DATEFORMAT が指定されていない場合は、次のように処理されます。

- デフォルトの日付フォーマットである DD-MON-YY が使用されます。"MON" は月の英語名の 3 文字の略記です。
- 次に示すデフォルトの日付／時刻フォーマットが使用されます。

Adabas D	YYYYMMDDHHIISS
DB2	YYYY-MM-DD-HH.II.SS
INFORMIX	YYYY-MM-DD HH:II:SS
ODBC	YYYY-MM-DD HH:II:SS
ORACLE	YYYYMMDDHHIISS
SYBASE DBLIB	YYYYMMDD HH:II:SS
SYBASE CTLIB	YYYYMMDD HH:II:SS
Microsoft SQL Server	YYYYMMDD HH:II:SS
その他	DD-MON-YY

DBPROCESS

このオプションは、Sybase および Microsoft SQL Server データベースにのみ有効です。

このオプションは、Sybase および Microsoft SQL Server の DBPROCESS に対する SQL ステートメントの割り当てに影響を与えるために使用します。 DBPROCESS は、Sybase および Microsoft SQL Server DBlib インターフェイスで提供されないデータベースカーソルをエミュレートするために、Entire Access で使用されます。

2 つの値を指定可能です。

MULTIPLE	DBPROCESS を MULTIPLE に設定すると、各 SELECT ステートメントではそれぞれ独自のセカンダリ DBPROCESS が使用されますが、その他すべての SQL ステートメントはプライマリ DBPROCESS 内で実行されます。 したがって、値 MULTIPLE を指定すると、データベースループが開いている場合でも、アプリケーションでさらに SQL ステートメントを実行できます。 ネストされたデータベースループも可能です。
SINGLE	DBPROCESS を SINGLE に設定すると、すべての SQL ステートメントが同じ DBPROCESS、つまりプライマリ DBPROCESS を使用します。 したがって、データベースループがアクティブである間は、新しいデータベースステートメントを実行することができません。1つの DBPROCESS は一度に 1 つの SQL バッチしか実行できないためです。 ただし、すべてのステートメントが同じ（プライマリ）DBPROCESS で実行されるため、非共有一時テーブルからの SELECT 処理が可能になります。

Notes:

1. 指定値は、アクティブなデータベースループがない場合にのみ変更可能です。
2. DBPROCESS オプションは Sybase および Microsoft SQL Server DBlib インターフェイスにしか適用されないため、アプリケーションでは一元的な CALLNAT ステートメントを使用して少なくとも SINGLE の値を変更し、Sybase クライアントライブラリがサポートされるようになった時点でこれらのコールを簡単に削除できるようにしておく必要があります。 デフォルト設定である MULTIPLE を確立する一元的なエラー処理もアプリケーションで使用する必要があります。

TIMEOUT

このオプションは、Sybase および Microsoft SQL Server データベースにのみ有効です。

Sybase および Microsoft SQL Server とともに使用する場合、Entire Access はタイムアウトの手法を使ってデータベースアクセスのデッドロックを検出します。 デフォルトのタイムアウト時間は 8 秒です。 このオプションでは、タイムアウト時間を秒単位で変更できます。

例えば、タイムアウト時間を 30 秒に設定するには、次のように指定します。

```
PROCESS SQL sql-ddm << SET SQLOPTION TIMEOUT = 30 >>
```

TRANSACTION

このオプションは、Sybase および Microsoft SQL Server データベースにのみ有効です。

このオプションは、トランザクションモードを有効または無効にするために使用します。 次の END TRANSACTION ステートメントまたは BACKOUT TRANSACTION ステートメントの後で有効になります。

トランザクションモードが有効（デフォルト）になると、Naturalは必要なすべてのステートメントを自動的に発行してトランザクションを開始します。

例：

トランザクションモードを無効にするには、次のように指定します。

```
PROCESS SQL sql-ddm << SET SQLOPTION TRANSACTION = NO >>
...
END TRANSACTION
```

トランザクションモードを有効にするには、次のように指定します。

```
PROCESS SQL sql-ddm << SET SQLOPTION TRANSACTION = YES >>
...
END TRANSACTION
```

SQLDISCONNECT

Entire Access では、SQLDISCONNECT を *statement-string* として指定することもできます。 このステートメントを次に示す SQLCONNECT ステートメントと組み合わせることによって、同じセッション内の1つのアプリケーションを使用して、または必要に応じて接続および切断するだけで、異なるデータベースにアクセスできます。

SQLDISCONNECT ステートメントが正常に実行されると、SQLCONNECT ステートメントによってそれまでに提供された情報がクリアされます。つまり、PROCESS SQL ステートメントで使用された DDM の DBID で決定された、現在接続中の SQL データベースからアプリケーションが切断されます。 接続が1つも確立されていない場合、SQLDISCONNECT ステートメントは無視されます。 トランザクションが開いている場合、このステートメントは失敗します。

Note: SQLDISCONNECT ステートメントのエラーが Natural によって報告された場合、接続状態は変更されません。 データベースによってエラーが報告された場合は、接続状態が定義されていません。

SQLCONNECT*option*=*value*

Entire Access では、SQLCONNECT *option=value* を *statement-string* として指定することもできます。 このステートメントは、PROCESS SQL ステートメントによってアドレス指定された DDM に指定された DBID に従って SQL データベースへの接続を確立するために使用できます。 指定した接続がすでに確立されている場合、SQLCONNECT ステートメントは失敗します。

サポートされるオプションは、以下のとおりです。

- USERID

- PASSWORD
- OS_PASSWORD
- OS_USERID
- DBMS_PARAMETER

Notes:

1. SQLCONNECT ステートメントが失敗した場合、接続状態は変更されません。
2. 複数のオプションを指定する場合は、コンマで区切る必要があります。
3. 指定する値には、リテラル文字またはフォーマット A の Natural 変数のいずれかを使用できます。
4. データベースへの接続が失われていたなどの理由で、Naturalが暗黙的な再接続を実行する場合は、SQLCONNECT ステートメントで指定された値が使用されます。

各オプションの評価は次のようになります。

USERID と PASSWORD

データベースログオンに USERID と PASSWORD を指定すると、デフォルトのログオンウィンドウの表示、および環境変数 SQL_DATABASE_USER と SQL_DATABASE_PASSWORD の評価は行われません。

USERID のみを指定した場合は、PASSWORD が空白であるとみなされます。この逆も同様です。

USERID も PASSWORD も指定しなかった場合は、デフォルトのログオン処理が適用されます。

Note: ユーザー ID やパスワードを必要としないデータベースシステムでは、ユーザー ID とパスワードを空白で指定して、デフォルトのログオン処理を抑制できます。

OS_USERID と OS_PASSWORD

オペレーティングシステムのログオンで OS_PASSWORD と OS_USERID を指定すると、ログオンウィンドウの表示、および環境変数 SQL_OS_USER と SQL_OS_PASSWORD の評価は行われません。

OS_USERID のみを指定した場合は、OS_PASSWORD が空白であるとみなされます。この逆も同様です。

OS_USERID も OS_PASSWORD も指定しなかった場合は、デフォルトのログオン処理が適用されます。

Note: ユーザー ID やパスワードを必要としないオペレーティングシステムでは、ユーザー ID とパスワードを空白で指定して、デフォルトのログオン処理を抑制できます。

DBMS_PARAMETER

DBMS_PARAMETER をダイナミックに指定すると、Natural グローバルコンフィグレーションファイル内の DBMS 割り当てが上書きされます。

例：

```
PROCESS SQL sql-ddm << SQLCONNECT USERID = 'DBA', PASSWORD = 'SECRET'    >>
```

この例では、Natural グローバルコンフィグレーションファイルに指定されたデータベースに、ユーザー ID "DBA" とパスワード "SECRET" を使って接続します。

```
DEFINE DATA LOCAL
1 #UID (A20)
1 #PWD (A20)
END-DEFINE
INPUT 'Please enter ADABAS D user ID and password' / #UID / #PWD
PROCESS SQL sql-ddm << SQLCONNECT USERID = : #UID,
                       PASSWORD          = : #PWD,
                       DBMS_PARAMETER    = 'ADABASD:mydb'
                    >>
```

この例では、Adabas D データベース "mydb" に、INPUT ステートメントから取得したユーザー ID とパスワードを使って接続します。

```
PROCESS SQL sql-ddm << SQLCONNECT USERID = ' ', PASSWORD = ' ',
                       DBMS_PARAMETER    = 'DB2:EXAMPLE' >>
```

この例では、DB2 データベース "EXAMPLE" に、ユーザー ID とパスワードを指定せずに接続します。オペレーティングシステムユーザー ID を使用する DB2 ではこれらは不要であるためです。

SELECT

SELECT ステートメントの INTO 節およびスカラ演算子は、RDBMS 固有で標準 SQL 構文に準拠しない（Natural 一般セット）か、または Entire Access で使用するときに制限があるかのいずれかです。

Entire Access では、INTO 節内の INDICATOR 節と LINDICATOR 節はサポートされません。 そのため、Entire Access では INTO 節に対して次の構文が必要になります。

```
INTO [ parameter, ...
       VIEW {view-name},... ]
```

Note: 連結演算子（||）は一般セットに属していないため、Entire Access ではサポートされません。

SELECT SINGLE

SELECT SINGLE ステートメントは非カーソル SELECT 処理（単独 SELECT）、つまり、カーソルを使用せずに最大で 1 行を検索する SELECT ステートメントの機能を提供します。

このステートメントは、Natural の FIND UNIQUE ステートメントと同じです。 ただし、Natural では返される行数が自動的にチェックされます。 複数行が選択された場合は、Natural によってエラーメッセージが返されます。

使用する RDBMS で非カーソル SELECT 処理のダイナミックな実行がサポートされない場合、Natural SELECT SINGLE ステートメントはセットレベルの SELECT のように実行され、その結果カーソル操作が発生します。 ただし、返される行数は引き続きチェックされ、複数行が選択された場合はエラーメッセージが発行されます。

UPDATE

Natural SQL UPDATE ステートメントは、カーソルを使わずにテーブルの行を更新します。

Natural は DML UPDATE ステートメントを位置決め UPDATE ステートメント、つまり SQL UPDATE WHERE CURRENT OF*cursor-name* ステートメントに変換しますが、Natural SQL UPDATE ステートメントは非カーソルまたは検索済みの UPDATE ステートメントです。 検索済み UPDATE ステートメントとは、どの SELECT ステートメントとも無関係のスタンドアロンステートメントです。

フレキシブル SQL

フレキシブル SQL を使用すると、任意の RDBMS 固有構文拡張を利用できます。 フレキシブル SQL は、次の SQL 構文項目と置き換えて使用できます。

- 原子
- 列参照
- スカラー式
- 条件

Natural コンパイラはフレキシブル SQL で使用される SQL テキストを認識しません。SQL ステートメント内で参照される Natural プログラム変数であるホスト変数の値を置き換えてから、RDBMS に渡す SQL 文字列に SQL テキストをコピーするだけです。 フレキシブル SQL テキストの構文エラーは、ランタイムに RDBMS がその文字列を実行するときに検出されます。

フレキシブル SQL の次のような特徴に注意してください。

- "<<" 文字と ">>" 文字に囲まれ、任意の SQL テキストとホスト変数を含めることができます。
- ホスト変数にはコロン（:）を接頭辞として付ける必要があります。
- SQL 文字列は複数のステートメント行に渡って指定でき、コメントも入力できます。

フレキシブル SQL は、選択式の節間でも使用できます。

```
SELECT selection
  << ... >>
  INTO ...
  FROM ...
  << ... >>
  WHERE ...
  << ... >>
  GROUP BY ...
  << ... >>
  HAVING ...
  << ... >>
  ORDER BY ...
  << ... >>
```

例：

```
SELECT NAME
FROM EMPLOYEES
WHERE << MONTH (BIRTH) >> = << MONTH (CURRENT_DATE) >>

SELECT NAME
FROM EMPLOYEES
WHERE << MONTH (BIRTH) = MONTH (CURRENT_DATE) >>

SELECT NAME
FROM EMPLOYEES
WHERE SALARY > 50000
<< INTERSECT
   SELECT NAME
   FROM EMPLOYEES
```

```
   WHERE DEPT = 'DEPT10'
>>
```

RDBMS 固有の要件および制限事項

このセクションでは、Entire Access で使用する Natural および一部の RDBMS に対する制限事項と特別な要件について説明します。

次のトピックについて説明します。

- 大文字／小文字を区別するデータベースシステム
- SYBASE および Microsoft SQL Server

大文字／小文字を区別するデータベースシステム

Naturalプログラムで指定されたすべての名前が自動的に小文字に変換されるため、大文字／小文字を区別するデータベースシステムではテーブル名や列名に小文字を使用してください。

Note: この制限は、フレキシブル SQL を使用するときには適用されません。

SYBASE および Microsoft SQL Server

SQL ステートメントを SYBASE や Microsoft SQL Server に対して実行するには、1 つ以上の DBPROCESS 構造を使用する必要があります。 DBPROCESS は SQL コマンドバッチを実行できます。

コマンドバッチとは特定の順番に並べられた一連の SQL ステートメントです。 各ステートメントは、コマンドバッチに定義されている順番で実行される必要があります。 SELECT ステートメントなど、結果を返すステートメントがある場合は、そのステートメントを先に実行し、その後で行を1行ずつフェッチする必要があります。コマンドバッチの後に次のステートメントをいったん実行すると、前のクエリから行をフェッチできなくなります。

SYBASE や Microsoft SQL Server では、アプリケーションで複数の DBPROCESS 構造を使用できるため、クエリごとに別個の DBPROCESS を使用する場合は、クエリをネストすることができます。 ただし、SYBASE や Microsoft SQL Server では各 DBPROCESS についてデータをロックするため、複数の DBPROCESS を使用するアプリケーション自体でデッドロックが発生する可能性があります。 Natural はデッドロックが発生するとタイムアウトします。

以下では次のトピックについて説明します。

- Natural ステートメントをデータベースコールに変換する方法

Natural ステートメントをデータベースコールに変換する方法

Natural ではオープンな各クエリごとに DBPROCESS を 1 つずつ使用し、その他すべての SQL ステートメント（UPDATE、DELETE、INSERT など）に別の DBPROCESS を 1 つ使用します。

位置決め UPDATE/DELETE ステートメントにクエリが参照される場合、Natural では生成される SELECT ステートメントに 自動的に FOR BROWSE 節が付加され、行の読み込み中の UPDATE 処理を可能にします。

位置決め UPDATE/DELETE ステートメントに対して、SYBASE の dbqual 機能を使用して次の検索条件が生成されます。

```
WHERE unique-index = value AND tsequal (timestamp,old-timestamp)
```

この検索条件は、クエリから現在の行を再選択するために使用できます。 tsequal 機能は、別のユーザーによって行が更新されたかどうかをチェックします。

Natural で SYBASE および Microsoft SQL Server を使用する際の制限事項

Natural を SYBASE および Microsoft SQL Server とともに使用するときは、次の制限が適用されます。

大文字／小文字の区別
: SYBASE および Microsoft SQL Server では大文字／小文字が区別されますが、Natural ではパラメータは小文字で渡されます。 このため、SYBASE および Microsoft SQL Server のテーブルやフィールドが大文字のみまたは大文字が混在して定義されている場合は、データベース SYNONYM または Natural のフレキシブル SQL を使用する必要があります。

位置決め **UPDATE/DELETE** ステートメント
: 位置決め UPDATE/DELETE ステートメントをサポートするには、アクセス対象のテーブルに一意のインデックスとタイムスタンプ列が必要です。 また、このタイプスタンプ列がクエリの選択リストに含まれてはなりません。

行のクエリ
: SYBASE および Microsoft SQL Server ではページがロックされ、ロックされたページは DBPROCESS 構造の所有になります。

  アクティブな DBPROCESS にロックされたページは、その後 END TRANSACTION ステートメントまたは BACKOUT TRANSACTION ステートメントでロックが解放されるまで、ロックした DBPROCESS でもそれ以外でも読み込むことはできません。

  したがって、テーブルの行を更新、挿入、または削除した場合は次の点に注意してください。

- 同じテーブルに対して新しい SELECT、FIND、READ などのループを開始しないでください。
- SELECT ステートメントに FOR BROWSE 節がない場合は、同じテーブルを参照するクエリから追加の行をフェッチしないでください。

Natural は、位置決め UPDATE/DELETE ステートメントによってクエリが参照される場合に、FOR BROWSE 節を自動的に付加します。

トランザクション／非トランザクションモード
SYBASE および Microsoft SQL Server では、トランザクションモードと非トランザクションモードが区別されます。 トランザクションモードでは、Natural がデータベースに接続して INSERT、UPDATE、および DELETE を発行可能にするため、CREATE TABLE など非トランザクションモードで実行されるコマンドは発行できません。

ストアドプロシージャ
PROCESS SQL ステートメントを使用すると、SYBASE および Microsoft SQL Server でストアドプロシージャを使用できます。ただし、ストアドプロシージャに以下のものを含めないでください。

- 非トランザクションモードでしか機能しないコマンド
- 戻り値

データタイプ変換

Natural プログラムがリレーショナルデータベース内のデータにアクセスすると、Entire Access は RDBMS 固有のデータタイプを Natural データフォーマットに変換します。この逆も同様です。 RDBMS データタイプとそれに対応する Natural データフォーマットについては、『エディタ』ドキュメントの「*DDM エディタ*」セクションの「*Adabas または RDBMS のデータ変換*」を参照してください。

特定のデータベースに固有の DATE/TIME または DATETIME のフォーマットは、Natural フォーマットの D と T に変換できます。次を参照してください。

日付／時刻の変換

RDBMS 固有フォーマットである DATE/TIME または DATETIME は、Natural フォーマットの D と T に変換できます。

この変換を使うには、最初に Natural DDM を編集して、日付と時刻のフィールドフォーマットを A（英数字）から D（日付）と T（時刻）に変更しておく必要があります。 SQLOPTION DATEFORMAT は、フォーマット D または T のフィールドには使用されなくなりました。

Note: Natural D（日付）／T（時刻）フォーマットに変換した日付フィールドまたは時刻フィールドを、Natural A（英数字）フォーマットに変換したフィールドと混在させないでください。

- 更新コマンドでは、Natural 日付／時刻フォーマットが、データベース依存表現である DATE/TIME/DATETIME に秒単位の精度レベルで変換されます。
- 検索コマンドでは、返されたデータベース依存文字表現が Natural 内部の日付／時刻フォーマットに変換されます。後述する変換テーブルを参照してください。Natural時刻の日付コンポーネントは無視されません。RDBMSの時刻フォーマットに日付コンポーネントが含まれていない場合は 0000-01-02（YYYY-MM-DD）に初期化されます。
- Natural 日付変数では、時刻部分が無視され、ゼロに初期化されます。
- Natural 時刻変数では、1/10 秒台が無視され、ゼロに初期化されます。

変換テーブル

Adabas D

RDBMS フォーマット	Natural 日付	Natural 時刻
DATE	YYYYMMDD	
TIME		00HHIISS

DB2

RDBMS フォーマット	Natural 日付	Natural 時刻
DATE	YYYY-MM-DD	
TIME		HH.II.SS

INFORMIX

RDBMS フォーマット	Natural 日付	Natural 時刻
DATETIME（年～日）	YYYY-MM-DD	
DATETIME（年～秒。これ以外のフォーマットはサポートされません）		YYYY-MM-DD-HH:II:SS*

ODBC

RDBMS フォーマット	Natural 日付	Natural 時刻
DATE	YYYY-MM-DD	
TIME		HH:II:SS

ORACLE

RDBMS フォーマット	Natural 日付	Natural 時刻
DATE（ORACLE セッションパラメータ NLS_DATE_FORMAT は YYYYMMDDHH24MISS に設定されます）	YYYYMMDD000000（ORACLE 時刻コンポーネントは、更新コマンドでは空値が設定され、検索コマンドでは無視されます）	YYYYMMDDHHIISS *

SYBASE

RDBMS フォーマット	Natural 日付	Natural 時刻
DATETIME	YYYYMMDD	YYYYMMDD HH:II:SS *

* 2 つの時刻値を比較するときは、日付コンポーネントの値が異なる可能性があることに注意してください。

Microsoft SQL Server

RDBMS フォーマット	Natural 日付	Natural 時刻
DATETIME	YYYYMMDD	YYYYMMDD HH:II:SS *

データベースエラーに関する診断情報の取得

データベースへのアクセス中にエラーが返された場合は、Natural 以外のプログラムである CMOSQERR を呼び出し、次の構文を使用して、エラーに関する診断情報を取得できます。

```
CALL 'CMOSQERR' parm1 parm2
```

パラメータは次のとおりです。

パラメータ	フォーマット／長さ	説明
parm1	I4	データベースから返されたエラーの番号
parm2	A70	データベースから返されたエラーのテキスト

SQL 認証

Natural コンフィグレーションユーティリティを使用すると、SQL データベースに自動ログインするためのユーザー ID とパスワードの DBID 固有の設定を追加できます。 現在のデータベースシステムに応じて、オペレーティングシステム認証とデータベース認証が区別されます。［**SQL 認証（SQL Authorization**）］テーブルの［自動ログイン（**Auto login**）］フラグが SQL DBID に設定されている場合、対話式ログインプロンプトは表示されません。 ログオン値はこのテーブル行から取得されます。

SQL 認証テーブルの詳細については、『*コンフィグレーションユーティリティ*』ドキュメントの「*SQL 割り当て*」を参照してください。

31 Tamino データベースのデータへのアクセス

このchapterでは、Natural のデータ操作言語（DML）を使用して Tamino データベースにアクセスするときのさまざまな面について説明します。

次のトピックについて説明します。

Tamino で作業するための Natural の設定方法の詳細については、『*データベース管理システムインターフェイス*』ドキュメントの「*Natural for Tamino*」を参照してください。

必要条件

Tamino では、構造化されたデータ指向の XML 文書が doctype と呼ばれるコンテナに保存されます。 doctypeは論理的なグループにまとめられ、このグループはコレクションと呼ばれます。コレクションは、データの物理的なコンテナである Tamino データベースに保存されます。

Tamino で保存でき、Natural for Tamino でアクセスできるデータは、Tamino XML スキーマで定義する必要があります。

Natural for Tamino で使用する DDM とビューの定義

このセクションでは、Tamino XML スキーマ言語、Natural DDMs、およびビューの定義の基本概念について、およびこれらと Natural for Tamino との対話方法について説明します。

次のトピックについて説明します。

- Tamino XML スキーマ言語について
- Tamino から生成した DDM
- Tamino から生成した DDM での配列
- DDM の例
- ビューの定義

Tamino XML スキーマ言語について

Tamino XML スキーマ言語は、XML 文書の構造を持つデータタイプ指向記述を定義するために使用されます。 Tamino では、1 つのコレクション内で同じルートエレメントと同じ構造を持つ XML 文書のコンテナが doctype によって表現されます。

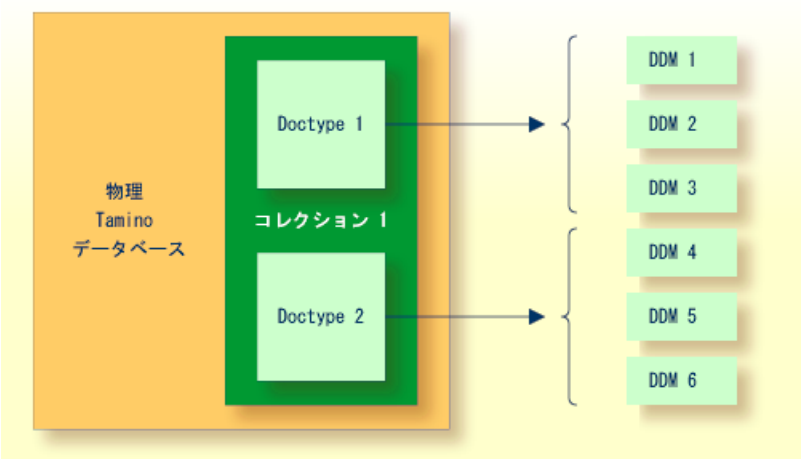
Tamino におけるコレクションとは、さまざまな doctype の集まりが入るコンテナであるため、同じものに属する doctype の論理的なグループとみなすことができます。

Tamino XML スキーマ定義では、doctype をそれが含まれるコレクションとともに定義します。1 つの Tamino XML スキーマで複数の doctype を定義でき、複数のコレクションに doctype を定義することもできます。

Tamino XML スキーマ言語の詳細については、Tamino のドキュメントを参照してください。

Tamino から生成した DDM

Natural から Tamino データベースにアクセスできるようにするには、Tamino の doctype と Natural のデータベース構造との間に論理的な接続を確立する必要があります。 このような論理接続は、データ定義モジュール（DDM）と呼ばれます。



Tamino データベースから生成される Natural DDM は、1 つのスキーマに定義された 1 つの doctype の表現です。 DDM には、対応する Tamino XML スキーマで定義されているように、各データフィールドのタイプに関する情報、および必要なすべての構造情報が含まれています。 新しい DDM を生成するには、指定されたコレクションで使用可能なすべての doctype のリストから doctype を選択する必要があります。 1 つのコレクションが 1 つの Natural データベース ID（DBID）に対応付けられるため、他のコレクションの doctype にアクセスするには 2 つ目の DBID を使用する必要があります。

Tamino XML スキーマでは、Natural のデータ定義とはかなり異なる方法でデータとデータ構造が記述されます。 このため、Natural データフォーマットを Tamino XML スキーマデータタイプから生成するために、専用のマッピングが導入されています。

DDM は Natural DDM エディタを使用して定義します。 Tamino XML スキーマのマッピングの詳細については、『エディタ』ドキュメントの「*DDM エディタ*」セクションの「*Tamino のデータ変換*」を参照してください。

DDM に定義されるフィールド属性については、『エディタ』ドキュメントの「*DDM エディタ*」セクションの「*DDM の編集 - フィールド属性定義*」を参照してください。

Tamino から生成した DDM での配列

Tamino XML スキーマに maxOccurs の値が 1 よりも大きい XML 要素を定義すると、この要素はその値が示す頻度で発生する可能性があります。このような指定は、Natural スタティック配列定義または Natural X-array 定義のいずれかにマッピングされます。 処理する XML 要素のタイプに応じて、次のような状況が起きる可能性があります。

- XML 要素が complexContent を持つ complexType である（要素に他の要素が含まれる）場合、生成される対応する Natural グループはインデックス付きグループになります。
- XML 要素が simpleType である（要素にデータのみが含まれる）場合、または simpleContent を持つ complexType である（要素にはデータと属性のみが含まれ、他の要素は含まれない）場合は、生成される Natural データフィールドは配列になります。

maxOccurs 定義を Natural 配列にマッピングする詳細については、『エディタ』ドキュメントの「*DDM エディタ*」セクションの「*Tamino のデータ変換*」を参照してください。 配列の境界または配列の種類（スタティック配列または X-array）は、通常と同様に、対応するビュー定義で変更できます。

DDM の例

次に示すのは、Tamino XML スキーマ定義から生成された EMPLOYEES DDM の例です。

このスキーマは、Natural デモアプリケーション SYSEXINS などを使用して定義できます。

```
DB: 00250 FILE: 00001  - EMPLOYEES-XML
TYPE: XML
COLLECTION: NATDemoData
SCHEMA: Employee
DOCTYPE: Employee
NAMESPACE-PREFIX: xs
NAMESPACE-URI: http://www.w3.org/2001/XMLSchema
T L  Name                            F Leng       D Remark
- -- ------------------------------- - ---------- - -----------
G  1 EMPLOYEE
     FLAGS=MULT_REQUIRED,MULT_ONCE
     TAG=Employee
     XPATH=/Employee
G  2 GROUP$1
     FLAGS=GROUP_ATTRIBUTES
   3 PERSONNEL-ID                    A          8 D xs:string
     FLAGS=ATTR_REQUIRED
     TAG=@Personnel-ID
     XPATH=/Employee/@Personnel-ID
G  2 GROUP$2
     FLAGS=GROUP_SEQUENCE,MULT_REQUIRED,MULT_ONCE
G  3 FULL-NAME
     FLAGS=MULT_OPTIONAL
     TAG=Full-Name
```

```
     XPATH=/Employee/Full-Name
G  4 GROUP$3
     FLAGS=GROUP_SEQUENCE,MULT_REQUIRED,MULT_ONCE
   5 FIRST-NAME                      A         20 D xs:string
     FLAGS=MULT_OPTIONAL
     TAG=First-Name
     XPATH=/Employee/Full-Name/First-Name
   5 MIDDLE-NAME                     A         20 D xs:string
     FLAGS=MULT_OPTIONAL
     TAG=Middle-Name
     XPATH=/Employee/Full-Name/Middle-Name
   5 MIDDLE-I                        A         20 D xs:string
     FLAGS=MULT_OPTIONAL
     TAG=Middle-I
     XPATH=/Employee/Full-Name/Middle-I
   5 NAME                            A         20 D xs:string
     FLAGS=MULT_OPTIONAL
     TAG=Name
     XPATH=/Employee/Full-Name/Name
       . . .
   3 LANG                            A          3   xs:string
     FLAGS=ARRAY,MULT_OPTIONAL
     OCC=1:4
     TAG=Lang
     XPATH=/Employee/Lang
```

ビューの定義

Natural プログラム内で Tamino データベースを使用して作業するには、必要な DDM のフィールドを Natural *view-definition* で指定する必要があります。`DEFINE DATA` ステートメントを参照してください。 一般に、ビューとは DDM で定義されているような完全なデータ構造の特殊なサブセットです。

Tamino XML スキーマ -> Natural for Tamino DDM -> Natural *view-definition*

ビューを XML オブジェクトの保存に使用する場合は、対応する Tamino XML スキーマ定義に従って、有効な文書を生成するために必要なすべてのフィールドがビューに含まれている必要があります。

ビューフィールドの1つがスタティック配列である EMPLOYEES-XML DDM のビューは、例えば次のようになります。

```
DEFINE DATA LOCAL
01 VW VIEW OF EMPLOYEES-XML
02 NAME
02 CITY
02 LANG (1:4)
END-DEFINE
```

Tamino データベースにアクセスするための Natural ステートメント

Tamino にアクセスするために提供されている Natural DML ステートメントは、さらに次の 2 つのカテゴリに分類できます。

- 純粋な検索ステートメント
- データベース更新ステートメント

Natural システム変数 `*ISN` は、Tamino `ino:id` 上でマッピングされます。

Natural for Tamino の検索ステートメント

次の Natural ステートメントをデータベース検索に使用できます。

- `FIND`

  このステートメントは、指定した検索条件に一致するレコードをデータベースから選択するために使用します。

- `GET`

  このステートメントは、一意の ID を使ってデータベースから 1 つの特別なレコードを選択するために使用します。

- `READ`

  このステートメントは、指定した順番でデータベースからレコードの範囲を選択するために使用します。

Tamino のアクセスでは、検索ステートメントの使用可能なオプションおよび節のすべてを使用できるわけではありません。 詳細については、『ステートメント』ドキュメントの該当するセクションを参照してください。

すべてのステートメントは、Tamino `_xquery` コマンド動詞で内部的に実現されます。 ステートメントの各節は、対応する Tamino XQuery 式にマッピングされます。例えば、検索条件は

Tamino XQuery 比較式にマッピングされ、順番の指定はソート方向を指定した Tamino XQuery 順番決定式にマッピングされます。

FIND ステートメントと READ ステートメントの結果セットはループの最初に決定され、ループ全体を通じて変更されません。

次の例では、1 つのビューフィールドが配列になっている Tamino データベースから一連の従業員レコードを読み取ります。

```
* READ 5 RECORDS DESCENDING CONTAINING A
* STATIC ARRAY IN THE VIEW DEFINE DATA LOCAL
01 VW VIEW OF EMPLOYEES-TAMINO
02 NAME
02 CITY
02 LANG (1:4)
END-DEFINE
*
READ(5) VW DESCENDING BY NAME = 'MAYER'
  DISPLAY NAME CITY LANG(*)
END-READ
*
END
```

Natural for Tamino のデータベース更新ステートメント

Natural for Tamino を使用するために、次のデータベース更新ステートメントが提供されます。

- STORE

  このステートメントは、新しい XML 文書をデータベースに挿入するために使用します。

- DELETE

  このステートメントは、データベースから文書を削除するために使用します。 DELETE ステートメントでは位置決め削除が実装されます。

各ステートメントの詳細については、『ステートメント』ドキュメントの該当するセクションを参照してください。

DELETE ステートメントは Tamino _delete コマンド動詞で現在の ino:id を使用して内部的に実現され、STORE ステートメントは _process コマンド動詞で実装されます。

例：

次のプログラム例では、何らかのデータを持つ新しい従業員レコードがデータベースに保存されます。

```
* STORE NEW EMPLOYEE
DEFINE DATA LOCAL
01 VW VIEW OF EMPLOYEES-TAMINO
02 PERSONNEL-ID
02 NAME
02 CITY
02 LANG (1:3)
END-DEFINE
*
* FILL VIEW
PERSONNEL-ID := '1230815'
NAME         := 'KENT'
CITY         := 'ROME'
LANG(1)      := 'ENG'
LANG(2)      := 'GER'
LANG(3)      := 'SPA'
*
* STORE VIEW
STORE RECORD IN VW
*
COMMIT
*
END
```

Tamino XML スキーマで doctype のデータ構造を必須と定義した場合は、STORE ステートメントが発行される前にこれらのデータ構造もビューに含める必要があります。含めないと Tamino エラーが発生します。

Natural for Tamino の論理トランザクション処理

Natural では、トランザクションに基づいてデータベース更新処理が実行されます。つまり、すべてのデータベース更新要求は論理トランザクション単位で処理されます。論理トランザクションは、データベースに含まれている情報が論理的に一貫性を持っていることを確実にするために、完全に実行されなければならない最小の業務ユニットです。業務ユニットの定義はユーザーが行います。

論理トランザクションは、データベース内の 1 つ以上の doctype に関係する 1 つ以上の更新ステートメント（DELETE または STORE）で構成される場合があります。また、論理トランザクションは、複数の Natural プログラムにまたがることもできます。

論理トランザクションは、データベース更新ステートメントが発行されたときに開始します。Natural ではこの処理が自動的に行われます。例えば、FIND ループに DELETE ステートメントが含まれている場合です。論理トランザクションの終了は、プログラムの END TRANSACTION ステートメントによって決まります。このステートメントによって、トランザクション内のすべての更新が正常に適用されたことが確実になります。

Natural for Tamino のエラー処理

Natural の標準エラーメッセージに加え、サブエラーコードを使って追加情報を提供する 2 つの特殊なエラーコードがあります。

NAT8400

Taminoエラー **...** が発生しました。

この特殊エラーでは、追加のサブコード番号が表示されます。 この番号は Tamino のエラーメッセージを表します。 Tamino の『*メッセージおよびコード*』ドキュメントを参照してください。 ライブラリ SYSEXT のユーザー出口 USR6007 は、NAT8400 エラーが発生した場合に診断情報を得るために提供されています。 使い方の例を次に示します。

```
DEFINE DATA LOCAL
 01 VW VIEW OF EMPLOYEES-TAMINO
  02 NAME
  02 CITY
 01 TAMINO_PARMS
  02 TAMINO_ERROR_NUM        (I4)   /* Error number of Tamino error
  02 TAMINO_ERROR_TEXT       (A70)  /* Tamino error text
  02 TAMINO_ERROR_LINE       (A253) /* Tamino error message line
END-DEFINE
*
NAME := 'MEYER'
CITY := 'BOSTON'
STORE VW
*
ON ERROR
 IF *ERROR EQ 8400     /* in case of error 8400 obtain diagnostic information

  CALLNAT 'USR6007N' TAMINO_PARMS
  PRINT 'Error 8400 occurred:'
  PRINT 'Error Number:' TAMINO_ERROR_NUM
  PRINT 'Error Text  :' TAMINO_ERROR_TEXT
  PRINT 'Error Line  :' TAMINO_ERROR_LINE
 END-IF
END-ERROR
```

```
*
END
```

NAT8411

HTTP 要求はレスポンスコード **...** で失敗しました。

HTTP サーバーからのエラーコードは追加情報として提供されます。 REQUEST DOCUMENT ステートメントの「*レスポンス番号の概要 - HTTP/HTTP 要求*」も参照してください。

Natural for Tamino と SQL データベースの対話の例

次に示すのは、Natural for Tamino と SQL データベースとのさらに高度な対話の例です。Tamino データベースからデータを検索し、SQL データベースの適切なテーブルの対応する行を挿入または更新します。

```
*
* TAMINO DB --> SQL RDBMS EXAMPLE
*
DEFINE DATA LOCAL
* DEFINE VIEW FOR TAMINO
01 VW-TAMINO VIEW OF EMPLOYEES-TAMINO
02 PERSONNEL-ID
02 NAME
02 CITY
* DEFINE VIEW FOR SQL DATABASE
01 VW-SQL VIEW OF EMPLOYEES-SQL
02 PERSONNEL_ID
02 NAME
02 CITY
END-DEFINE
*
* OPEN A TAMINO LOGICAL READ LOOP
*
TAMINO. READ VW-TAMINO BY NAME
*
* SEARCH RECORD IN SQL DATABASE AND
* INSERT A NEW RECORD IF NOT FOUND OR
* UPDATE THE EXISTING ONE WITH THE DATA
* FROM TAMINO DB
SQL. FIND(1) VW-SQL WITH PERSONNEL_ID = PERSONNEL-ID (TAMINO.)
       IF NO RECORDS FOUND
         PERSONNEL_ID := PERSONNEL-ID (TAMINO.)
         NAME         := NAME         (TAMINO.)
         CITY         := CITY         (TAMINO.)
         STORE VW-SQL
         ESCAPE BOTTOM
       END-NOREC
       PERSONNEL_ID := PERSONNEL-ID (TAMINO.)
```

```
       NAME          := NAME          (TAMINO.)
       CITY          := CITY          (TAMINO.)
       UPDATE
     END-FIND
*
END-READ
*
END TRANSACTION
*
END
```

Natural for Tamino の制限事項

Natural for Tamino DDM を生成するスキーマの作成に使用できる Tamino XML スキーマ言語の適用範囲に関して次のような制限があります。

- Natural for Tamino でサポートされるのは、Tamino XML スキーマ言語のコンストラクタと属性のみです。詳細については『*エディタ*』ドキュメントの「*DDM エディタ*」セクションの「*Tamino XML スキーマコンストラクタ*」を参照してください。 その他のコンストラクタ（`xs:any` や `xs:anyAttribute` など）は、Natural for Tamino とともに使おうとしても Tamino XML スキーマに適用できません。
- `xs:import` の機能は Natural for Tamino ではサポートされません。 つまり、外部スキーマコンポーネントを、Natural との使用に適した Tamino XML スキーマ内で参照することはできません。 すなわち、Tamino XML スキーマでの doctype 定義を Natural for Tamino とともに使用する場合は、この Tamino XML スキーマ自体の内部ですべての参照を解決する必要があります。
- コンストラクタ `xs:complexType` の属性 `mixed` は、デフォルト値である "false" でのみサポートされます。 Natural for Tamino では、内容が混在する文書定義（`mixed="true"` を指定して設定するなど）はサポートされません。 `mixed="true"` を使用すると、DDM の生成中にエラーが発生します。
- Natural for Tamino DDM でのネスト構造のレベルは 99 までに制限されています。 新しい DDM レベルは、次のいずれかのコンストラクタが Tamino XML スキーマで発生するたびに生成されます。

  `xs:element`
  `xs:attribute`
  `xs:choice`
  `xs:all`
  `xs:sequence`

- Tamino XML スキーマで再帰的に定義される構造を Natural for Tamino とともに使用することはできません。

- Tamino XML スキーマ言語コンストラクタ xs:choice は、すべての選択肢が含まれる Natural グループにマッピングされます。 特定の 1 つの選択に処理を制限するには、必要な選択を示した適切なビューを作成する必要があります。
- Natural for Tamino では、「クローズドコンテンツの整合性チェックモード」 のみがサポートされます。「オープンコンテンツの整合性モード」 のある Tamino XML スキーマは、Natural for Tamino とともに使用できません。
- Tamino XML スキーマ言語コンストラクタの xs:choice、xs:sequence、および xs:all について、属性 maxOccurs の値が 1 よりも大きくなると Natural データ構造では処理できません。したがって、1 よりも大きい値では DDM 生成中に常にエラーが発生します。
- Natural for Tamino では、W3C スキーマのサブセットとして Tamino XML スキーマで定義された Tamino オブジェクトのみを処理できます。 Natural for Tamino では、特に XML 以外のデータ（tsd:nonXML）やスキーマが定義されていないインスタンス（ino:etc）はサポートされません。

32 データ出力制御

This part describes how to proceed if a Natural program is to produce multiple reports. さらに、Naturalで作成した出力レポートのフォーマットを制御する方法、つまり、データの表示方法について、さまざまな面から説明します。

次のトピックについて説明します。

- レポート指定 - (*rep*) 表記
- 出力ページのレイアウト
- **DISPLAY** および **WRITE** ステートメント
- マルチプルバリューフィールドとピリオディックグループのインデックス表記
- ページタイトル、改ページ、空行
- 列ヘッダー
- フィールドの出力に影響を与えるパラメータ
- 編集マスク - **EM** パラメータ
- **Unicode** 編集マスク - **EMU** パラメータ
- 垂直表示

33 レポート指定 - (rep) 表記

(*rep*) は、ステートメントを適用できる出力レポートの識別子です。

このchapterでは、次のトピックについて説明します。

レポート指定の使用

Natural プログラムで複数のレポートを作成する場合、最初のレポート（レポート 0）以外のレポートを出力する出力ステートメント（下記の「*ステートメントに関する考慮事項*」を参照）ごとに、表記 (*rep*) を指定する必要があります。

0～31 の値を指定できます。

(*rep*) の値には、DEFINE PRINTER ステートメントを使用して割り当てた論理名も指定できます。下記の*例 2* を参照してください。

ステートメントに関する考慮事項

表記 (*rep*) は以下の出力ステートメントで使用できます。

AT END OF PAGE | AT TOP OF PAGE | DISPLAY | EJECT | FORMAT | NEWPAGE | PRINT | SKIP | SUSPEND IDENTICAL SUPPRESS | WRITE | WRITE TITLE | WRITE TRAILER

レポート指定の例

例 **1 -** 複数のレポート

```
DISPLAY (1) NAME ...
WRITE (4) NAME ...
```

例 **2 -** 論理名の使用

```
DEFINE PRINTER (LIST=5) OUTPUT 'LPT1'
WRITE (LIST) NAME ...
```

34 出力ページのレイアウト

このchapterでは、レポートの特定のレイアウト定義に使用できるステートメントの概要について説明します。

次のトピックについて説明します。

レポートレイアウトに影響するステートメント

レポートのレイアウトに影響するステートメントは、以下のとおりです。

ステートメント	機能
WRITE TITLE	ページタイトル、つまりページの先頭に出力するテキストを指定できます。 デフォルトでは、ページタイトルは中央揃えで下線なしです。
WRITE TRAILER	ページトレーラ、つまりページの下に出力するテキストを指定できます。デフォルトでは、トレーラ行は中央揃えで下線なしです。
AT TOP OF PAGE	レポートの新しいページが開始されるときに常に実行される処理を指定できます。 この処理による出力は、ページタイトルの下に出力されます。
AT END OF PAGE	ページ終了条件が発生したときに常に実行される処理を指定できます。この処理による出力は、`WRITE TRAILER` ステートメントで指定したページトレーラの下に出力されます。
AT START OF DATA	データベース処理ループで最初のレコードが読み込まれた後に実行される処理を指定します。この処理による出力は、最初のフィールド値の前に出力されます。
AT END OF DATA	処理ループのすべてのレコードが処理された後に実行される処理を指定します。この処理からの出力は、最後のフィールド値の直後に出力されます。
DISPLAY / WRITE	これらのステートメントで、読み込まれたフィールド値を出力する際のフォーマットを制御します。 「*DISPLAY および WRITE ステートメント*」を参照してください。

データ出力に対する `AT START OF DATA` および `AT END OF DATA` ステートメントに関連する説明は、データベースアクセスの「*AT START/END OF DATA ステートメント*」を参照してください。 上記リストの他のステートメントについては、「*データ出力制御*」の各セクションを参照してください。

一般的なレイアウトの例

以下のプログラム例は、出力ページの一般的なレイアウトを示しています。

```
** Example 'OUTPUX01': Several sections of output
************************************************************************
DEFINE DATA LOCAL
1 EMP-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 BIRTH
END-DEFINE
*
WRITE TITLE   '********** Page Title **********'
WRITE TRAILER '********** Page Trailer **********'
*
AT TOP OF PAGE
  WRITE '===== Top of Page ====='
END-TOPPAGE
AT END OF PAGE
  WRITE '===== End of Page ====='
END-ENDPAGE
*
READ (10) EMP-VIEW BY NAME
  /*
  DISPLAY NAME FIRST-NAME BIRTH (EM=YYYY-MM-DD)
  /*
  AT START OF DATA
    WRITE '>>>>> Start of Data >>>>>'
  END-START
  AT END OF DATA
    WRITE '<<<<< End of Data <<<<<'
  END-ENDDATA
END-READ
END
```

プログラム OUTPUX01 の出力：

```
                    ********** Page Title **********
===== Top of Page =====
        NAME              FIRST-NAME         DATE
                                              OF
                                            BIRTH
-------------------- -------------------- ----------

>>>>> Start of Data >>>>>
```

```
ABELLAN              KEPA                 1961-04-08
ACHIESON             ROBERT               1963-12-24
ADAM                 SIMONE               1952-01-30
ADKINSON             JEFF                 1951-06-15
ADKINSON             PHYLLIS              1956-09-17
ADKINSON             HAZEL                1954-03-19
ADKINSON             DAVID                1946-10-12
ADKINSON             CHARLIE              1950-03-02
ADKINSON             MARTHA               1970-01-01
ADKINSON             TIMMIE               1970-03-03
<<<<< End of Data <<<<<
                  ********** Page Trailer **********
===== End of Page =====
```

35 DISPLAY および WRITE ステートメント

このchapterでは、DISPLAY および WRITE ステートメントを使用してデータを出力する方法および情報を出力するフォーマットを制御する方法について説明します。

次のトピックについて説明します。

DISPLAY ステートメント

DISPLAY ステートメントでは列形式の出力が作成されます。つまり、1 つのフィールドの値が 1 つの列の下に縦に出力されます。 複数のフィールドが出力される場合、つまり複数の列が作成される場合、これらの列は互いに隣り合って横に出力されます。

表示されるフィールドの順番は、DISPLAY ステートメントに指定したフィールド名の順番で決定されます。

以下のプログラムの DISPLAY ステートメントでは、従業員ごとに、最初に従業員番号、次に名前、その次に職種が表示されます。

```
** Example 'DISPLX01': DISPLAY
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 BIRTH
  2 JOB-TITLE
END-DEFINE
*
READ (3) VIEWEMP BY BIRTH
  DISPLAY PERSONNEL-ID NAME JOB-TITLE
END-READ
END
```

プログラム DISPLX01 の出力：

```
Page      1                                                 04-11-11  14:15:54

PERSONNEL         NAME                  CURRENT
   ID                                   POSITION
--------- -------------------- -------------------------

30020013  GARRET               TYPIST
```

```
30016112  TAILOR               WAREHOUSEMAN
20017600  PIETSCH              SECRETARY
```

出力レポートに表示される列の順番を変更するには、DISPLAY ステートメントのフィールド名を単に並べ換えます。 例えば、従業員名、職種、従業員番号の順番でリストする場合、適切な DISPLAY ステートメントは、以下のようになります。

```
** Example 'DISPLX02': DISPLAY
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 BIRTH
  2 JOB-TITLE
END-DEFINE
*
READ (3) VIEWEMP BY BIRTH
  DISPLAY NAME JOB-TITLE PERSONNEL-ID
END-READ
END
```

プログラム DISPLX02 の出力：

```
Page      1                                              04-11-11  14:15:54

        NAME                  CURRENT           PERSONNEL
                             POSITION             ID
-------------------- ------------------------- ---------

GARRET               TYPIST                    30020013
TAILOR               WAREHOUSEMAN              30016112
PIETSCH              SECRETARY                 20017600
```

ヘッダーが各列の上に出力されます。このヘッダーに影響するさまざまな方法については、『*列ヘッダー*』ドキュメントを参照してください。

WRITE ステートメント

WRITE ステートメントは、フリーフォーマット（列なし）の出力を作成するために使用します。DISPLAY ステートメントとは対照的に、WRITE ステートメントには以下が適用されます。

- 必要に応じて、自動的に行送りを行います。つまり、現在の出力行に納まらないフィールドまたはテキスト要素は、自動的に次の行に出力されます。
- ヘッダーは作成されません。
- マルチプルバリューフィールドの値は、互いに隣り合って横に出力されます。縦には出力されません。

以下の2つのプログラム例は、DISPLAY ステートメントと WRITE ステートメントの基本的な違いを示しています。

『*垂直表示*』ドキュメントの「***DISPLAY と WRITE の組み合わせ***」で説明しているように、2つのステートメントを組み合わせて使用することもできます。

DISPLAY ステートメントの例

```
** Example 'DISPLX03': DISPLAY
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 SALARY (1:3)
END-DEFINE
*
READ (2) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY NAME FIRST-NAME SALARY (1:3)
END-READ
END
```

プログラム DISPLX03 の出力：

```
Page      1                                                  04-11-11  14:15:54

        NAME              FIRST-NAME        ANNUAL
                                            SALARY
-------------------- -------------------- ----------

JONES                VIRGINIA                  46000
                                               42300
                                               39300
JONES                MARSHA                    50000
                                               46000
                                               42700
```

WRITE ステートメントの例

```
** Example 'WRITEX01': WRITE
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 SALARY (1:3)
END-DEFINE
*
READ (2) VIEWEMP BY NAME STARTING FROM 'JONES'
  WRITE NAME FIRST-NAME SALARY (1:3)
END-READ
END
```

プログラム WRITEX01 の出力：

```
Page      1                                                  04-11-11  14:15:55
```

```
JONES                VIRGINIA                 46000       42300       39300
JONES                MARSHA                   50000       46000       42700
```

列の間隔 - SF パラメータと *n*X 表記

デフォルトでは、DISPLAY ステートメントで出力される列は互いに *1* 文字の空白で区切られます。

セッションパラメータ SF を使用すると、DISPLAY ステートメントで出力される列の間に挿入するデフォルトの空白文字数を指定できます。 空白文字数は 1～30 の任意の値に設定できます。

パラメータは、FORMAT ステートメント を使用してレポート全体に適用するか、または DISPLAY ステートメントを使用して要素レベルではなくステートメントレベルで指定することができます。

DISPLAY ステートメントの *n*X 表記で、2 つの列の間に挿入する空白文字数（*n*）を指定できます。 *n*X 表記は、SF パラメータによる指定を上書きします。

```
** Example 'DISPLX04': DISPLAY (with nX)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 BIRTH
  2 JOB-TITLE
END-DEFINE
*
FORMAT SF=3
READ (3) VIEWEMP BY BIRTH
  DISPLAY PERSONNEL-ID NAME 5X JOB-TITLE
END-READ
END
```

プログラム DISPLX04 の出力：

上記のプログラム例では、以下の出力が作成されます。最初の 2 つの列は FORMAT ステートメントの SF パラメータにより 3 文字の空白で区切られています。2 番目と 3 番目の列は、DISPLAY ステートメントの 5X 表記により 5 文字の空白で区切られています。

```
Page      1                                                  04-11-11  14:15:54

PERSONNEL           NAME                       CURRENT
   ID                                          POSITION
---------    --------------------     -------------------------

30020013    GARRET                   TYPIST
30016112    TAILOR                   WAREHOUSEMAN
20017600    PIETSCH                  SECRETARY
```

*n*X 表記は WRITE ステートメントでも使用可能であり、個々の出力要素間に空白を挿入します。

```
WRITE PERSONNEL-ID 5X NAME 3X JOB-TITLE
```

上記のステートメントでは、フィールド PERSONNEL-ID と NAME の間に 5 文字の空白が挿入され、NAME と JOB-TITLE の間には 3 文字の空白が挿入されます。

タブ設定 - *n*T 表記

*n*T 表記は DISPLAY および WRITE ステートメントで使用可能であり、この表記を使用して出力要素を出力する位置を指定できます。

```
** Example 'DISPLX05': DISPLAY (with nT)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
END-DEFINE
*
READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY 5T NAME 30T FIRST-NAME
END-READ
END
```

プログラム DISPLX05 の出力：

上記のプログラムでは、以下の出力が作成されます。フィールド NAME はページの左マージンから数えて 5 桁目から始まり、フィールド FIRST-NAME は 30 桁目から始まります。

```
Page      1                                                  04-11-11  14:15:54

             NAME                  FIRST-NAME
    -------------------     --------------------

    JONES                   VIRGINIA
    JONES                   MARSHA
    JONES                   ROBERT
```

行送り - スラッシュ表記

DISPLAY または WRITE ステートメントでスラッシュ（/）を使用すると、行送りを行うことができます。

- DISPLAY ステートメントでは、スラッシュにより*フィールド間*および*テキスト内*で行送りが行われます。
- WRITE ステートメントでは、スラッシュが*フィールド間*に指定された場合にのみ行送りが行われます。テキスト内に指定された場合は、通常のテキスト文字と同様に処理されます。

フィールド間に指定する場合は、スラッシュの両側に空白が必要です。

行送りを複数回行う場合は、複数のスラッシュを指定します。

例**1 - DISPLAY** ステートメントの行送り：

```
** Example 'DISPLX06': DISPLAY (with slash '/')
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 DEPARTMENT
END-DEFINE
*
READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY NAME / FIRST-NAME 'DEPART-/MENT' DEPARTMENT
END-READ
END
```

プログラム DISPLX06 の出力：

上記の DISPLAY ステートメントでは、フィールド NAME の各値の後、およびテキスト "DEPART-MENT" 内で行送りが行われます。

```
Page      1                                                  04-11-11  14:15:54

        NAME         DEPART-
     FIRST-NAME       MENT
-------------------- -------

JONES                SALE
VIRGINIA
JONES                MGMT
MARSHA
JONES                TECH
ROBERT
```

例**2 - WRITE** ステートメントの行送り：

```
** Example 'WRITEX02': WRITE (with line advance)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 DEPARTMENT
END-DEFINE
*
READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'
  WRITE NAME / FIRST-NAME 'DEPART-/MENT' DEPARTMENT //
END-READ
END
```

プログラム WRITEX02 の出力：

上記の WRITE ステートメントでは、フィールド NAME の各値の後に 1 行、フィールド DEPARTMENT の各値の後に 2 行行送りが行われますが、テキスト "DEPART-/MENT" 内では行送りは行われません。

```
Page      1                                                  04-11-11  14:15:55

JONES
VIRGINIA             DEPART-/MENT SALE


JONES
MARSHA               DEPART-/MENT MGMT
```

```
JONES
ROBERT               DEPART-/MENT TECH
```

例**3 - DISPLAY** および **WRITE** の行送り：

```
** Example 'DISPLX21': DISPLAY (usage of slash '/' in DISPLAY and WRITE)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 FIRST-NAME
  2 ADDRESS-LINE (1)
END-DEFINE
*
WRITE TITLE LEFT JUSTIFIED UNDERLINED
       *TIME
   5X  'PEOPLE LIVING IN SALT LAKE CITY'
   21X 'PAGE:' *PAGE-NUMBER /
   15X 'AS OF' *DAT4E //
*
WRITE TRAILER UNDERLINED 'REGISTER OF' / 'SALT LAKE CITY'
*
READ (2) EMPLOY-VIEW WITH CITY = 'SALT LAKE CITY'
  DISPLAY  NAME /
           FIRST-NAME
           'HOME/CITY' CITY
           'STREET/OR BOX NO.' ADDRESS-LINE (1)
  SKIP 1
END-READ
END
```

プログラム DISPLX21 の出力：

```
14:15:54.6     PEOPLE LIVING IN SALT LAKE CITY                    PAGE:      1
               AS OF 11/11/2004


------------------------------------------------------------------------------
        NAME                 HOME                 STREET
     FIRST-NAME              CITY               OR BOX NO.
-------------------- -------------------- --------------------

ANDERSON             SALT LAKE CITY       3701 S. GEORGE MASON
JENNY

SAMUELSON            SALT LAKE CITY       7610 W. 86TH STREET
MARTIN
```

```
                                    REGISTER OF
                                   SALT LAKE CITY
--------------------------------------------------------------------------------
```

DISPLAY および WRITE ステートメントの他の例

次の例のプログラムを参照してください。

- *DISPLX13 - DISPLAY（WRITE を使用している WRITEX08 と比較）*
- *WRITEX08 - WRITE（DISPLAY を使用している DISPLX13 と比較）*
- *DISPLX14 - DISPLAY（AL、SF、および nX を使用）*
- *WRITEX09 - WRITE（AT END OF DATA との組み合わせ）*

36 マルチプルバリューフィールドとピリオディックグループのインデックス表記

このchapterでは、インデックス表記 (*n*:*n*) を使用して、出力するマルチプルバリューフィールドの値の個数、または出力するピリオディックグループのオカレンス数を指定する方法について説明します。

次のトピックについて説明します。

インデックス表記の使用

インデックス表記 (*n*:*n*) を使用すると、出力するマルチプルバリューフィールドの値の個数またはピリオディックグループのオカレンス数を指定できます。

例えば、DDM EMPLOYEES のフィールド INCOME は、ある従業員が会社に雇用されてからの年ごとの年間収入記録を保持するピリオディックグループです。

これらの年間収入は年順に保持されています。 最新の年収は、オカレンス "1" に含まれています。

従業員の最近3年間の年間収入、つまりオカレンス "1"～"3" を表示する場合は、DISPLAY または WRITE ステートメントのフィールド名の後に (1:3) の表記を指定します。以下のプログラム例を参照してください。

DISPLAY ステートメントのインデックス表記の例

```
** Example 'DISPLX07': DISPLAY (with index notation)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 BIRTH
  2 INCOME  (1:3)
    3 CURR-CODE
    3 SALARY
    3 BONUS (1:1)
END-DEFINE
*
READ (3) VIEWEMP BY BIRTH
  DISPLAY PERSONNEL-ID NAME INCOME (1:3)
  SKIP 1
END-READ
END
```

プログラム DISPLX07 の出力：

DISPLAY ステートメントでは、マルチプルバリューフィールドの複数の値が縦に出力されることに注意してください。

```
Page      1                                                 04-11-11  14:15:54

PERSONNEL          NAME                      INCOME
   ID
                                  CURRENCY   ANNUAL     BONUS
                                    CODE     SALARY
--------- -------------------- -------- ---------- ----------

30020013  GARRET               UKL            4200          0
                               UKL            4150          0
                                                 0          0

30016112  TAILOR               UKL            7450          0
                               UKL            7350          0
                               UKL            6700          0

20017600  PIETSCH              USD           22000          0
                               USD           20200          0
                               USD           18700          0
```

WRITE ステートメントでは複数の値が縦ではなく横に表示されるため、これにより行あふれおよび不要な行送りが発生する場合があります。

ピリオディックグループ全体ではなくピリオディックグループ内の単一フィールド（SALARY など）のみを使用し、さらに以下の例の NAME と JOB-TITLE の間のようにスラッシュ（/）を挿入して行送りを行うと、扱いやすいレポートフォーマットになります。

WRITE ステートメントのインデックス表記の例

```
** Example 'WRITEX03': WRITE (with index notation)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 BIRTH
  2 JOB-TITLE
  2 SALARY (1:3)
END-DEFINE
*
READ (3) VIEWEMP BY BIRTH
  WRITE PERSONNEL-ID NAME / JOB-TITLE SALARY (1:3)
  SKIP 1
```

```
END-READ
END
```

プログラム WRITEX03 の出力：

```
Page      1                                                 04-11-11  14:15:55

30020013 GARRET
TYPIST                          4200       4150          0

30016112 TAILOR
WAREHOUSEMAN                    7450       7350       6700

20017600 PIETSCH
SECRETARY                      22000      20200      18700
```

37 ページタイトル、改ページ、空行

このchapterでは、レポートの改ページを制御する方法、各レポートページの先頭にページタイトルを出力する方法、および出力レポートに空白行を追加する方法について説明します。

次のトピックについて説明します。

デフォルトのページタイトル

DISPLAY または WRITE ステートメントによるページ出力ごとに、Natural によって自動的に単一のデフォルトタイトル行が生成されます。 このタイトル行には、ページ番号、日付、および時刻が含まれています。

例：

```
WRITE 'HELLO'
END
```

上記のプログラムでは、以下のようにデフォルトのページタイトルを含む出力が作成されます。

```
Page      1                                                  04-12-14  13:19:33

HELLO
```

ページタイトルの省略 - NOTITLE オプション

ページタイトルなしでレポートを出力する場合は、キーワード NOTITLE を DISPLAY または WRITE ステートメントに追加します。

例 **- NOTITLE** を指定した **DISPLAY**：

```
** Example 'DISPLX20': DISPLAY (with NOTITLE)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 FIRST-NAME
END-DEFINE
*
READ (5) EMPLOY-VIEW BY CITY FROM 'BOSTON'
  DISPLAY NOTITLE NAME FIRST-NAME CITY
END-READ
```

```
END
```

プログラム DISPLX20 の出力：

```
        NAME              FIRST-NAME              CITY
-------------------- -------------------- --------------------

SHAW                 LESLIE               BOSTON
STANWOOD             VERNON               BOSTON
CREMER               WALT                 BOSTON
PERREAULT            BRENDA               BOSTON
COHEN                JOHN                 BOSTON
```

例 **- NOTITLE** を指定した **WRITE**：

```
WRITE NOTITLE 'HELLO'
END
```

上記のプログラムでは、以下のようにページタイトルを含まない出力が作成されます。

```
HELLO
```

独自ページタイトル定義 - WRITE TITLE ステートメント

Naturalのデフォルトページタイトルの代わりに独自のページタイトルを出力する場合は、`WRITE TITLE` ステートメントを使用します。

以下では次のトピックについて説明します。

- タイトル用テキストの指定
- タイトルの後の空白行の指定
- タイトルの桁揃えと下線

- ページ番号付きのタイトル

タイトル用テキストの指定

WRITE TITLE ステートメントで、タイトル用のテキストをアポストロフィで囲んで指定します。

```
WRITE TITLE 'THIS IS MY PAGE TITLE'
WRITE 'HELLO'
END
```

上述したプログラムにより、次の出力が生成されます。

```
                            THIS IS MY PAGE TITLE
HELLO
```

タイトルの後の空白行の指定

WRITE TITLE ステートメントの SKIP オプションで、タイトル行のすぐ下に出力される空白行の数を指定できます。 キーワード SKIP の後に、挿入する空白行の数を指定します。

```
WRITE TITLE 'THIS IS MY PAGE TITLE' SKIP 2
WRITE 'HELLO'
END
```

上述したプログラムにより、次の出力が生成されます。

```
                            THIS IS MY PAGE TITLE


HELLO
```

SKIP は WRITE TITLE ステートメントの一部としてのみではなく、スタンドアロンステートメントとしても使用できます。

タイトルの桁揃えと下線

デフォルトでは、ページタイトルはページの中央に位置付けられ、下線は付きません。

WRITE TITLE ステートメントでは、互いに独立して使用可能な以下のオプションを指定できます。

オプション	効果
LEFT JUSTIFIED	ページタイトルを左揃えで表示します。
UNDERLINED	タイトルを下線付きで表示します。 行サイズの幅に下線が引かれます（Natural プロファイルおよびセッションパラメータ LS も参照）。デフォルトでは、タイトルにハイフン（-）で下線が引かれます。ただし、UC セッションパラメータを使用すると、下線用の文字として使用する他の文字を指定してすることができます（「*タイトルおよびヘッダーの下線用の文字*」を参照）。

以下の例は、LEFT JUSTIFIED および UNDERLINED オプションの効果を示しています。

```
WRITE TITLE LEFT JUSTIFIED UNDERLINED 'THIS IS MY PAGE TITLE'
SKIP 2
WRITE 'HELLO'
END
```

上述したプログラムにより、次の出力が生成されます。

```
THIS IS MY PAGE TITLE
------------------------------------------------------------------------------


HELLO
```

レポートの新しいページが開始されるときには、常に WRITE TITLE ステートメントが実行されます。

ページ番号付きのタイトル

以下の例では、システム変数 *PAGE-NUMBER を WRITE TITLE ステートメントとともに使用して、タイトル行にページ番号を出力します。

```
** Example 'WTITLX01': WRITE TITLE (with *PAGE-NUMBER)
************************************************************************
DEFINE DATA LOCAL
1 VEHIC-VIEW VIEW OF VEHICLES
  2 MAKE
  2 YEAR
  2 MAINT-COST (1)
END-DEFINE
*
LIMIT 5
*
READ VEHIC-VIEW
END-ALL
SORT BY YEAR USING MAKE MAINT-COST (1)
  DISPLAY NOTITLE YEAR MAKE MAINT-COST (1)
  AT BREAK OF YEAR
    MOVE 1 TO *PAGE-NUMBER
    NEWPAGE
  END-BREAK
  /*
  WRITE TITLE LEFT JUSTIFIED
        'YEAR:' YEAR 15X 'PAGE' *PAGE-NUMBER
END-SORT
END
```

プログラム WTITLX01 の出力：

```
YEAR:  1980               PAGE      1
YEAR          MAKE         MAINT-COST
----- -------------------- ----------

 1980 RENAULT                  20000
```

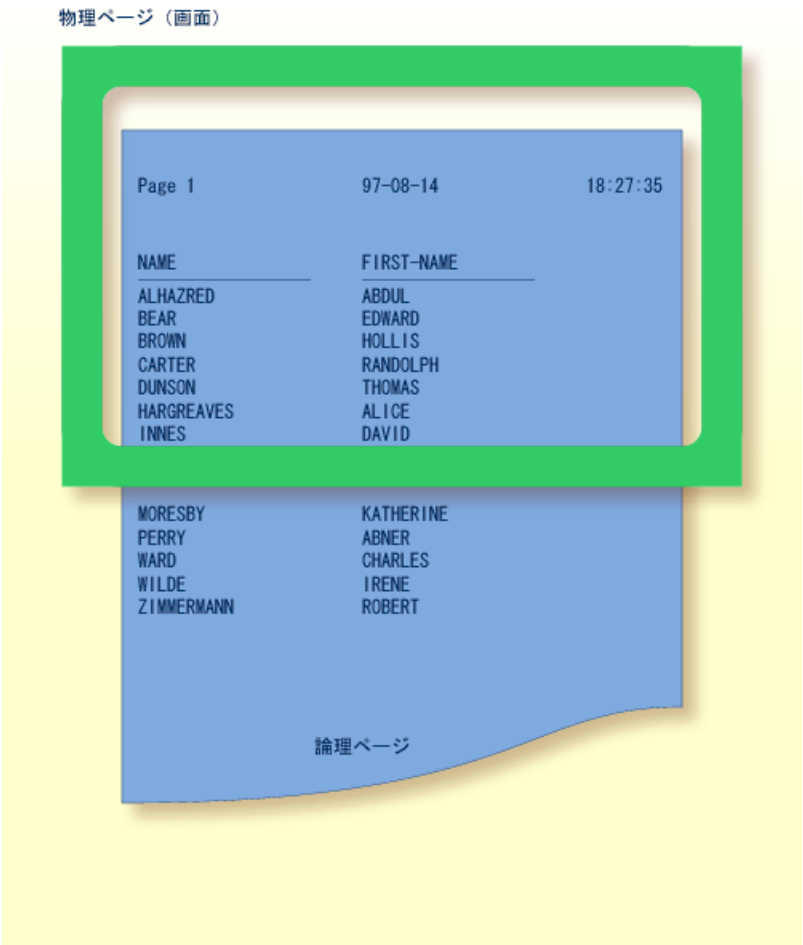
```
1980 RENAULT                  20000
1980 PEUGEOT                  20000
```

論理ページおよび物理ページ

*論理*ページは、Naturalプログラムによって作成される出力です。*物理*ページは、出力が表示される端末画面です。または、出力がプリントされる用紙 1 枚の場合もあります。

論理ページのサイズは、Natural プログラムによって出力される行数で決定されます。

1画面分より多い行数が出力された場合、論理ページは物理画面を超過し、残りの行は次の画面に表示されます。



Note: 画面の下に表示される情報（`WRITE TRAILER` または `AT END OF PAGE` ステートメントで作成される出力など）が次の画面に出力される場合は、下記で説明しているセッションパラメータ `PS` を使用して、論理ページのサイズを適宜に減らします。

ページサイズ - PS パラメータ

パラメータ PS（Natural レポートのページサイズ）を使用して、レポートの論理ページ当たりの最大行数を指定します。

PS パラメータで指定した行数に達すると、改ページが発生します。ただし、NEWPAGE または EJECT ステートメントで改ページが制御されていない場合に限られます。下記の「*EJ パラメータで制御する改ページ*」を参照してください。

PS パラメータは、システムコマンド GLOBALS を使用してセッションレベルで設定するか、または以下のステートメントを使用してプログラム内で設定できます。

レポートレベル：

- FORMAT PS=*nn*

ステートメントレベル：

- DISPLAY (PS=*nn*)
- WRITE (PS=*nn*)
- WRITE TITLE (PS=*nn*)
- WRITE TRAILER (PS=*nn*)
- INPUT (PS=*nn*)

改ページ

以下のいずれかの方法で改ページできます。

- EJ パラメータで制御する改ページ
- EJECT または NEWPAGE ステートメントで制御する改ページ
- *n* 行より少ない行数が残っている場合のページ換えまたは新しいページ

これらの方法について以下で説明します。

EJ パラメータで制御する改ページ

セッションパラメータ EJ（ページ換え）を使用して、ページ換えを実行するかどうかを指定します。 デフォルトでは EJ=ON が適用され、指定に従ってページ換えが実行されます。

EJ=OFF を指定すると、改ページ情報は無視されます。 これは、ページ換えを必要としないテスト実行時に用紙を節約するために役立ちます。

EJ パラメータは、次の例のように、システムコマンド GLOBALS を使用してセッションレベルで設定できます。

```
GLOBALS EJ=OFF
```

EJ パラメータの設定は、EJECT ステートメントによって上書きされます。

EJECT または NEWPAGE ステートメントで制御する改ページ

以下では次のトピックについて説明します。

- 次ページにタイトルまたはヘッダーを生成しない改ページ
- ページ終了処理およびページ開始処理を行う改ページ

次ページにタイトルまたはヘッダーを生成しない改ページ

EJECT ステートメントでは、次のページにタイトル行またはヘッダー行を生成しないで改ページが行われます。 新しい物理ページは、ページ開始処理またはページ終了処理（WRITE TRAILER、AT END OF PAGE、WRITE TITLE、AT TOP OF PAGE、*PAGE-NUMBER などの処理）を実行しないで、開始されます。

EJECT ステートメントは EJ パラメータの設定を上書きします。

ページ終了処理およびページ開始処理を行う改ページ

NEWPAGE ステートメントでは、関連するページ終了処理およびページ開始処理とともに改ページが行われます。 指定されている場合には、トレーラ行が表示されます。 DISPLAY または WRITE ステートメント で NOTITLE オプションが指定されていない場合には（上記参照）、デフォルトまたはユーザー指定のタイトル行が新しいページに表示されます。

NEWPAGE ステートメントを使用しない場合、改ページは PS パラメータの設定によって自動的に制御されます。上記の「*ページサイズ - PS パラメータ*」を参照してください。

n 行より少ない行数が残っている場合のページ換えまたは新しいページ

NEWPAGE ステートメントおよび EJECT ステートメントの両方で、WHEN LESS THAN *n* LINES LEFT オプションを指定できます。 このオプションでは、行数（*n*）を指定します。 NEWPAGE および EJECT ステートメントは、ステートメント処理時に現在のページの使用可能行数が *n* 行より少ない場合に実行されます。

例 **1**：

```
FORMAT PS=55
...
NEWPAGE WHEN LESS THAN 7 LINES LEFT
...
```

この例では、ページサイズは 55 行に設定されています。

NEWPAGE ステートメントの処理時に現在のページの残りの行数が 6 行以下の場合にのみ、NEWPAGE ステートメントが実行され、改ページが発生します。 残りの行数が 7 行以上の場合、NEWPAGE ステートメントは実行されず、改ページも発生しません。改ページはその後、セッションパラメータ PS（Natural レポートのページサイズ）の設定に従って、55 行目の後に発生します。

例 **2**：

```
** Example 'NEWPAX02': NEWPAGE (in combination with EJECT and
**                      parameter PS)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 JOB-TITLE
END-DEFINE
*
FORMAT PS=15
*
READ (9) EMPLOY-VIEW BY CITY STARTING FROM 'BOSTON'
  AT START OF DATA
    EJECT
    WRITE /// 20T '%' (29) /
              20T '%%'                            47T '%%' /
              20T '%%' 3X 'REPORT OF EMPLOYEES' 47T '%%' /
              20T '%%' 3X '  SORTED BY CITY   ' 47T '%%' /
              20T '%%'                            47T '%%' /
              20T '%' (29) /
    NEWPAGE
  END-START
  AT BREAK OF CITY
```

```
    NEWPAGE WHEN LESS 3 LINES LEFT
  END-BREAK
  DISPLAY CITY (IS=ON) NAME JOB-TITLE
END-READ
END
```

タイトル付きの新しいページ

NEWPAGE ステートメントでは、WITH TITLE オプションも使用できます。 このオプションを使用しない場合、デフォルトタイトルが新しいページの先頭に表示されるか、または WRITE TITLE ステートメントや NOTITLE 節が実行されます。

NEWPAGE ステートメントの WITH TITLE オプションにより、独自に選択したタイトルでこれらを上書きすることができます。 WITH TITLE オプションの構文は、WRITE TITLE ステートメントと同じです。

例：

```
NEWPAGE WITH TITLE LEFT JUSTIFIED 'PEOPLE LIVING IN BOSTON:'
```

以下のプログラムは、セッションパラメータ PS（Natural レポートのページサイズ）と NEWPAGE ステートメントの使用法を示しています。さらに、システム変数 *PAGE-NUMBER を使用して現在のページ番号を表示しています。

```
** Example 'NEWPAX01': NEWPAGE
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 DEPT
END-DEFINE
*
FORMAT PS=20
READ (5) VIEWEMP BY CITY STARTING FROM 'M'
  DISPLAY NAME 'DEPT' DEPT 'LOCATION' CITY
  AT BREAK OF CITY
    NEWPAGE WITH TITLE LEFT JUSTIFIED
           'EMPLOYEES BY CITY - PAGE:' *PAGE-NUMBER
  END-BREAK
END-READ
END
```

プログラム NEWPAX01 の出力：

改ページの位置とタイトル行に注意してください。

```
Page      1                                                  04-11-11  14:15:54

        NAME          DEPT        LOCATION
-------------------- ------ --------------------

FICKEN               TECH10 MADISON
KELLOGG              TECH10 MADISON
ALEXANDER            SALE20 MADISON
```

2 ページ目：

```
EMPLOYEES BY CITY - PAGE:      2
        NAME          DEPT        LOCATION
-------------------- ------ --------------------

DE JUAN              SALE03 MADRID
DE LA MADRID         PROD01 MADRID
```

3 ページ目：

```
EMPLOYEES BY CITY - PAGE: 3
```

ページトレーラ - WRITE TRAILER ステートメント

以下では次のトピックについて説明します。

- ページトレーラの指定
- 論理ページサイズの考慮事項

- ページトレーラの桁揃えと下線

ページトレーラの指定

WRITE TRAILER ステートメントは、ページの下に（アポストロフィで囲んだ）テキストを出力するために使用されます。

```
WRITE TRAILER 'THIS IS THE END OF THE PAGE'
```

ステートメントは、ページ終了条件が検出されたとき、または SKIP や NEWPAGE ステートメントの結果として実行されます。

論理ページサイズの考慮事項

ページ終了条件は DISPLAY または WRITE ステートメント全体が処理された後でのみチェックされるため、論理ページサイズ（DISPLAY または WRITE ステートメントによって出力される行数）によっては、WRITE TRAILER ステートメントが実行される前に、出力ページの物理サイズを超過する可能性があります。

実際に物理ページの下にページトレーラが表示されるようにするには、PS セッションパラメータを使用して、論理ページサイズを物理ページサイズより少ない値に設定する必要があります。

ページトレーラの桁揃えと下線

デフォルトでは、ページトレーラはページの中央に位置付けられ、下線は付きません。

WRITE TRAILER ステートメントでは、互いに独立して使用可能な以下のオプションを指定できます。

オプション	効果
LEFT JUSTIFIED	ページトレーラを左揃えで表示します。
UNDERLINED	行サイズの幅に下線が引かれます（Natural プロファイルおよびセッションパラメータ LS も参照）。 デフォルトでは、タイトルにハイフン（-）で下線が引かれます。 ただし、UC セッションパラメータを使用すると、下線用の文字として使用する他の文字を指定してすることができます（「*タイトルおよびヘッダーの下線用の文字*」を参照）。

以下の例は、WRITE TRAILER ステートメントの LEFT JUSTIFIED および UNDERLINED オプションの使用法を示しています。

例 **1**：

```
WRITE TRAILER LEFT JUSTIFIED UNDERLINED 'THIS IS THE END OF THE PAGE'
```

例 **2**：

```
** Example 'WTITLX02': WRITE TITLE AND WRITE TRAILER
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 FIRST-NAME
  2 ADDRESS-LINE (1)
END-DEFINE
*
WRITE TITLE LEFT JUSTIFIED UNDERLINED
       *TIME
   5X  'PEOPLE LIVING IN SALT LAKE CITY'
   21X 'PAGE:' *PAGE-NUMBER /
   15X 'AS OF' *DAT4E //
*
WRITE TRAILER UNDERLINED 'REGISTER OF' / 'SALT LAKE CITY'
*
READ (2) EMPLOY-VIEW WITH CITY = 'SALT LAKE CITY'
  DISPLAY  NAME /
           FIRST-NAME
           'HOME/CITY' CITY
           'STREET/OR BOX NO.' ADDRESS-LINE (1)
  SKIP 1
END-READ
END
```

空行の生成 - SKIP ステートメント

SKIP ステートメントは、1 つまたは複数の空行を出力レポートに生成するために使用します。

例 **1 - WRITE** および **DISPLAY** とともに使用する **SKIP**：

```
** Example 'SKIPX01': SKIP (in conjunction with WRITE and DISPLAY)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 FIRST-NAME
  2 ADDRESS-LINE (1)
```

```
END-DEFINE
*
WRITE TITLE LEFT JUSTIFIED UNDERLINED
     'PEOPLE LIVING IN SALT LAKE CITY AS OF' *DAT4E 7X
     'PAGE:' *PAGE-NUMBER
SKIP 3
*
READ (2) EMPLOY-VIEW WITH CITY = 'SALT LAKE CITY'
  DISPLAY NAME / FIRST-NAME CITY ADDRESS-LINE (1)
  SKIP 1
END-READ
END
```

例 **2 - DISPLAY VERT** とともに使用する **SKIP**：

```
** Example 'SKIPX02': SKIP (in conjunction with DISPLAY VERT)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
  2 JOB-TITLE
END-DEFINE
*
READ (2) EMPLOY-VIEW WITH JOB-TITLE = 'SECRETARY'
  DISPLAY NOTITLE VERT
          NAME FIRST-NAME / CITY
  SKIP 3
END-READ
*
NEWPAGE
*
READ (2) EMPLOY-VIEW WITH JOB-TITLE = 'SECRETARY'
  DISPLAY NOTITLE
          NAME FIRST-NAME / CITY
  SKIP 3
```

```
END-READ
END
```

AT TOP OF PAGE ステートメント

AT TOP OF PAGE ステートメントは、レポートの新しいページが始まるときに常に実行される処理を指定するために使用します。

AT TOP OF PAGE 処理で何らかの出力が作成される場合、ページタイトルの下に、間にスキップした行をはさんで出力されます。

デフォルトでは、この出力は左揃えでページに表示されます。

例：

```
** Example 'ATTOPX01': AT TOP OF PAGE
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 MAR-STAT
  2 BIRTH
  2 CITY
  2 JOB-TITLE
  2 DEPT
END-DEFINE
*
LIMIT 10
READ EMPLOY-VIEW BY PERSONNEL-ID FROM '20017000'
  DISPLAY NOTITLE (AL=10)
          NAME DEPT JOB-TITLE CITY 5X
          MAR-STAT 'DATE OF/BIRTH' BIRTH (EM=YY-MM-DD)
  /*
  AT TOP OF PAGE
    WRITE /   '-BUSINESS INFORMATION-'
          26X '-PRIVATE INFORMATION-'
  END-TOPPAGE
END-READ
END
```

プログラム ATTOPX01 の出力：

```
-BUSINESS INFORMATION-                          -PRIVATE INFORMATION-
   NAME    DEPARTMENT  CURRENT      CITY         MARITAL   DATE OF
              CODE     POSITION                   STATUS    BIRTH
---------- ---------- ---------- ----------     ---------- --------

CREMER     TECH10     ANALYST    GREENVILLE     S          70-01-01
MARKUSH    SALE00     TRAINEE    LOS ANGELE     D          79-03-14
GEE        TECH05     MANAGER    CHAPEL HIL     M          41-02-04
KUNEY      TECH10     DBA        DETROIT        S          40-02-13
NEEDHAM    TECH10     PROGRAMMER CHATTANOOG     S          55-08-05
JACKSON    TECH10     PROGRAMMER ST LOUIS       D          70-01-01
PIETSCH    MGMT10     SECRETARY  VISTA          M          40-01-09
PAUL       MGMT10     SECRETARY  NORFOLK        S          43-07-07
HERZOG     TECH05     MANAGER    CHATTANOOG     S          52-09-16
DEKKER     TECH10     DBA        MOBILE         W          40-03-03
```

AT END OF PAGE ステートメント

AT END OF PAGE ステートメントは、ページ終了条件が発生したときに常に実行される処理を指定するために使用します。

AT END OF PAGE 処理により何らかの出力が作成される場合、WRITE TRAILER ステートメントで指定したページトレーラの後に出力されます。

デフォルトでは、この出力は左揃えでページに表示されます。

上記で説明したページトレーラに関わる物理および論理ページサイズと DISPLAY または WRITE ステートメントで出力される行数の考慮事項は、AT END OF PAGE 出力にも同様に適用されます。

例：

```
** Example 'ATENPX01': AT END OF PAGE (with system function available
**                      via GIVE SYSTEM FUNCTIONS in DISPLAY)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 JOB-TITLE
  2 SALARY (1)
END-DEFINE
*
READ (10) EMPLOY-VIEW BY PERSONNEL-ID = '20017000'
  DISPLAY NOTITLE GIVE SYSTEM FUNCTIONS
          NAME JOB-TITLE 'SALARY' SALARY(1)
```

```
  /*
  AT END OF PAGE
    WRITE / 24T 'AVERAGE SALARY: ...' AVER(SALARY(1))
  END-ENDPAGE
END-READ
END
```

プログラム ATENPX01 の出力：

```
        NAME                  CURRENT            SALARY
                             POSITION
-------------------- -------------------------- ----------

CREMER               ANALYST                         34000
MARKUSH              TRAINEE                         22000
GEE                  MANAGER                         39500
KUNEY                DBA                             40200
NEEDHAM              PROGRAMMER                      32500
JACKSON              PROGRAMMER                      33000
PIETSCH              SECRETARY                       22000
PAUL                 SECRETARY                       23000
HERZOG               MANAGER                         48500
DEKKER               DBA                             48000


AVERAGE SALARY: ...      34270
```

その他の例

次の例のプログラムを参照してください。

- ***DISPLX21 - DISPLAY（スラッシュ '/' を使用、WRITE と比較）***

38 列ヘッダー

このchapterでは、DISPLAY ステートメントで作成される列ヘッダーの表示を制御する方法について説明します。

次のトピックについて説明します。

デフォルトの列ヘッダー

デフォルトでは、DISPLAY ステートメントによるデータベースフィールドの各出力は、DDM のフィールドに定義されているデフォルトの列ヘッダー付きで表示されます。

```
** Example 'DISPLX01': DISPLAY
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 BIRTH
  2 JOB-TITLE
END-DEFINE
*
READ (3) VIEWEMP BY BIRTH
  DISPLAY PERSONNEL-ID NAME JOB-TITLE
END-READ
END
```

プログラム DISPLX01 の出力：

上記のプログラム例では、デフォルトのヘッダーを使用して、以下の出力が作成されます。

```
Page      1                                              04-11-11  14:15:54

PERSONNEL         NAME                  CURRENT
   ID                                   POSITION
--------- -------------------- -------------------------

30020013  GARRET               TYPIST
```

```
30016112  TAILOR               WAREHOUSEMAN
20017600  PIETSCH              SECRETARY
```

デフォルトの列ヘッダーの省略 - NOHDR オプション

列ヘッダーなしでレポートを出力する場合は、DISPLAY ステートメントにキーワード NOHDR を追加します。

```
DISPLAY NOHDR PERSONNEL-ID NAME JOB-TITLE
```

独自の列ヘッダーの定義

デフォルトヘッダーの代わりに独自の列ヘッダーを出力する場合、フィールドの直前に '*text*'（アポストロフィ付き）を指定します。*text* がそのフィールドに使用するヘッダーとなります。

```
** Example 'DISPLX08': DISPLAY  (with column title in 'text')
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 BIRTH
  2 JOB-TITLE
END-DEFINE
*
READ (3) VIEWEMP BY BIRTH
  DISPLAY PERSONNEL-ID
          'EMPLOYEE' NAME
          'POSITION' JOB-TITLE
END-READ
END
```

プログラム DISPLX08 の出力：

上記のプログラムには、フィールド NAME のヘッダー "EMPLOYEE"、およびフィールド JOB-TITLE のヘッダー "POSITION" が含まれています。フィールド PERSONNEL-ID についてはデフォルトヘッダーが使用されます。 プログラムが次の出力を生成します。

```
Page      1                                                  04-11-11  14:15:54

PERSONNEL       EMPLOYEE               POSITION
   ID
--------- -------------------- -------------------------

30020013  GARRET               TYPIST
30016112  TAILOR               WAREHOUSEMAN
20017600  PIETSCH              SECRETARY
```

NOTITLE と NOHDR の組み合わせ

ページタイトルも列ヘッダーもないレポートを作成するには、NOTITLEおよびNOHDRオプションを以下の順番で一緒に指定します。

```
DISPLAY NOTITLE NOHDR PERSONNEL-ID NAME JOB-TITLE
```

列ヘッダーの中央揃え - HC パラメータ

デフォルトでは、列ヘッダーは列の上に中央揃えで出力されます。HCパラメータを使用すると、列ヘッダーの配置を操作することができます。

有効な指定は以下のとおりです。

HC=L	ヘッダーは左揃えで出力されます。
HC=R	ヘッダーは右揃えで出力されます。
HC=C	ヘッダーは中央揃えで出力されます。

HCパラメータをFORMATステートメントで使用してレポート全体に適用させるか、またはDISPLAYステートメントで使用してステートメントレベルおよび要素レベルの両方で適用させることができます。

```
DISPLAY (HC=L) PERSONNEL-ID NAME JOB-TITLE
```

列ヘッダーの幅 - HW パラメータ

HW パラメータを使用して、DISPLAY ステートメントで出力される列の幅を指定します。

有効な指定は以下のとおりです。

HW=ON	DISPLAY 列の幅は、ヘッダーテキストの長さまたはフィールドの長さのどちらか長い方で決定されます。 これは、デフォルトで適用されます。
HW=OFF	DISPLAY 列の幅は、フィールドの長さのみで決定されます。 ただし、HW=OFF はヘッダーを作成しない DISPLAY ステートメント、つまり、NOHDR オプション付きの最初の DISPLAY ステートメントまたは後続の DISPLAY ステートメントにのみ適用されます。

HW パラメータを FORMAT ステートメントで使用してレポート全体に適用させるか、または DISPLAY ステートメントで使用してステートメントレベルおよび要素（フィールド）レベルの両方で適用させることができます。

ヘッダーの充填文字 - FC および GC パラメータ

列の幅をヘッダーではなくフィールド長で決定する場合、FC パラメータを使用して、DISPLAY ステートメントで出力されるヘッダーの両側に列幅全体に渡って表示される*充填文字*を指定します（上記の HW パラメータを参照）。フィールド長で決定しない場合、FC は無視されます。

フィールドグループまたはピリオディックグループを DISPLAY ステートメントで出力する場合、グループ内の個々のフィールドのヘッダーの上に、そのグループに属しているすべてのフィールド列に渡って、*グループヘッダー*が表示されます。 GC パラメータを使用すると、このようなグループヘッダーの両側に表示される*充填文字*を指定できます。

FC パラメータが個々のフィールドのヘッダーに適用されるのに対して、GC パラメータはフィールドグループのヘッダーに適用されます。

パラメータ FC および GC を FORMAT ステートメントで指定してレポート全体に適用させるか、または、DISPLAY ステートメントで指定してステートメントレベルおよび要素（フィールド）レベルの両方で適用させることができます。

```
** Example 'FORMAX01': FORMAT (with parameters FC, GC)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 INCOME (1:1)
    3 CURR-CODE
```

```
    3 SALARY
    3 BONUS (1:1)
END-DEFINE
*
FORMAT FC=* GC=$
*
READ (3) VIEWEMP BY NAME
  DISPLAY NAME (FC==) INCOME (1)
END-READ
END
```

プログラム FORMAX01 の出力：

```
Page      1                                                  04-11-11  14:15:54

========NAME======== $$$$$$$$$$$$INCOME$$$$$$$$$$$$

                     CURRENCY **ANNUAL** **BONUS***
                         CODE     SALARY
-------------------- -------- ---------- ----------

ABELLAN              PTA         1450000          0
ACHIESON             UKL           10500          0
ADAM                 FRA          159980      23000
```

タイトルおよびヘッダーの下線付き文字 - UC パラメータ

デフォルトでは、タイトルおよびヘッダーにハイフン（-）で下線が引かれます。

UC パラメータを使用すると、下線用の文字として使用する別の文字を指定できます。

UC パラメータを FORMAT ステートメントで指定してレポート全体に適用させるか、または DISPLAY ステートメントで指定してステートメントレベルおよび要素（フィールド）レベルの両方で適用させることができます。

```
** Example 'FORMAX02': FORMAT (with parameter UC)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 BIRTH
  2 JOB-TITLE
END-DEFINE
*
FORMAT UC==
```

```
*
WRITE TITLE LEFT JUSTIFIED UNDERLINED 'EMPLOYEES REPORT'
SKIP 1
READ (3) VIEWEMP BY BIRTH
  DISPLAY PERSONNEL-ID (UC=*) NAME JOB-TITLE
END-READ
END
```

上記のプログラムでは、UC パラメータがプログラムレベルおよび要素（フィールド）レベルで指定されています。FORMAT ステートメントで指定された下線文字（=）は、別の下線文字（*）が指定されているフィールド PERSONNEL-ID を除いて、レポート全体に適用されます。

プログラム FORMAX02 の出力：

```
EMPLOYEES REPORT
================================================================================

PERSONNEL         NAME                 CURRENT
   ID                                  POSITION
********* ==================== =========================

30020013  GARRET               TYPIST
30016112  TAILOR               WAREHOUSEMAN
20017600  PIETSCH              SECRETARY
```

列ヘッダーの省略 - スラッシュ表記

アポストロフィスラッシュアポストロフィ（'/'）の表記を使用すると、DISPLAY ステートメントで表示される個々のフィールドのデフォルトの列ヘッダーを省略できます。 NOHDR オプションがすべての列ヘッダーを省略するのに対して、'/' 表記は個々の列ヘッダーを省略するために使用できます。

アポストロフィスラッシュアポストロフィ（'/'）の表記は、DISPLAY ステートメントで、列ヘッダーを省略するフィールドの名前の直前に指定します。

以下の 2 つの例を比較してください。

例 **1**：

```
DISPLAY NAME PERSONNEL-ID JOB-TITLE
```

この場合、3 つすべてのフィールドのデフォルトの列ヘッダーが表示されます。

```
Page      1                                                  04-11-11  14:15:54

        NAME         PERSONNEL          CURRENT
                          ID             POSITION
-------------------- --------- -------------------------

ABELLAN              60008339  MAQUINISTA
ACHIESON             30000231  DATA BASE ADMINISTRATOR
ADAM                 50005800  CHEF DE SERVICE
ADKINSON             20008800  PROGRAMMER
ADKINSON             20009800  DBA
ADKINSON             20011000  SALES PERSON
```

例 **2**：

```
DISPLAY '/' NAME PERSONNEL-ID JOB-TITLE
```

この場合、'/' 表記によってフィールド NAME の列ヘッダーが省略されます。

```
Page      1                                                  04-11-11  14:15:54

                     PERSONNEL          CURRENT
                        ID             POSITION
                     --------- -------------------------

ABELLAN              60008339  MAQUINISTA
ACHIESON             30000231  DATA BASE ADMINISTRATOR
ADAM                 50005800  CHEF DE SERVICE
ADKINSON             20008800  PROGRAMMER
ADKINSON             20009800  DBA
ADKINSON             20011000  SALES PERSON
```

列ヘッダーの他の例

次の例のプログラムを参照してください。

- ***DISPLX15 - DISPLAY（FC、UC を使用）***
- ***DISPLX16 - DISPLAY（'/'、'text'、'text/text' を使用）***

39 フィールドの出力に影響を与えるパラメータ

このchapterでは、フィールドの出力フォーマットの制御に使用できる Natural プロファイルやセッションパラメータの使用法について説明します。

次のトピックについて説明します。

フィールド出力関連パラメータの概要

Naturalには、フィールドを出力する際のフォーマットを制御するために使用できるプロファイルやセッションパラメータが複数用意されています。

パラメータ	機能
LC、IC、および TC	これらのセッションパラメータで、フィールドの前後またはフィールド値の前に表示する文字を指定できます。
LCU、ICU、および TCU	これらのセッションパラメータで、フィールドの前後またはフィールド値の前に表示する文字を Unicode フォーマットで指定できます。
AL および NL	これらのセッションパラメータで、フィールドの出力長を増減できます。
DL	このセッションパラメータで、フォーマット U の英数字マップフィールドのデフォルトの出力長を指定できます。
SG	このセッションパラメータで、負の値をマイナス記号付きまたは記号なしのどちらで表示するかを決定できます。
IS	このセッションパラメータで、後続の同一フィールド値の表示を省略できます。
ZP	このプロファイルおよびセッションパラメータで、"0" のフィールド値を表示するかどうかを決定できます。
ES	このセッションパラメータで、`DISPLAY` または `WRITE` ステートメントによって生成された空白行の表示を省略できます。

これらのパラメータについて以下で説明します。

先頭文字 - LC パラメータ

セッションパラメータ `LC` では、`DISPLAY` ステートメントで出力される*フィールドの直前*に表示される先頭文字を指定できます。 出力列の幅は、それに従って拡大します。 1～10 文字を指定できます。

デフォルトでは、値は、英数字フィールドでは左揃え、数値フィールドでは右揃えで表示されます。 これらのデフォルトは `AD` パラメータで変更できます。『*パラメータリファレンス*』を参照してください。 英数字フィールドに先頭文字を指定すると、文字はフィールド値の直前に表示されます。数値フィールドの場合、先頭文字とフィールド値の間に多数のスペースが表示されることがあります。

LC パラメータは、以下のステートメントで使用できます。

- FORMAT
- DISPLAY

ステートメントレベルと要素レベルで設定できます。

Unicode 先頭文字 - LCU パラメータ

セッションパラメータ LCU は、セッションパラメータ LC と同一です。 違いは、先頭文字が必ず Unicode フォーマットで保存されることです。

そのため、ユーザーは別のコードページからの文字が混在した先頭文字を指定でき、インストールされているシステムコードページに関係なく、いつでも確実に正しい文字が表示されるようにすることができます。

詳細については、「*Natural プログラミング言語の Unicode とコードページのサポート*」の「*セッションパラメータ*」にある「*EMU、ICU、LCU、TCU と EM、IC、LC、TC の比較*」を参照してください。

パラメータ LCU および ICU の両方を 1 つのフィールドに適用することはできません。

挿入文字 - IC パラメータ

セッションパラメータ IC を使用して、DISPLAY ステートメントで出力される*フィールド値の直前*に挿入され文字を指定します。 1～10 文字を指定できます。

数値フィールドの場合、挿入文字は、出力される最初の有効桁の直前に挿入されます。指定した文字とフィールド値の間に空白は入りません。 英数字フィールドの場合、IC パラメータの効果は LC パラメータと同じです。

パラメータ LC および IC の両方を 1 つのフィールドに適用することはできません。

IC パラメータは、以下のステートメントで使用できます。

- FORMAT
- DISPLAY

ステートメントレベルと要素レベルで設定できます。

Unicode 挿入文字 - ICU パラメータ

セッションパラメータ ICU は、セッションパラメータ IC と同一です。 違いは、挿入文字が必ず Unicode フォーマットで保存されることです。

そのため、ユーザーは別のコードページからの文字が混在した挿入文字を指定でき、インストールされているシステムコードページに関係なく、いつでも確実に正しい文字が表示されるようにすることができます。

詳細については、「*Natural プログラミング言語の Unicode とコードページのサポート*」の「*セッションパラメータ*」にある「*EMU、ICU、LCU、TCU と EM、IC、LC、TC の比較*」を参照してください。

パラメータ LCU および ICU の両方を 1 つのフィールドに適用することはできません。

末尾文字 - TC パラメータ

セッションパラメータ TC では、DISPLAY ステートメントで出力される*フィールドのすぐ右側*に表示される末尾文字を指定できます。 出力列の幅は、それに従って拡大します。 1〜10 文字を指定できます。

TC パラメータは、以下のステートメントで使用できます。

- FORMAT
- DISPLAY

ステートメントレベルと要素レベルで設定できます。

Unicode 末尾文字 - TCU パラメータ

セッションパラメータ TCU は、セッションパラメータ TC と同一です。 違いは、末尾文字が必ず Unicode フォーマットで保存されることです。

そのため、ユーザーは別のコードページからの文字が混在した末尾文字を指定でき、インストールされているシステムコードページに関係なく、いつでも確実に正しい文字が表示されるようにすることができます。

詳細については、「*Natural プログラミング言語の Unicode とコードページのサポート*」の「*セッションパラメータ*」にある「*EMU、ICU、LCU、TCU と EM、IC、LC、TC の比較*」を参照してください。

出力長 - AL パラメータと NL パラメータ

セッションパラメータ AL では、*英数字フィールドの出力長*を指定できます。NL パラメータでは、*数値フィールドの出力長*を指定できます。 これにより、出力されるフィールドの長さが決定されます。出力されるフィールドの長さは、実際のフィールドの長さより短い場合も長い場合もあります。実際のフィールドの長さは、データベースフィールドの場合はDDM、ユーザー定義変数の場合は DEFINE DATA ステートメントで定義されます。

どちらのパラメータも以下のステートメントで使用できます。

- FORMAT
- DISPLAY
- WRITE
- PRINT
- INPUT

ステートメントレベルと要素レベルで設定できます。

Note: 編集マスクを指定している場合、NL または AL 指定は上書きされます。 編集マスクについては、「*編集マスク - EM パラメータ*」に記載されています。

出力の表示長 - DL パラメータ

Unicode文字列の表示幅は文字列の長さの2倍になる可能性があり、ユーザーは文字列全体を表示できる必要があるため、セッションパラメータ DL で、*フォーマット A または U のフィールドの表示長*を指定できます。 デフォルトはフィールド長であり、例えば、フォーマットおよび長さが U10 の場合、DL が指定されていないときのデフォルトの長さが 10 であるのに対し、表示長は 10～20 になる可能性があります。

このセッションパラメータは以下のステートメントで使用できます。

- FORMAT
- DISPLAY
- WRITE
- PRINT
- INPUT

ステートメントレベルと要素レベルで設定できます。

セッションパラメータ AL と DL の違いは、AL がフィールドのデータ長を定義するのに対し、DL はフィールドを表示するために使用される画面上の桁数を定義します。 DL セッションパラメータで指定した値がフィールドデータの長さよりも少ない場合、ユーザーは入力フィールド内をスクロールして、フィールドの内容全体を表示することができます。

Note: DL は A フォーマットフィールドでも同様に使用できます。 これにより、編集のコントロールサイズをフィールドの内容よりも小さくすることができます。

例：

```
DEFINE DATA LOCAL
1    #U1 (U10)
1    #U2 (U10)
END-DEFINE
*
#U1 := U'latintxt00'
#U2 := U'特別是伺服器都需要支'
*
INPUT (AD=M) #U1 #U2
END
```

上記のプログラムでは以下の出力が作成されます。フィールド #U2 の内容は不完全です。

```
#U1 latintxt00 #U2 特別是伺服
```

セッションパラメータ DL をフィールド #U2 で使用して適宜に指定すると、このフィールドの内容が正しく表示されます。

```
DEFINE DATA LOCAL
1    #U1 (U10)
1    #U2 (U10)
END-DEFINE
*
#U1 := U'latintxt00'
#U2 := U'特別是伺服器都需要支'
*
```

```
INPUT (AD=M) #U1 #U2 (DL=20)
END
```

結果：

```
#U1 latintxt00 #U2 特別是伺服器都需要支
```

符号の位置 - SG パラメータ

セッションパラメータ SG では、数値フィールドに符号の位置を割り当てるかどうかを決定できます。

- デフォルトでは SG=ON が適用され、数値フィールドに符号の位置が割り当てられます。
- SG=OFF を指定すると、数値フィールドの負の値はマイナス記号（-）なしで出力されます。

SG パラメータは以下のステートメントで使用できます。

- FORMAT
- DISPLAY
- PRINT
- WRITE
- INPUT

ステートメントレベルと要素レベルの両方で設定できます。

Note: 編集マスクを指定している場合、SG 指定は上書きされます。 編集マスクについては、「*編集マスク - EM パラメータ*」に記載されています。

パラメータなしのプログラム例

```
** Example 'FORMAX03': FORMAT (without FORMAT and compare with FORMAX04)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 SALARY (1:1)
  2 BONUS  (1:1,1:1)
END-DEFINE
*
READ (5) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY NAME
          FIRST-NAME
```

```
          SALARY (1:1)
          BONUS  (1:1,1:1)
END-READ
END
```

上記のプログラムにはパラメータ設定が含まれず、出力は以下のようになります。

```
 Page      1                                                 04-11-11  11:11:11

          NAME              FIRST-NAME        ANNUAL     BONUS
                                              SALARY
 -------------------- -------------------- ---------- ----------

 JONES                VIRGINIA                  46000       9000
 JONES                MARSHA                    50000          0
 JONES                ROBERT                    31000          0
 JONES                LILLY                     24000          0
 JONES                EDWARD                    37600          0
```

パラメータ AL、NL、LC、IC、および TC を使用したプログラム例

この例では、セッションパラメータ AL、NL、LC、IC、および TC が使用されています。

```
** Example 'FORMAX04': FORMAT (with parameters AL, NL, LC, TC, IC and
**                      compare with FORMAX03)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 SALARY (1:1)
  2 BONUS  (1:1,1:1)
END-DEFINE
*
FORMAT AL=10 NL=6
*
READ (5) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY NAME            (LC=*)
          FIRST-NAME      (TC=*)
          SALARY (1:1)    (IC=$)
          BONUS (1:1,1:1) (LC=>)
END-READ
END
```

上記のプログラムでは、次の出力が生成されます。個々のパラメータの影響を確認するために、この出力のレイアウトを前のプログラムと比較します。

```
Page      1                                                  04-11-11  11:11:11

   NAME      FIRST-NAME    ANNUAL    BONUS
                           SALARY
----------- ----------- -------- --------

*JONES       VIRGINIA  *   $46000 >   9000
*JONES       MARSHA    *   $50000 >      0
*JONES       ROBERT    *   $31000 >      0
*JONES       LILLY     *   $24000 >      0
*JONES       EDWARD    *   $37600 >      0
```

上記の例でわかるように、AL または NL パラメータで指定した出力長には、LC、IC、および TC パラメータで指定した文字は含まれません。例えば、NAME 列の幅は、フィールド値の 10 文字（AL=10）に先頭文字の 1 文字を足した 11 文字になります。

SALARY および BONUS 列の幅は、フィールド値の 6 文字（NL=6）に、先頭または挿入文字の 1 文字、および符号の位置（SG=ON が適用）の 1 文字を足した 8 文字になります。

重複抑制 - IS パラメータ

セッションパラメータ IS では、WRITE または DISPLAY ステートメントによって作成される連続行で同一情報の表示を抑制できます。

- デフォルトでは、IS=OFF が適用され、同一のフィールド値が表示されます。
- IS=ON を指定した場合、そのフィールドの前の値と同一の値は表示されません。

IS パラメータは以下のステートメントで指定できます。

- レポート全体に適用される FORMAT ステートメント。
- ステートメントレベルおよび要素レベルで適用される DISPLAY または WRITE ステートメント。

ステートメント SUSPEND IDENTICAL SUPPRESS を使用すると、パラメータ IS=ON の効果を 1 件のレコードに対して一時的に無効にできます。詳細については、『ステートメント』ドキュメントを参照してください。

次の 2 つのプログラム例の出力を比較して、IS パラメータの影響を確認します。 2 番目のプログラムでは、NAME フィールドの同一の値は表示されません。

IS パラメータなしのプログラム例

```
** Example 'FORMAX05': FORMAT (without parameter IS
**                      and compare with FORMAX06)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
END-DEFINE
*
READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY NAME FIRST-NAME
END-READ
END
```

上述したプログラムにより、次の出力が生成されます。

```
  Page      1                                                 04-11-11  11:11:11

           NAME              FIRST-NAME
  ------------------- --------------------

  JONES                VIRGINIA
  JONES                MARSHA
  JONES                ROBERT
```

IS パラメータありのプログラム例

```
** Example 'FORMAX06': FORMAT (with parameter IS
**                      and compare with FORMAX05)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
END-DEFINE
*
FORMAT IS=ON
*
READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY NAME FIRST-NAME
END-READ
END
```

上述したプログラムにより、次の出力が生成されます。

```
Page      1                                                  04-11-11  11:54:01

        NAME              FIRST-NAME
-------------------- --------------------

JONES                VIRGINIA
                     MARSHA
                     ROBERT
```

ゼロ出力 - ZP パラメータ

プロファイルおよびセッションパラメータ ZP では、ゼロのフィールド値をどのように表示するかを決定します。

- デフォルトでは ZP=ON が適用されるため、0 の各フィールド値に対して 1 つの "0"（数値フィールド）またはすべてゼロ（時間フィールド）が表示されます。
- ZP=OFF を指定すると、ゼロの各フィールド値は表示されません。

ZP パラメータは以下のステートメントで指定できます。

- レポート全体に適用される FORMAT ステートメント。
- ステートメントレベルおよび要素レベルで適用される DISPLAY または WRITE ステートメント。

次の 2 つの例のプログラムの出力を比較して、パラメータ ZP および ES の影響を確認します。

空行の省略 - ES パラメータ

セッションパラメータ ES では、DISPLAY または WRITE ステートメントによって作成された空行の出力を省略できます。

- デフォルトでは ES=OFF が適用されるため、すべて空白値を含む行が表示されます。
- ES=ON を指定した場合、すべて空白値を含む DISPLAY または WRITE ステートメントの結果行は表示されません。 マルチプルバリューフィールドまたはピリオディックグループの一部のフィールドを表示するときに、大量の空行が出力される可能性がある場合には、特に有効です。

ES パラメータは以下のステートメントで指定できます。

- レポート全体に適用される FORMAT ステートメント。
- ステートメントレベルで適用される DISPLAY または WRITE ステートメント。

Note: 数値に対して空白の省略を行うには、ES=ON 以外にパラメータ ZP=OFF も関連するフィールドに設定する必要があります。これは、NULL 値を空白に変えてどちらも出力されないようにするためです。

次の 2 つの例のプログラムの出力を比較して、パラメータ ZP および ES の影響を確認します。

パラメータ ZP および ES を使用しないプログラム例

```
** Example 'FORMAX07': FORMAT (without parameter ES and ZP
**                      and compare with FORMAX08)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 BONUS (1:2,1:1)
END-DEFINE
*
READ (4) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY NAME FIRST-NAME BONUS (1:2,1:1)
END-READ
END
```

上述したプログラムにより、次の出力が生成されます。

```
Page      1                                                 04-11-11  11:58:23

        NAME              FIRST-NAME        BONUS
-------------------- -------------------- ----------

JONES                VIRGINIA                  9000
                                               6750
JONES                MARSHA                       0
                                                  0
JONES                ROBERT                       0
                                                  0
```

```
JONES                LILLY                          0
                                                    0
```

パラメータ ZP および ES を使用したプログラム例

```
** Example 'FORMAX08': FORMAT (with parameters ES and ZP
**                       and compare with FORMAX07)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 BONUS (1:2,1:1)
END-DEFINE
*
FORMAT ES=ON
*
READ (4) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY NAME FIRST-NAME BONUS (1:2,1:1)(ZP=OFF)
END-READ
END
```

上述したプログラムにより、次の出力が生成されます。

```
Page      1                                              04-11-11  11:59:09

        NAME              FIRST-NAME        BONUS
-------------------- -------------------- ----------

JONES                VIRGINIA                  9000
                                               6750
JONES                MARSHA
JONES                ROBERT
JONES                LILLY
```

フィールド出力関連パラメータの他の例

パラメータ LC、IC、TC、AL、NL、IS、ZP、ES、および SUSPEND IDENTICAL SUPPRESS ステートメントの他の例については、以下のプログラム例を参照してください。

- *DISPLX17 - DISPLAY（NL、AL、IC、LC、TC を使用）*
- *DISPLX18 - DISPLAY（SF、AL、UC、LC、IC、TC のデフォルト設定を使用、DISPLX19 と比較）*
- *DISPLX19 - DISPLAY（SF、AL、LC、IC、TC を使用、DISPLX18 と比較）*

- ***SUSPEX01 - SUSPEND IDENTICAL SUPPRESS（DISPLAY でパラメータ IS、ES、ZP とともに使用）***
- ***SUSPEX02 - SUSPEND IDENTICAL SUPPRESS（DISPLAY でパラメータ IS、ES、ZP とともに使用）。 SUSPEX01 と同一、ただし IS=OFF を使用。***
- ***COMPRX03 - COMPRESS***

40 編集マスク - EM パラメータ

このchapterでは、英数字または数値フィールドの編集マスクを指定する方法について説明します。

次のトピックについて説明します。

EM パラメータの使用

セッションパラメータ EM を使用すると、英数字または数値フィールドに対して編集マスクを指定できます。つまり、フィールド値が出力されるフォーマットを文字単位で指定できます。セッションパラメータ EMU を使用すると、下記で説明する EM セッションパラメータと同じ方法で、Unicode 文字を使用した編集マスクを定義できます。

例：

```
DISPLAY NAME (EM=X^X^X^X^X^X^X^X^X)
```

この例で、それぞれの "X" は表示される英数字フィールド値の 1 文字を表し、それぞれの "^" は空白を表します。 DISPLAY ステートメントで表示された場合、名前 "JOHNSON" は次のように表示されます。

```
J O H N S O N
```

セッションパラメータ EM は以下のレベルで指定できます。

- レポートレベル（FORMAT ステートメント）。
- ステートメントレベル（DISPLAY、WRITE、INPUT、MOVE EDITED、または PRINT ステートメント）。
- 要素レベル（DISPLAY、WRITE、または INPUT ステートメント）。

セッションパラメータ EM で指定した編集マスクは、**DDM** のフィールドに指定されたデフォルトの編集マスクを上書きします。「*DDM の編集 - フィールド属性定義*」の「*拡張フィールド属性*」を参照してください。

EM=OFF を指定すると、編集マスクは使用されません。

ステートメントレベルで指定した編集マスクは、レポートレベルで指定した編集マスクを上書きします。

要素レベルで指定した編集マスクは、ステートメントレベルで指定した編集マスクを上書きします。

数値フィールドの編集マスク

数値フィールド（フォーマット N、I、P、F）の編集マスクは、数字（0 の場合も含む）で埋める出力位置ごとに "9" を含んでいる必要があります。

- 有効数字が 0 以外の場合にのみ出力位置を埋めるよう指示するには、"Z" を使用します。
- 小数点はピリオド（.）で示します。

小数点の右側に "Z" を指定することはできません。 符号インジケータなどの先頭文字、末尾文字、および挿入文字は追加できます。

英数字フィールドの編集マスク

英数字フィールドの編集マスクは、出力する英数字ごとに "X" が含まれている必要があります。

いくつかの例外を除いて、先頭文字、末尾文字、および挿入文字は、アポストロフィで囲んでも囲まなくても、追加できます。

文字 "^" は、編集マスクに空白を挿入するために、数値フィールドおよび英数字フィールドの両方で使用します。

フィールドの長さ

編集マスクを割り当てるフィールドの長さに注意することが重要です。

- 編集マスクがフィールドより長い場合、予期しない結果が発生します。
- 編集マスクがフィールドより短い場合、フィールド出力は、編集マスクに指定された桁数に切り詰められます。

例：

ある英数字フィールドの長さが 12 文字で、出力されるフィールド値が "JOHNSON" であると仮定した場合、編集マスクによって以下のような結果が得られます。

編集マスク	出力
EM=X.X.X.X.X	J.O.H.N.S
EM=****XXXXXX****	****JOHNSO**

日付／時刻フィールドの編集マスク

日付フィールドの編集マスクには、"D"（日）、"M"（月）、および "Y"（年）の文字をさまざまな組み合せで指定できます。

時刻フィールドの編集マスクには、"H"（時）、"I"（分）、"S"（秒）、および "T"（10 分の 1 秒）の文字をさまざまな組み合せで指定できます。

日付フィールドおよび時刻フィールドの編集マスクとともに、日時システム変数も参照してください。

セパレータ文字の表示のカスタマイズ

Natural プログラムは世界中のビジネスアプリケーションで使用されます。数値データフィールドおよび日時を含むフィールドを I/O ステートメントで表示する場合には、地域的な慣習に応じて、特別な出力スタイルで表示するのが一般的です。 それぞれの表示方法は、プログラムが実行されるロケールの機能として、選択的に処理される代替のプログラムコーディングで認識される必要はありません。ただし、小数点文字および 「千桁単位セパレータ文字」 を指定する一連のランタイムパラメータとともに、同じプログラムイメージを使用して実行される必要があります。

以下では次のトピックについて説明します。

- 小数点文字
- ダイナミック千桁単位セパレータ

- 例

小数点文字

Natural パラメータ DC（小数点文字）を使用すると、編集マスク内で小数点記号（「基数点文字」 とも呼ぶ）を表すために使用されている文字の代わりに挿入される文字を指定できます。このパラメータを使用すると、Naturalプログラムまたはアプリケーションのユーザーは、任意（特定）の文字を選択して、数値データ項目の整数部と小数部を分けることができます。例えば、米国の店舗では小数点（.）を使用し、ヨーロッパの店舗ではコンマ（,）を使用することなどができます。

ダイナミック千桁単位セパレータ

大きい整数値の出力を構築する場合、一般的には整数の3桁ごとにセパレータが挿入されて、千桁ごとに値が分割されます。 このセパレータは、「千桁単位セパレータ」と呼ばれます。 例えば、米国では通常、このためにコンマを使用しますが（1,000,000 など）、ドイツではピリオド（1.000.000）、フランスでは空白（1 000 000）を使用します。

Natural編集マスクでは、「ダイナミック千桁単位セパレータ」にコンマ（またはピリオド）を使用して、ランタイム時に千桁単位セパレータ文字（THSEPCHパラメータで定義）が挿入される位置を示します。コンパイル時に、Naturalプロファイルパラメータ THSEP またはシステムコマンド COMPOPT のオプション THSEP を使用することで、コンマ（またはピリオド）をダイナミック千桁単位セパレータとして解釈可能または解釈不可にできます。

THSEP を OFF に設定した場合（デフォルト）、編集マスクで千桁単位セパレータとして使用されている文字はすべてリテラルとして処理され、ランタイム時にそのまま表示されます。 この設定は下位互換性を保持します。

THSEP を ON に設定した場合、編集マスク内のコンマ（またはピリオド）はすべてダイナミック千桁単位セパレータとして解釈されます。 一般的に、ダイナミック千桁単位セパレータはコンマですが、コンマがすでに小数点文字（DC）として使用されている場合には、ピリオドが千桁単位セパレータとして使用されます。

ランタイム時に、ダイナミック千桁単位セパレータは、THSEPCHパラメータの現在の値（千桁単位セパレータ文字）に置き換えられます。

例

パラメータ設定 DC='.' および THSEP=ON でカタログされている Natural プログラムで編集マスク（EM=ZZ,ZZZ,ZZ9.99）を使用します。

ランタイム時のパラメータ設定	表示内容
DC='.' および THSEPCH=','	1,234,567.89
DC=',' および THSEPCH='.'	1.234.567,89
DC=',' および THSEPCH='/'	1/234/567,89
DC=',' および THSEPCH=' '	1 234 567,89
DC=',' および THSEPCH='"'	1'234'567,89

編集マスクの例

編集マスクと編集マスクで作成される出力の例を以下に示します。

さらに、各編集マスクの省略表記法も示します。 省略形または長い表記のどちらでも使用できます。

編集マスク	省略形	出力 A	出力 B
EM=999.99	EM=9(3).9(2)	367.32	005.40
EM=ZZZZZ9	EM=Z(5)9(1)	0	579
EM=X^XXXXX	EM=X(1)^X(5)	B LUE	A 19379
EM=XXX...XX	EM=X(3)...X(2)	BLU...E	AAB...01
EM=MM.DD.YY	*	01.05.87	12.22.86
EM=HH.II.SS.T	**	08.54.12.7	14.32.54.3

* 日付システム変数を使用します。

** 時刻システム変数を使用します。

編集マスクの詳細については、『パラメータリファレンス』のセッションパラメータ EM を参照してください。

EM パラメータなしのプログラム例

```
** Example 'EDITMX01': Edit mask (using default edit masks)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
  2 SALARY (1:3)
  2 CITY
END-DEFINE
*
READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'
  DISPLAY 'N A M E'    NAME         /
          'OCCUPATION' JOB-TITLE
          'SALARY'     SALARY (1:3)
          'LOCATION'   CITY
  SKIP 1
END-READ
END
```

プログラム EDITMX01 の出力：

上記のプログラムの出力は、使用可能なデフォルトの編集マスクを示しています。

```
Page      1                                                 04-11-11  14:15:54

         N A M E            SALARY         LOCATION
       OCCUPATION
------------------------ ---------- --------------------

JONES                          46000 TULSA
MANAGER                        42300
                               39300

JONES                          50000 MOBILE
DIRECTOR                       46000
                               42700

JONES                          31000 MILWAUKEE
```

```
PROGRAMMER                     29400
                               27600
```

EM パラメータありのプログラム例

```
** Example 'EDITMX02': Edit mask (using EM)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 JOB-TITLE
  2 SALARY (1:3)
END-DEFINE
*
READ (3) VIEWEMP BY NAME STARTING FROM 'JONES'
 DISPLAY 'N A M E'    NAME         (EM=X^X^X^X^X^X^X^X^X^X^X^X^X^X^X^X) /
                       FIRST-NAME   (EM=...X(10)...)
         'OCCUPATION' JOB-TITLE    (EM=' ___ 'X(12))
         'SALARY'     SALARY (1:3) (EM=' USD 'ZZZ,999)
  SKIP 1
END-READ
END
```

プログラム EDITMX02 の出力：

前のプログラム（*EM パラメータなしのプログラム例*）の出力と比較して、EM 指定がフィールドの表示方法に与える影響を確認します。

```
Page      1                                                 04-11-11  14:15:54

           N A M E               OCCUPATION      SALARY
         FIRST-NAME
---------------------------- ---------------- -----------

J O N E S                     ___ MANAGER      USD  46,000
..VIRGINIA  ...                                USD  42,300
                                               USD  39,300

J O N E S                     ___ DIRECTOR     USD  50,000
..MARSHA    ...                                USD  46,000
                                               USD  42,700

J O N E S                     ___ PROGRAMMER   USD  31,000
```

```
..ROBERT     ...                                    USD  29,400
                                                    USD  27,600
```

編集マスクの他の例

次の例のプログラムを参照してください。

- ***EDITMX03 -** 編集マスク（英数字フィールドの異なる **EM**）*
- ***EDITMX04 -** 編集マスク（数値フィールドの異なる **EM**）*
- ***EDITMX05 -** 編集マスク（日付および時刻システム変数の **EM**）*

41 Unicode 編集マスク - EMU パラメータ

Unicode 編集マスクは、コードページ編集マスクと同様に使用できます。 違いは、編集マスクが必ず Unicode フォーマットで保存されることです。

そのため、ユーザーは別のコードページからの文字が混在した編集マスクを指定でき、インストールされているシステムコードページに関係なく、いつでも確実に正しい文字が表示されるようにすることができます。

編集マスクの一般的な使用法については、「*編集マスク - EM パラメータ*」を参照してください。

セッションパラメータ `EMU` の詳細については、『*パラメータリファレンス*』の「*EMU - Unicode 編集マスク*」を参照してください。

42 垂直表示

このchapterでは、DISPLAY と WRITE ステートメントの機能を組み合わせて、フィールドの値を垂直表示する方法について説明します。

次のトピックについて説明します。

垂直表示の作成

垂直表示の作成には 2 つの方法があります。

- DISPLAY ステートメントと WRITE ステートメントの組み合わせを使用できます。
- DISPLAY ステートメントの VERT オプションを使用できます。

DISPLAY と WRITE の組み合わせ

「*DISPLAY および WRITE ステートメント*」で説明しているように、DISPLAY ステートメントは、通常、デフォルトヘッダー付きの列にデータを表示しますが、WRITE ステートメントはヘッダーなしでデータを横に表示します。

2 つのステートメントの機能を組み合わせて、フィールド値を垂直表示することができます。

DISPLAY ステートメントは、同じレコードの個々のフィールドの値を、フィールドごとに 1 列に、ページをまたがって出力します。 各レコードのフィールドの値は、前のレコードの値の下に表示されます。

DISPLAY ステートメントの後に WRITE ステートメントを使用することによって、WRITE ステートメントで指定したテキストやフィールド値を、DISPLAY ステートメントで表示されるレコードの間に挿入できます。

以下のプログラムは、DISPLAY と WRITE の組み合わせを示しています。

```
** Example 'WRITEX04': WRITE (in combination with DISPLAY)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
  2 CITY
  2 DEPT
END-DEFINE
*
READ (3) VIEWEMP BY CITY STARTING FROM 'SAN FRANCISCO'
  DISPLAY NAME JOB-TITLE
```

```
  WRITE 22T 'DEPT:' DEPT
  SKIP 1
END-READ
END
```

プログラム WRITEX04 の出力：

```
Page      1                                                  04-11-11  14:15:55

        NAME                  CURRENT
                             POSITION
-------------------- -------------------------

KOLENCE              MANAGER
                     DEPT: TECH05

GOSDEN               ANALYST
                     DEPT: TECH10

WALLACE              SALES PERSON
                     DEPT: SALE20
```

タブ表記 - T*field

前の例において、フィールド DEPT の位置はタブ表記 nT で決定されます。この例の 20T の場合、画面の 20 桁目から表示が始まることを意味しています。

WRITE ステートメントで指定したフィールド値は、タブ表記 T*field を使用することによって、プログラムの最初の DISPLAY ステートメントで指定したフィールド値に自動的に揃えることができます。field は、前者のフィールドを揃える対象となる後者のフィールドの名前です。

以下のプログラムでは、WRITE ステートメントによる出力は、T*JOB-TITLE という表記を使用してフィールド JOB-TITLE に揃えられています。

```
** Example 'WRITEX05': WRITE  (in combination with DISPLAY)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
  2 DEPT
  2 CITY
END-DEFINE
*
READ (3) VIEWEMP BY CITY STARTING FROM 'SAN FRANCISCO'
```

```
  DISPLAY NAME JOB-TITLE
  WRITE T*JOB-TITLE 'DEPT:' DEPT
  SKIP 1
END-READ
END
```

プログラム WRITEX05 の出力：

```
Page      1                                                  04-11-11  14:15:55

        NAME                   CURRENT
                              POSITION
------------------- -------------------------

KOLENCE              MANAGER
                     DEPT: TECH05

GOSDEN               ANALYST
                     DEPT: TECH10

WALLACE              SALES PERSON
                     DEPT: SALE20
```

位置指定表記 *x*/*y*

DISPLAY および WRITE ステートメントを連続して使用し、WRITE ステートメントで複数行が出力される場合、*x*/*y*（数字スラッシュ数字）表記を使用して、表示する行および列を指定できます。 位置指定表記は、DISPLAY または WRITE ステートメントの次の要素を最後の出力の *x* 行下の *y* 桁目から開始します。

以下のプログラムは、この表記の使用法を示しています。

```
** Example 'WRITEX06': WRITE  (with n/n)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 MIDDLE-I
  2 ADDRESS-LINE (1:1)
  2 CITY
  2 ZIP
END-DEFINE
*
READ (3) VIEWEMP BY CITY STARTING FROM 'NEW YORK'
```

```
  DISPLAY 'NAME AND ADDRESS' NAME
  WRITE   1/5  FIRST-NAME
          1/30 MIDDLE-I
          2/5  ADDRESS-LINE (1:1)
          3/5  CITY
          3/30 ZIP /
END-READ
END
```

プログラム WRITEX06 の出力：

```
Page      1                                                 04-11-11  14:15:55

  NAME AND ADDRESS
--------------------

RUBIN
    SYLVIA                   L
    2003 SARAZEN PLACE
    NEW YORK                 10036

WALLACE
    MARY                     P
    12248 LAUREL GLADE C
    NEW YORK                 10036

KELLOGG
    HENRIETTA                S
    1001 JEFF RYAN DR.
    NEWARK                   19711
```

DISPLAY VERT ステートメント

Natural の標準の表示モードは水平方向です。

DISPLAY ステートメントの VERT 節オプションを使用すると、標準表示を上書きして、フィールドを垂直表示できます。

同じ DISPLAY ステートメントで使用できる HORIZ 節オプションは、標準の水平表示モードを再度有効にします。

垂直モードの列ヘッダーは、AS 節のさまざまなフォームで制御されます。 以下のプログラム例は、DISPLAY VERT ステートメントの使用法を示しています。

- AS 節のない DISPLAY VERT
- DISPLAY VERT AS CAPTIONED および HORIZ
- DISPLAY VERT AS '*text*'

- DISPLAY VERT AS '*text*' CAPTIONED
- タブ表記 P**field*

AS 節のない DISPLAY VERT

以下のプログラムでは AS 節を使用していません。つまり、列ヘッダーは出力されません。

```
** Example 'DISPLX09': DISPLAY  (without column title)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
END-DEFINE
*
READ (3) VIEWEMP BY CITY STARTING FROM 'NEW YORK'
  DISPLAY VERT NAME FIRST-NAME / CITY
  SKIP 2
END-READ
END
```

プログラム DISPLX09 の出力：

すべてのフィールド値が垂直方向に表示されることに注意してください。

```
Page      1                                                 04-11-11  14:15:54


RUBIN
SYLVIA

NEW YORK


WALLACE
MARY

NEW YORK


KELLOGG
HENRIETTA
```

```
NEWARK
```

DISPLAY VERT AS CAPTIONED および HORIZ

以下のプログラムには VERT および HORIZ 節が含まれており、一部の列の値が垂直方向に出力され、その他の値が水平方向に出力されます。さらに、AS CAPTIONEDによってデフォルトの列ヘッダーが表示されます。

```
** Example 'DISPLX10': DISPLAY (with VERT as CAPTIONED and HORIZ clause)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
  2 JOB-TITLE
  2 SALARY (1:1)
END-DEFINE
*
READ (3) VIEWEMP BY CITY STARTING FROM 'NEW YORK'
  DISPLAY VERT AS CAPTIONED NAME FIRST-NAME
          HORIZ JOB-TITLE SALARY (1:1)
  SKIP 1
END-READ
END
```

プログラム DISPLX10 の出力：

```
Page      1                                                 04-11-11  14:15:54

        NAME                  CURRENT            ANNUAL
     FIRST-NAME              POSITION            SALARY
-------------------- ------------------------- ----------

RUBIN                SECRETARY                      17000
SYLVIA

WALLACE              ANALYST                        38000
MARY
```

```
KELLOGG              DIRECTOR                        52000
HENRIETTA
```

DISPLAY VERT AS '*text*'

以下のプログラムには AS '*text*' 節が含まれており、指定した '*text*' が列ヘッダーとして表示されます。

Note: DISPLAY ステートメントのテキスト要素内のスラッシュ（/）は行送りを発生させます。

```
** Example 'DISPLX11': DISPLAY  (with VERT AS 'text' clause)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
  2 JOB-TITLE
  2 SALARY (1:1)
END-DEFINE
*
READ (3) VIEWEMP BY CITY STARTING FROM 'NEW YORK'
  DISPLAY VERT AS 'EMPLOYEES' NAME FIRST-NAME
          HORIZ JOB-TITLE SALARY (1:1)
  SKIP 1
END-READ
END
```

プログラム DISPLX11 の出力：

```
Page      1                                                 04-11-11  14:15:54

     EMPLOYEES               CURRENT            ANNUAL
                            POSITION            SALARY
-------------------- ------------------------- ----------

RUBIN                SECRETARY                      17000
SYLVIA

WALLACE              ANALYST                        38000
MARY
```

```
KELLOGG              DIRECTOR                        52000
HENRIETTA
```

DISPLAY VERT AS '*text*' CAPTIONED

AS '*text*' CAPTIONED 節によって指定したテキストが列ヘッダーとして表示され、出力する各行のフィールド値の直前にデフォルトの列ヘッダーが表示されます。

以下のプログラムには、AS '*text*' CAPTIONED 節が含まれています。

```
** Example 'DISPLX12': DISPLAY (with VERT AS 'text' CAPTIONED clause)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
  2 JOB-TITLE
  2 SALARY (1:1)
END-DEFINE
*
READ (3) VIEWEMP BY CITY STARTING FROM 'NEW YORK'
  DISPLAY VERT AS 'EMPLOYEES' CAPTIONED NAME FIRST-NAME
          HORIZ JOB-TITLE SALARY (1:1)
  SKIP 1
END-READ
END
```

プログラム DISPLX12 の出力：

この節によって、デフォルトの列ヘッダー（NAME および FIRST-NAME）がフィールド値の前に出力されます。

```
Page      1                                              04-11-11  14:15:54

           EMPLOYEES                    CURRENT           ANNUAL
                                       POSITION           SALARY
------------------------------ ------------------------- ----------

NAME RUBIN                     SECRETARY                    17000
FIRST-NAME SYLVIA

NAME WALLACE                   ANALYST                      38000
FIRST-NAME MARY
```

```
NAME KELLOGG                    DIRECTOR                        52000
FIRST-NAME HENRIETTA
```

タブ表記 P*field

DISPLAY VERT ステートメントと後続の WRITE ステートメントを組み合わせて使用する場合、WRITE ステートメントでタブ表記 P*field-name を使用して、フィールド位置を DISPLAY VERT ステートメントで指定した特定のフィールドの列および行の位置に揃えることができます。

以下のプログラムでは、フィールド SALARY および BONUS が同じ列に表示され、1 番目の各行に SALARY、2 番目の各行に BONUS が表示されます。 テキスト "***SALARY PLUS BONUS***" は SALARY に揃えられます。つまり、テキストは SALARY と同じ列の 1 番目の行に表示されます。一方、テキスト "(IN US DOLLARS)" は BONUS に揃えられるため、BONUS と同じ列の 2 番目の行に表示されます。

```
** Example 'WRITEX07': WRITE  (with P*field)
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 JOB-TITLE
  2 SALARY (1:1)
  2 BONUS  (1:1,1:1)
END-DEFINE
*
READ (3) VIEWEMP BY CITY STARTING FROM 'LOS ANGELES'
  DISPLAY NAME JOB-TITLE
          VERT AS 'INCOME' SALARY (1) BONUS (1,1)
  WRITE P*SALARY '***SALARY PLUS BONUS***'
        P*BONUS  '(IN US DOLLARS)'
  SKIP 1
END-READ
END
```

プログラム WRITEX07 の出力：

```
Page      1                                                  04-11-11  14:15:55

        NAME                  CURRENT            INCOME
                             POSITION
-------------------- ------------------------ ----------

SMITH                                                  0
                                                       0
                                              ***SALARY PLUS BONUS***
```

```
                                        (IN US DOLLARS)

POORE JR             SECRETARY                25000
                                                 0
                                        ***SALARY PLUS BONUS***
                                        (IN US DOLLARS)

PREPARATA            MANAGER                  46000
                                              9000
                                        ***SALARY PLUS BONUS***
                                        (IN US DOLLARS)
```

DISPLAY VERT と WRITE ステートメントの他の例

次の例のプログラムを参照してください。

- ***WRITEX10 - WRITE（nT、T*field、および P*field を使用）***

43 プログラミングのその他のポイント

このセクションでは、次のトピックについて説明します。

- ステートメント、プログラム、またはアプリケーションの終了
- 条件付き処理 - **IF** ステートメント
- ループ処理
- コントロールブレイク
- データ計算
- システム変数とシステム関数
- スタック
- 日付情報の処理
- テキスト表記
- ユーザーコメント
- 論理条件基準
- 演算割り当てのルール
- **3GL** プログラムからの **Natural** サブプログラムの呼び出し
- **Natural** プログラム内からのオペレーティングシステムコマンドの発行
- インターネットおよび **XML** アクセス用のステートメント

44 ステートメント、プログラム、またはアプリケーションの終了

このchapterでは、次のトピックについて説明します。

ステートメントの終了

ステートメントの終了を明示的に示すには、セミコロン（;）をステートメントとその次のステートメントの間に記述します。 これにより、プログラム構造をより明確にできますが、必須ではありません。

プログラムの終了

END ステートメントは、Natural のプログラム、関数、サブプログラム、外部サブルーチン、またはヘルプルーチンの終了を示すために使用します。

これらの各オブジェクトでは、最後のステートメントとして END ステートメントを使用する必要があります。

どのオブジェクトでも、END ステートメントを 1 つだけ使用します。

アプリケーションの終了

STOP ステートメントによる、アプリケーションの実行の終了

STOP ステートメントは、Natural アプリケーションの実行を終了するために使用します。 アプリケーション内の任意の場所で STOP ステートメントを実行すると、アプリケーション全体の実行が即座に停止されます。

TERMINATE ステートメントによる、アプリケーションの実行の終了

TERMINATE ステートメントを使用すると、Natural アプリケーションの実行が停止され、Natural セッションも終了します。

実行中の Natural アプリケーションの中断

Naturalアプリケーションの開発におけるテスト時には、実行中に無限ループなどで応答しなくなったNaturalアプリケーションを中断できる必要があります。 Naturalセッションは終了させる必要がないため、システムを中断する典型的なキーの組み合わせ（Windows の場合は Ctrl + Break、UNIX および OpenVMS の場合は Ctrl + C）を使用して、実行中の Natural アプリケーションを中断します。 Natural エラー NAT1016 が発生し、ランタイムエラー処理が実行されます。エラーは、`ON ERROR` 処理で制御できます。

実稼働環境では、プログラムの任意の位置でユーザーがアプリケーションを中断すると、そのアプリケーションは復帰できなくなるため、通常はこの機能を無効にする必要があります。

中断が可能かどうかは、Naturalプロファイルパラメータ `RTINT` によって決まります。 デフォルトでは、中断は認められていません。

このパラメータが `ON` に設定されている場合は、中断に対するオペレーティングシステムのキーの組み合わせ（Windows の場合は Ctrl + Break。UNIX の場合は、通常は Ctrl + C。ただし、`stty` コマンドで再設定できます。OpenVMS の場合は Ctrl + C）を使用して Natural アプリケーションを中断できます。

Natural はこの中断要求を捕捉すると、以下の実行可能な選択肢をユーザーに提示します。

- NAT1016 エラーを発生させ、標準のエラー処理を実行します。
- 中断をキャンセルして、アプリケーションの処理を続行します。

これらの選択肢は、中断シグナルの捕捉後に開くウィンドウに表示されます。

Note: Naturalアプリケーションは、Naturalアプリケーションまたはアプリケーションを開始した Natural スタジオに入力フォーカスがある場合にのみ、中断できます。

45 条件付き処理 - IF ステートメント

IF ステートメントで論理条件を定義します。IF ステートメントに付随するステートメントの実行は、その条件に依存します。

このchapterでは、次のトピックについて説明します。

IF ステートメントの構造

IF ステートメントには、以下の 3 つの構成要素があります。

IF	IF 節には、満たす必要のある論理条件を指定します。
THEN	THEN節には、この条件を満たしている場合に実行するステートメント（複数可）を指定します。
ELSE	（任意の）ELSE節では、この条件を満たしていない場合に実行するステートメント（複数可）を指定できます。

したがって、IF ステートメントの一般的な形式は以下のようになります。

```
IF condition
   THEN execute statement(s)
   ELSE execute other statement(s)
END-IF
```

Note: IF条件を満たしていないときにのみ特定の処理を実行する場合、THEN IGNORE節を指定できます。IGNOREステートメントにより、条件を満たすとIF条件は無視されます。

例 **1**：

```
** Example 'IFX01': IF
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 BIRTH
  2 CITY
  2 SALARY (1:1)
END-DEFINE
*
LIMIT 7
READ MYVIEW BY CITY STARTING FROM 'C'
 IF SALARY (1) LT 40000 THEN
    WRITE NOTITLE '*****' NAME 30X 'SALARY LT 40000'
  ELSE
    DISPLAY NAME BIRTH (EM=YYYY-MM-DD) SALARY (1)
  END-IF
```

```
END-READ
END
```

上記のプログラムの IF ステートメントブロックでは、以下の条件処理が実行されます。

- （IF）給与が 40000 未満の場合、（THEN）WRITE ステートメントが実行されます。
- それ以外の場合（ELSE）、つまり、給与が 40000 以上の場合は DISPLAY ステートメントが実行されます。

プログラム IFX01 の出力：

```
        NAME            DATE      ANNUAL
                         OF       SALARY
                       BIRTH
-------------------- ---------- ----------

***** KEEN                                          SALARY LT 40000
***** FORRESTER                                     SALARY LT 40000
***** JONES                                         SALARY LT 40000
***** MELKANOFF                                     SALARY LT 40000
DAVENPORT            1948-12-25      42000
GEORGES              1949-10-26     182800
***** FULLERTON                                     SALARY LT 40000
```

例 **2**：

```
** Example 'IFX03': IF
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 BONUS  (1,1)
  2 SALARY (1)
*
1 #INCOME  (N9)
1 #TEXT    (A26)
END-DEFINE
*
WRITE TITLE '-- DISTRIBUTION OF CATALOGS I AND II --' /
*
READ (3) EMPLOY-VIEW BY CITY = 'SAN FRANSISCO'
  COMPUTE #INCOME = BONUS(1,1) + SALARY(1)
  /*
 IF #INCOME > 40000
    MOVE 'CATALOGS I AND II' TO #TEXT
  ELSE
    MOVE 'CATALOG I'         TO #TEXT
```

```
  END-IF
  /*
  DISPLAY NAME 5X 'SALARY' SALARY(1) / BONUS(1,1)
  WRITE T*SALARY '-'(10) /
        16X 'INCOME:' T*SALARY #INCOME 3X #TEXT /
        16X '='(19)
  SKIP 1
END-READ
END
```

プログラム IFX03 の出力：

```
                    -- DISTRIBUTION OF CATALOGS I AND II --

        NAME               SALARY
                           BONUS
--------------------     ----------

COLVILLE JR                   56000
                                  0
                           ----------
                INCOME:       56000   CATALOGS I AND II
                ====================

RICHMOND                       9150
                                  0
                           ----------
                INCOME:        9150   CATALOG I
                ====================

MONKTON                       13500
                                600
                           ----------
                INCOME:       14100   CATALOG I
                ====================
```

IF ステートメントのネスト

IF ステートメントは、複数のネスト構造にして使用できます。例えば、THEN 節の実行を、THEN 節に指定した別の IF ステートメントに依存させることができます。

例：

```
** Example 'IFX02': IF (two IF statements nested)
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 SALARY (1:1)
  2 BIRTH
  2 PERSONNEL-ID
1 MYVIEW2 VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
*
1 #BIRTH   (D)
END-DEFINE
*
MOVE EDITED '19450101' TO #BIRTH (EM=YYYYMMDD)
*
LIMIT 20
FND1. FIND MYVIEW WITH CITY = 'BOSTON'
                  SORTED BY NAME
 IF SALARY (1) LESS THAN 20000
    WRITE NOTITLE '*****' NAME 30X 'SALARY LT 20000'
  ELSE
    IF BIRTH GT #BIRTH
      FIND MYVIEW2 WITH PERSONNEL-ID = PERSONNEL-ID (FND1.)
        DISPLAY (IS=ON) NAME BIRTH (EM=YYYY-MM-DD)
                        SALARY (1) MAKE (AL=8 IS=OFF)
      END-FIND
    END-IF
  END-IF
  SKIP 1
END-FIND
END
```

プログラム IFX02 の出力：

```
        NAME            DATE      ANNUAL     MAKE
                         OF       SALARY
                       BIRTH
-------------------- ---------- ---------- --------

***** COHEN                                          SALARY LT 20000

CREMER               1972-12-14      20000 FORD

***** FLEMING                                        SALARY LT 20000

PERREAULT            1950-05-12      30500 CHRYSLER
```

```
***** SHAW                                                SALARY LT 20000

STANWOOD             1946-09-08      31000 CHRYSLER
                                           FORD
```

46 ループ処理

記述した条件が満たされるまで、または一定の条件が有効である限り、処理ループはグループ化されたステートメントを繰り返し実行します。

このchapterでは、次のトピックについて説明します。

処理ループの使用

処理ループはデータベースループと非データベースループに分類できます。

- データベース処理ループ
  READ、FIND、または HISTOGRAM の各ステートメントの結果としてデータベースから取得したデータを処理するために、Naturalによって自動的に作成されます。これらのステートメントの詳細については、「*Adabas データベースのデータへのアクセス*」を参照してください。
- 非データベース処理ループ
  ステートメント REPEAT、FOR、CALL FILE、CALL LOOP、SORT、および READ WORK FILE によって開始されます。

複数の処理ループを同時にアクティブにできます。 ループは、アクティブな（開いている）他のループ内に埋め込んだり、ネストしたりできます。

処理ループは、対応する END-... ステートメント（END-REPEAT、END-FOR など）で明示的に閉じる必要があります。

オペレーティングシステムのソートプログラムを呼び出す SORT ステートメントは、すべてのアクティブな処理ループを閉じて、新しい処理ループを開始します。

データベースループの制限

以下では次のトピックについて説明します。

- 有効なデータベースループの制限方法
- LT セッションパラメータ
- LIMIT ステートメント
- リミット表記

- 制限設定の優先順位

有効なデータベースループの制限方法

ステートメント READ、FIND、または HISTOGRAM で開始した処理ループの繰り返し回数を制限するには、以下の 3 つの方法があります。

- セッションパラメータ LT を使用します。
- LIMIT ステートメントを使用します。
- または、READ/FIND/HISTOGRAM ステートメント自体のリミット表記を使用します。

LT セッションパラメータ

システムコマンド GLOBALS には、セッションパラメータ LT を指定できます。このパラメータを使用して、データベース処理ループで読み込むレコード件数を制限します。

例：

```
GLOBALS LT=100
```

この制限は、セッション全体のすべての READ、FIND、および HISTOGRAM の各ステートメントに適用されます。

LIMIT ステートメント

プログラムでは、LIMIT ステートメントを使用して、データベース処理ループで読み込むレコード件数を制限できます。

例：

```
LIMIT 100
```

別の LIMIT ステートメントまたはリミット表記によって上書きされるまで、LIMIT ステートメントはプログラムの残りの部分に適用されます。

リミット表記

READ、FIND、または HISTOGRAM の各ステートメント自体で、読み込むレコード件数をステートメント名の直後のカッコ内に指定できます。

例：

```
READ (10) VIEWXYZ BY NAME
```

このリミット表記は、その他の有効なあらゆる制限を上書きします。ただし、リミット表記が適用されるのは、その表記が指定されているステートメントのみです。

制限設定の優先順位

LT パラメータに設定されている制限が、LIMIT ステートメントに指定されている制限またはリミット表記よりも小さい場合、LT の制限はこれらの他の制限より優先されます。

非データベースループの制限 - REPEAT ステートメント

非データベース処理ループは、論理条件基準、または他に指定された制限条件に基づいて開始および終了します。

ここでは、非データベースループステートメントとして、REPEAT ステートメントについて説明します。

REPEAT ステートメントを使用して、繰り返し実行する1つ以上のステートメントを指定します。さらに、ある条件に一致するまで、または、ある条件に一致している間にのみステートメントが実行されるように、論理条件を指定できます。 この目的のためには、UNTIL 節または WHILE 節を使用します。

論理条件を指定する場合、以下のようにループは制御されます。

- UNTIL 節を使用すると、論理条件が満たされるまで REPEAT ループが続けられます。
- WHILE 節を使用すると、論理条件が真である限り REPEAT ループが続けられます。

論理条件を指定しない場合、以下のステートメントのいずれかを使用して、REPEAT ループを抜ける必要があります。

- ESCAPE：処理ループの実行を終了し、ループの外側の処理を継続します（下記参照）。
- STOP：Natural アプリケーション全体の実行を停止します。
- TERMINATE：Natural アプリケーションの実行を停止し、Natural セッションも終了します。

REPEAT ステートメントの例

```
** Example 'REPEAX01': REPEAT
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 SALARY (1:1)
*
1 #PAY1    (N8)
END-DEFINE
*
READ (5) MYVIEW BY NAME WHERE SALARY (1) = 30000 THRU 39999
  MOVE SALARY (1) TO #PAY1
  /*
  REPEAT WHILE #PAY1 LT 40000
    MULTIPLY #PAY1 BY 1.1
    DISPLAY NAME (IS=ON) SALARY (1)(IS=ON) #PAY1
  END-REPEAT
  /*
  SKIP 1
END-READ
END
```

プログラム REPEAX01 の出力：

```
Page      1                                                 04-11-11  14:15:54

        NAME           ANNUAL     #PAY1
                       SALARY
-------------------- ---------- ---------

ADKINSON                  34500     37950
                                    41745

                          33500     36850
                                    40535

                          36000     39600
                                    43560

AFANASSIEV                37000     40700
```

```
ALEXANDER                 34500      37950
                                     41745
```

処理ループの終了 - ESCAPE ステートメント

ESCAPE ステートメントは、論理条件に基づいて処理ループの実行を終了するために使用します。

ESCAPE ステートメントは、IF 条件ステートメントグループのループ内、およびブレイク処理ステートメントグループ（AT END OF DATA、AT END OF PAGE、AT BREAK）のループ内で指定できます。また、非データベースループの基本論理条件を実装している独立操作可能なステートメントとして指定できます。

ESCAPE ステートメントには、ESCAPE ステートメントで処理ループを抜けた後にどこから処理を継続するのかを指定するオプション TOP と BOTTOM が用意されています。

- ESCAPE TOP は、処理ループの先頭から処理を継続するために使用します。
- ESCAPE BOTTOMは、処理ループの後の最初のステートメントから処理を継続するために使用します。

同じ処理ループ内に複数の ESCAPE ステートメントを指定できます。

ESCAPE ステートメントの詳細と例については、『ステートメント』ドキュメントを参照してください。

ループ内のループ

別のデータベースステートメントによって開始されたデータベース処理ループ内にデータベースステートメントを指定できます。 データベースループ開始ステートメントがこの方法で埋め込まれていると、ループの 「階層」 が作成され、階層ごとに、選択条件を満たす各レコードが処理されます。

複数のレベルのループを埋め込むことができます。例えば、非データベースループをデータベースループ内にネストできます。 また、データベースループを非データベースループ内にネストすることもできます。 データベースループおよび非データベースループは、条件付きステートメントグループ内でネストできます。

FIND ステートメントのネストの例

以下のプログラムは、2 つのループ階層を示しています。1 つの FIND ループ内に別の FIND ループがネスト、つまり埋め込まれています。

```
** Example 'FINDX06': FIND (two FIND statements nested)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 PERSONNEL-ID
1 VEH-VIEW VIEW OF VEHICLES
  2 MAKE
  2 PERSONNEL-ID
END-DEFINE
*
FND1. FIND EMPLOY-VIEW WITH CITY = 'NEW YORK' OR = 'BEVERLEY HILLS'
  FIND (1) VEH-VIEW WITH PERSONNEL-ID = PERSONNEL-ID (FND1.)
    DISPLAY NOTITLE NAME CITY MAKE
  END-FIND
END-FIND
END
```

上記のプログラムでは、複数のファイルからデータを選択しています。 外側の FIND ループでは、EMPLOYEES ファイルから、ニューヨークまたはビバリーヒルズに住んでいるすべての個人を選択しています。 内側の FIND ループでは、外側のループで選択されたレコードごとに、VEHICLES ファイルからその個人の車のデータを選択しています。

プログラム FINDX06 の出力：

```
        NAME                 CITY                 MAKE
-------------------- -------------------- --------------------

RUBIN                NEW YORK             FORD
OLLE                 BEVERLEY HILLS       GENERAL MOTORS
WALLACE              NEW YORK             MAZDA
```

```
JONES                BEVERLEY HILLS       FORD
SPEISER              BEVERLEY HILLS       GENERAL MOTORS
```

プログラム内のステートメント参照

ステートメント参照表記は、以下の目的で使用します。

- 特定の範囲のデータに対する処理を指定するために、プログラム内で前に処理したステートメントを参照するため。
- Natural のデフォルトの参照設定を上書きするため。
- コードをわかりやすくするため。

処理ループの開始、またはデータベースのデータ要素へのアクセスを発生させる、あらゆる Natural ステートメントを参照できます。 例えば、以下のステートメントを参照できます。

- READ
- FIND
- HISTOGRAM
- SORT
- REPEAT
- FOR

プログラムで複数の処理ループを使用するときは、参照表記を使用して、データベースフィールドに最初にアクセスしたステートメントを参照することにより、処理対象のデータベースフィールドを一意に指定します。

このような方法でフィールドを参照できるかどうかは、『*ステートメント*』ドキュメントに記載されている、対応するステートメントの説明の「*オペランド定義テーブル*」の"ステートメント参照"列を参照してください。「ユーザー定義変数」の「表記(*r*)を使用したデータベースフィールドの参照」を参照してください。

また、参照表記はいくつかのステートメントで指定できます。 次に例を示します。

- AT START OF DATA
- AT END OF DATA
- AT BREAK
- ESCAPE BOTTOM

参照表記を使用しないと、AT START OF DATA、AT END OF DATA、または AT BREAK ステートメントは、*最も外側のアクティブな* READ、FIND、HISTOGRAM、SORT、または READ WORK FILE の各

ループに関連付けられます。 参照表記を使用すると、別のアクティブな処理ループに関連付けることができます。

参照表記を ESCAPE BOTTOM ステートメントに指定すると、参照表記で指定した処理ループの後の最初のステートメントから処理が継続されます。

ステートメント参照表記は、*ステートメント参照ラベル*形式または*ソースコード行番号*形式で指定できます。

- ステートメント参照ラベル
  ステートメント参照ラベルはいくつかの文字で構成され、最後には必ずピリオド（.）を使用します。 ピリオドによって、そのエントリはラベルとして識別されます。

  ステートメントが含まれる行の先頭にラベルを指定することにより、参照されるステートメントをラベルでマークします。 次に例を示します。

```
0030 ...
0040 READ1. READ VIEWXYZ BY NAME
0050 ...
```

  マークされたステートメントを参照するステートメントでは、ステートメントの構文図に示されている位置に、ラベルをカッコで囲んで指定します（『ステートメント』ドキュメントを参照）。 次に例を示します。

```
AT BREAK (READ1.) OF NAME
```

- ソースコード行番号
  ソースコード行番号を使用して参照する場合、行番号は、カッコで囲んだ4桁の数字（先行ゼロは省略不可）で指定する必要があります。 次に例を示します。

```
AT BREAK (0040) OF NAME
```

  ステートメント内でラベル／行番号を使用して特定のフィールドを前のステートメントに関連付ける場合、ラベル／行番号は、フィールド名の後にカッコで囲んで指定します。 次に例を示します。

```
DISPLAY NAME (READ1.) JOB-TITLE (READ1.) MAKE MODEL
```

行番号とラベルは区別なく使用できます。

「ユーザー定義変数」の「表記 (*r*) を使用したデータベースフィールドの参照」を参照してください。

行番号を使用した参照の例

以下のプログラムでは、参照にソースコード行番号（カッコで囲んだ4桁の数字）を使用しています。

この例では、参照先のステートメントへのすべての参照において、デフォルトで行番号が使用されています。

```
0010 ** Example 'LABELX01': Labels for READ and FIND loops (line numbers)
0020 ************************************************************************
0030 DEFINE DATA LOCAL
0040 1 MYVIEW1 VIEW OF EMPLOYEES
0050   2 NAME
0060  2 FIRST-NAME
0070  2 PERSONNEL-ID
0080 1 MYVIEW2 VIEW OF VEHICLES
0090  2 PERSONNEL-ID
0100  2 MAKE
0110 END-DEFINE
0120 *
0130 LIMIT 15
0140 READ MYVIEW1 BY NAME STARTING FROM 'JONES'
0150  FIND MYVIEW2 WITH PERSONNEL-ID = PERSONNEL-ID (0140)
0160    IF NO RECORDS FOUND
0170      MOVE '***NO CAR***' TO MAKE
0180    END-NOREC
0190    DISPLAY NOTITLE NAME       (0140) (IS=ON)
0200                    FIRST-NAME (0140) (IS=ON)
0210                    MAKE       (0150)
0220  END-FIND /* (0150)
```

```
0230 END-READ   /* (0140)
0240 END
```

ステートメント参照ラベルを使用した参照の例

以下の例は、ステートメント参照ラベルの使用方法を示しています。

行番号の代わりにラベルが参照に使用されている点以外は、前述のプログラムと同一です。

```
** Example 'LABELX02': Labels for READ and FIND loops (user labels)
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW1 VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 PERSONNEL-ID
1 MYVIEW2 VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
END-DEFINE
*
LIMIT 15
RD. READ MYVIEW1 BY NAME STARTING FROM 'JONES'
  FD. FIND MYVIEW2 WITH PERSONNEL-ID = PERSONNEL-ID (RD.)
    IF NO RECORDS FOUND
      MOVE '***NO CAR***' TO MAKE
    END-NOREC
    DISPLAY NOTITLE NAME       (RD.) (IS=ON)
                    FIRST-NAME (RD.) (IS=ON)
                    MAKE       (RD.)
  END-FIND /* (RD.)
END-READ   /* (RD.)
END
```

どちらのプログラムでも、以下の出力が生成されます。

```
        NAME              FIRST-NAME              MAKE
-------------------- -------------------- --------------------

JONES                 VIRGINIA             CHRYSLER
                      MARSHA               CHRYSLER
                                           CHRYSLER
                      ROBERT               GENERAL MOTORS
                      LILLY                FORD
                                           MG
                      EDWARD               GENERAL MOTORS
```

```
                    LAUREL              GENERAL MOTORS
                    KEVIN               DATSUN
                    GREGORY             FORD
JOPER               MANFRED             ***NO CAR***
JOUSSELIN           DANIEL              RENAULT
JUBE                GABRIEL             ***NO CAR***
JUNG                ERNST               ***NO CAR***
JUNKIN              JEREMY              ***NO CAR***
KAISER              REINER              ***NO CAR***
KANT                HEIKE               ***NO CAR***
```

47 コントロールブレイク

このchapterでは、ステートメントの実行をコントロールブレイクに依存させる方法、およびNatural システム関数の評価にコントロールブレイクを使用する方法について説明します。

次のトピックについて説明します。

コントロールブレイクの使用

コントロールブレイクは、コントロールフィールドの値が変わると発生します。

ステートメントの実行をコントロールブレイクに依存させることができます。

また、Natural システム関数の評価にコントロールブレイクを使用することもできます。

システム関数については、「*システム変数とシステム関数*」で説明します。 有効なシステム関数の詳細については、『システム*関数*』ドキュメントを参照してください。

AT BREAK ステートメント

AT BREAK ステートメントでは、コントロールブレイクが起こるたび、つまり、AT BREAK ステートメントに指定したコントロールフィールドの値が変わるたびに実行する処理を指定します。コントロールフィールドとして、データベースフィールドまたはユーザー定義変数を使用できます。

以下では次のトピックについて説明します。

- **データベースフィールドに基づくコントロールブレイク**
- **ユーザー定義変数に基づくコントロールブレイク**
- **複数のコントロールブレイクレベル**

データベースフィールドに基づくコントロールブレイク

AT BREAK ステートメントにコントロールフィールドとして指定されるフィールドは、通常はデータベースフィールドです。

例：

```
...
AT BREAK OF DEPT
  statements
END-BREAK
...
```

この例では、コントロールフィールドはデータベースフィールド DEPT です。このフィールドの値が変わると（例："SALE01" から "SALE02" へ）、AT BREAK ステートメントに指定した *statements* が実行されます。

フィールド全体ではなく、フィールドの一部のみをコントロールフィールドとして使用することもできます。 */n/* 表記を使用して、フィールドの先頭の *n* 桁のみを使用して値の変更をチェックするように指定できます。

例：

```
...
AT BREAK OF DEPT /4/
  statements
END-BREAK
...
```

この例では、指定した *statements* は、フィールド DEPT の先頭 4 桁の値が変わった場合（例："SALE" から "TECH" へ）にのみ実行されます。ただし、フィールド値が "SALE01" から "SALE02" に変わっても、この変更は無視され、AT BREAK 処理は実行されません。

例：

```
** Example 'ATBREX01': AT BREAK OF (with database field)
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 COUNTRY
  2 JOB-TITLE
  2 SALARY (1:1)
END-DEFINE
*
READ (5) MYVIEW BY CITY WHERE COUNTRY = 'USA'
  DISPLAY CITY (AL=9) NAME 'POSITION' JOB-TITLE 'SALARY' SALARY(1)
  /*
  AT BREAK OF CITY
    WRITE /   OLD(CITY) (EM=X^X^X^X^X^X^X^X^X^X^X^X^)
```

```
          5X 'AVERAGE:' T*SALARY AVER(SALARY(1)) //
              COUNT(SALARY(1)) 'RECORDS FOUND' /
  END-BREAK
  /*
  AT END OF DATA
    WRITE 'TOTAL (ALL RECORDS):' T*SALARY(1) TOTAL(SALARY(1))
  END-ENDDATA
END-READ
END
```

上記のプログラムでは、フィールド CITY の値が変わるたびに、最初の WRITE ステートメントが実行されます。

AT BREAK ステートメントでは、Natural システム関数 OLD、AVER、および COUNT が評価され、その結果が WRITE ステートメントで出力されます。

AT END OF DATA ステートメントでは、Natural システム関数 TOTAL が評価されます。

プログラム ATBREX01 の出力：

```
Page      1                                                 04-12-14  14:07:26

  CITY            NAME                 POSITION            SALARY
--------- -------------------- ------------------------ ----------

AIKEN     SENKO                PROGRAMMER                    31500

A I K E N                  AVERAGE:                         31500

       1 RECORDS FOUND

ALBUQUERQ HAMMOND              SECRETARY                     22000
ALBUQUERQ ROLLING              MANAGER                       34000
ALBUQUERQ FREEMAN              MANAGER                       34000
ALBUQUERQ LINCOLN              ANALYST                       41000

A L B U Q U E R Q U E      AVERAGE:                         32750

       4 RECORDS FOUND
```

```
TOTAL (ALL RECORDS):                                          162500
```

ユーザー定義変数に基づくコントロールブレイク

ユーザー定義変数を AT BREAK ステートメントのコントロールフィールドとして使用することもできます。

以下のプログラムでは、ユーザー定義変数 #LOCATION が、コントロールフィールドとして使用されています。

```
** Example 'ATBREX02': AT BREAK OF  (with user-defined variable and
**                      in conjunction with BEFORE BREAK PROCESSING)
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 CITY
  2 COUNTRY
  2 JOB-TITLE
  2 SALARY (1:1)
*
1 #LOCATION (A20)
END-DEFINE
*
READ (5) MYVIEW BY CITY WHERE COUNTRY = 'USA'
  BEFORE BREAK PROCESSING
    COMPRESS CITY 'USA' INTO #LOCATION
  END-BEFORE
  DISPLAY #LOCATION 'POSITION' JOB-TITLE 'SALARY' SALARY (1)
  /*
  AT BREAK OF #LOCATION
    SKIP 1
  END-BREAK
END-READ
END
```

プログラム ATBREX02 の出力：

```
Page      1                                                 04-12-14  14:08:36

     #LOCATION               POSITION            SALARY
-------------------- ------------------------- ----------

AIKEN USA            PROGRAMMER                     31500

ALBUQUERQUE USA      SECRETARY                      22000
ALBUQUERQUE USA      MANAGER                        34000
```

```
ALBUQUERQUE USA      MANAGER                          34000
ALBUQUERQUE USA      ANALYST                          41000
```

複数のコントロールブレイクレベル

上記の説明のとおり、*/n/* 表記を使用すると、フィールドの一部分をコントロールブレイクのためにチェックできます。 フィールド全体をあるブレイクのコントロールフィールドとして使用し、同じフィールドの一部を別のブレイクのコントロールフィールドとして使用することにより、複数の AT BREAK ステートメントを組み合わせることができます。

このような場合、下位レベル（フィールド全体）のブレイクは、上位レベル（フィールドの一部）のブレイクの前に指定する必要があります。つまり、最初の AT BREAK ステートメントでフィールド全体をコントロールフィールドとして指定し、2番目のステートメントでフィールドの一部をコントロールフィールドとして指定する必要があります。

以下のプログラム例は、フィールド DEPT とその最初の 4 桁（DEPT /4/）を使用してこれを説明したものです。

```
** Example 'ATBREX03': AT BREAK OF (two statements in combination)
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
  2 DEPT
  2 SALARY    (1:1)
  2 CURR-CODE (1:1)
END-DEFINE
*
READ MYVIEW BY DEPT STARTING FROM 'SALE40' ENDING AT 'TECH10'
     WHERE SALARY(1) GT 47000 AND CURR-CODE(1) = 'USD'
  /*
  AT BREAK OF DEPT
    WRITE '*** LOWEST BREAK LEVEL ***' /
  END-BREAK
  AT BREAK OF DEPT /4/
    WRITE '*** HIGHEST BREAK LEVEL ***'
  END-BREAK
  /*
  DISPLAY DEPT NAME 'POSITION' JOB-TITLE
END-READ
END
```

プログラム ATBREX03 の出力：

```
Page      1                                                  04-12-14  14:09:20

DEPARTMENT         NAME                 POSITION
   CODE
---------- -------------------- -------------------------

TECH05     HERZOG               MANAGER
TECH05     LAWLER               MANAGER
TECH05     MEYER                MANAGER
*** LOWEST BREAK LEVEL ***

TECH10     DEKKER               DBA
*** LOWEST BREAK LEVEL ***

*** HIGHEST BREAK LEVEL ***
```

以下のプログラムでは、フィールド DEPT の値が変わるたびに 1 行の空白行が出力されます。また、DEPT の先頭 4 桁の値が変わるたびにシステム関数 COUNT が評価され、レコード件数が算出されます。

```
** Example 'ATBREX04': AT BREAK OF (two statements in combination)
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 DEPT
  2 REDEFINE DEPT
    3 #GENDEP (A4)
  2 NAME
  2 SALARY    (1)
END-DEFINE
*
WRITE TITLE '** PERSONS WITH SALARY > 30000, SORTED BY DEPARTMENT **' /
LIMIT 9
READ MYVIEW BY DEPT FROM 'A' WHERE SALARY(1) > 30000
  DISPLAY 'DEPT' DEPT NAME 'SALARY' SALARY(1)
  /*
  AT BREAK OF DEPT
    SKIP 1
  END-BREAK
  AT BREAK OF DEPT /4/
    WRITE COUNT(SALARY(1)) 'RECORDS FOUND IN:' OLD(#GENDEP) /
  END-BREAK
END-READ
END
```

プログラム ATBREX04 の出力：

```
           ** PERSONS WITH SALARY > 30000, SORTED BY DEPARTMENT **

 DEPT          NAME            SALARY
------ -------------------- ----------

ADMA01 JENSEN                   180000
ADMA01 PETERSEN                 105000
ADMA01 MORTENSEN                320000
ADMA01 MADSEN                   149000
ADMA01 BUHL                     642000

ADMA02 HERMANSEN                391500
ADMA02 PLOUG                    162900
ADMA02 HANSEN                   234000

       8 RECORDS FOUND IN: ADMA

COMP01 HEURTEBISE               168800

       1 RECORDS FOUND IN: COMP
```

自動ブレイク処理

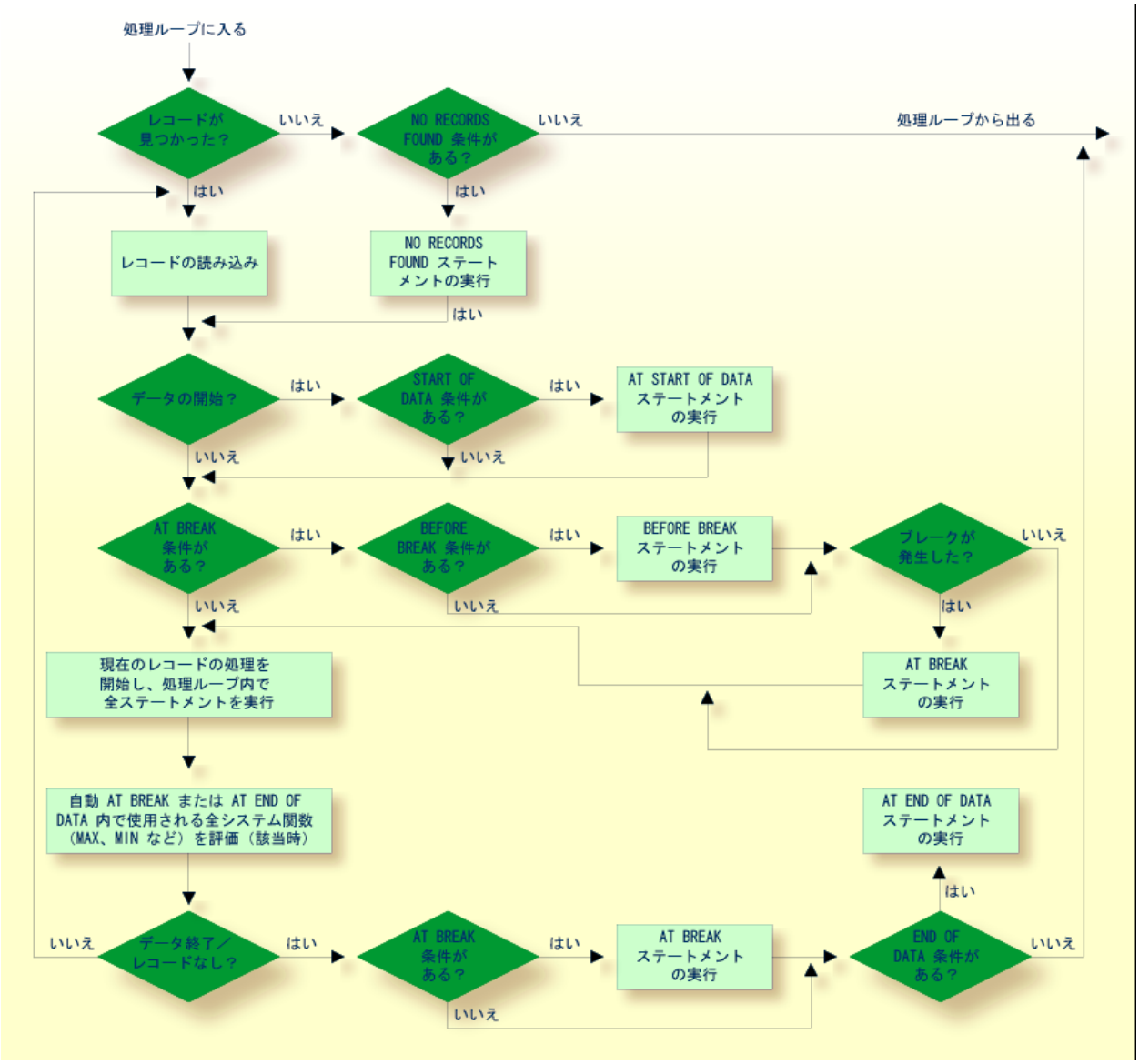
自動ブレイク処理は、AT BREAK ステートメントが含まれている処理ループで実行されます。 これは以下のステートメントに適用されます。

- FIND
- READ
- HISTOGRAM
- SORT
- READ WORK FILE

AT BREAK ステートメントに指定されているコントロールフィールドの値は、WITH 節および WHERE 節の選択条件を満たしているレコードに対してのみチェックされます。

Natural システム関数（AVER、MAX、MIN など）は、処理ループ内のすべてのステートメントを実行した後に、レコードごとに評価されます。 システム関数は、WHERE 条件で拒否されたレコードに対しては評価されません。

下の図は、自動ブレイク処理のフローロジックを示しています。



AT BREAK ステートメントとシステム関数の例

以下の例は、AT BREAK ステートメントにおける Natural システム関数 OLD、MIN、AVER、MAX、SUM、COUNT の使用方法（および AT END OF DATA ステートメントにおけるシステム関数 TOTAL の使用方法）を示しています。

```
** Example 'ATBREX05': AT BREAK OF (with system functions)
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 SALARY    (1:1)
  2 CURR-CODE (1:1)
END-DEFINE
*
LIMIT 3
READ MYVIEW BY CITY = 'SALT LAKE CITY'
  DISPLAY NOTITLE CITY NAME 'SALARY' SALARY(1) 'CURRENCY' CURR-CODE(1)
  /*
  AT BREAK OF CITY
    WRITE /  OLD(CITY) (EM=X^X^X^X^X^X^X^X^X^X^X^X^X^X^X^X^X^X)
      31T ' - MINIMUM:' MIN(SALARY(1))  CURR-CODE(1) /
      31T ' - AVERAGE:' AVER(SALARY(1)) CURR-CODE(1) /
      31T ' - MAXIMUM:' MAX(SALARY(1))  CURR-CODE(1) /
      31T ' -     SUM:' SUM(SALARY(1))  CURR-CODE(1) /
      33T COUNT(SALARY(1)) 'RECORDS FOUND' /
  END-BREAK
  /*
  AT END OF DATA
    WRITE 22T 'TOTAL (ALL RECORDS):'
              T*SALARY TOTAL(SALARY(1)) CURR-CODE(1)
  END-ENDDATA
END-READ
END
```

プログラム ATBREX05 の出力：

```
        CITY                   NAME            SALARY   CURRENCY
-------------------- -------------------- ---------- --------

SALT LAKE CITY       ANDERSON                  50000 USD
SALT LAKE CITY       SAMUELSON                 24000 USD

S A L T   L A K E   C I T Y    - MINIMUM:      24000 USD
                               - AVERAGE:      37000 USD
```

```
                                - MAXIMUM:      50000 USD
                                -     SUM:      74000 USD
                                        2 RECORDS FOUND

SAN DIEGO            GEE                        60000 USD

S A N   D I E G O            - MINIMUM:      60000 USD
                             - AVERAGE:      60000 USD
                             - MAXIMUM:      60000 USD
                             -     SUM:      60000 USD
                                     1 RECORDS FOUND

                     TOTAL (ALL RECORDS):     134000 USD
```

AT BREAK ステートメントの他の例

次の例のプログラムを参照してください。

- *ATBREX06 - AT BREAK OF（NMIN、NAVER、NCOUNT を MIN、AVER、COUNT と比較）*

BEFORE BREAK PROCESSING ステートメント

BEFORE BREAK PROCESSING ステートメントを使用すると、コントロールブレイクの直前、つまり、コントロールフィールドの値のチェック前、AT BREAK ブロックに指定したステートメントの実行前、および、あらゆるNaturalシステム関数の実行前に、実行するステートメントを指定できます。

BEFORE BREAK PROCESSING ステートメントの例

```
** Example 'BEFORX01': BEFORE BREAK PROCESSING
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 SALARY (1:1)
  2 BONUS  (1:1,1:1)
*
1 #INCOME  (P11)
END-DEFINE
*
```

```
LIMIT 5
READ MYVIEW BY NAME FROM 'B'
 BEFORE BREAK PROCESSING
    COMPUTE #INCOME = SALARY(1) + BONUS(1,1)
  END-BEFORE
  /*
  DISPLAY NOTITLE NAME FIRST-NAME (AL=10)
                 'ANNUAL/INCOME' #INCOME 'SALARY' SALARY(1) (LC==) /
                 '+ BONUS' BONUS(1,1) (IC=+)
  AT BREAK OF #INCOME
    WRITE T*#INCOME '-'(24)
  END-BREAK
END-READ
END
```

プログラム BEFORX01 の出力：

```
        NAME          FIRST-NAME    ANNUAL      SALARY
                                    INCOME      + BONUS
-------------------- ---------- ------------ -----------

BACHMANN             HANS              56800 =     52800
                                                   +4000
                                 ------------------------
BAECKER              JOHANNES          81000 =     74400
                                                   +6600
                                 ------------------------
BAECKER              KARL              52650 =     48600
                                                   +4050
                                 ------------------------
BAGAZJA              MARJAN           152700 =    129700
                                                  +23000
                                 ------------------------
BAILLET              PATRICK          198500 =    188000
                                                  +10500
                                 ------------------------
```

ユーザー開始のブレイク処理 - PERFORM BREAK PROCESSING ステートメント

自動ブレイク処理では、処理ループ内の AT BREAK ステートメントの位置に関係なく、指定したコントロールフィールドの値が変わるたびに、AT BREAK ブロックに指定したステートメントが実行されます。

PERFORM BREAK PROCESSING ステートメントを使用すると、処理ループ内の特定の位置でブレイク処理を実行できます。PERFORM BREAK PROCESSING ステートメントは、プログラムの処理フローの中で検出されると実行されます。

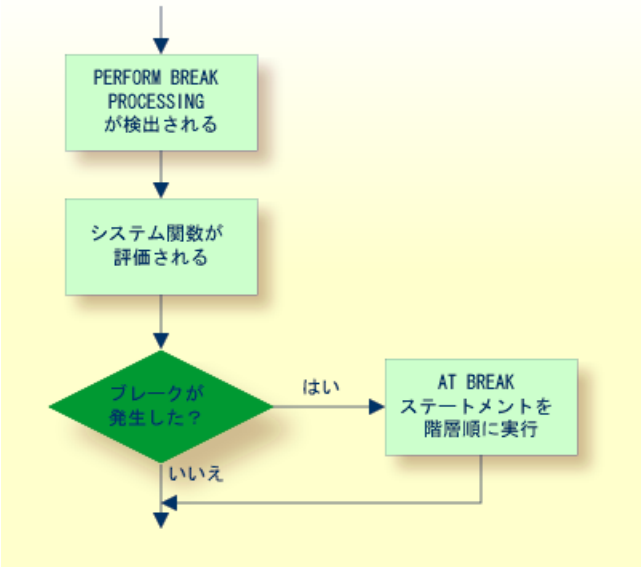
PERFORM BREAK PROCESSING の直後に、1 つ以上の AT BREAK ステートメントブロックを指定します。

```
...
PERFORM BREAK PROCESSING
  AT BREAK OF field1
    statements
  END-BREAK
  AT BREAK OF field2
    statements
  END-BREAK
...
```

PERFORM BREAK PROCESSING が実行されると、Natural によって、ブレイクが起こったかどうか、つまり、指定したコントロールフィールドの値が変わったかどうかがチェックされます。値が変わっている場合、指定したステートメントが実行されます。

PERFORM BREAK PROCESSING では、ブレイクが起こったかどうかをチェックする*前*に、システム関数が評価されます。

以下の図は、ユーザー開始のブレイク処理のフローロジックを示しています。



PERFORM BREAK PROCESSING ステートメントの例

```
** Example 'PERFBX01': PERFORM BREAK PROCESSING (with BREAK option
**                      in IF statement)
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 DEPT
  2 SALARY (1:1)
*
1 #CNTL    (N2)
END-DEFINE
*
LIMIT 7
READ MYVIEW BY DEPT
  AT BREAK OF DEPT              /* <- automatic break processing
    SKIP 1
    WRITE 'SUMMARY FOR ALL SALARIES         '
          'SUM:'   SUM(SALARY(1))
          'TOTAL:' TOTAL(SALARY(1))
    ADD 1 TO #CNTL
  END-BREAK
  /*
  IF SALARY (1) GREATER THAN 100000 OR BREAK #CNTL
    PERFORM BREAK PROCESSING    /* <- user-initiated break processing
    AT BREAK OF #CNTL
      WRITE 'SUMMARY FOR SALARY GREATER 100000'
            'SUM:'   SUM(SALARY(1))
            'TOTAL:' TOTAL(SALARY(1))
    END-BREAK
  END-IF
  /*
  IF SALARY (1) GREATER THAN 150000 OR BREAK #CNTL
    PERFORM BREAK PROCESSING    /* <- user-initiated break processing
    AT BREAK OF #CNTL
      WRITE 'SUMMARY FOR SALARY GREATER 150000'
            'SUM:'   SUM(SALARY(1))
            'TOTAL:' TOTAL(SALARY(1))
    END-BREAK
  END-IF
  DISPLAY NAME DEPT SALARY(1)
END-READ
END
```

プログラム PERFBX01 の出力：

```
Page      1                                                 04-12-14  14:13:35

        NAME          DEPARTMENT   ANNUAL
                         CODE      SALARY
-------------------- ---------- ----------

JENSEN               ADMA01         180000
PETERSEN             ADMA01         105000
MORTENSEN            ADMA01         320000
MADSEN               ADMA01         149000
BUHL                 ADMA01         642000

SUMMARY FOR ALL SALARIES          SUM:    1396000 TOTAL:    1396000
SUMMARY FOR SALARY GREATER 100000 SUM:    1396000 TOTAL:    1396000
SUMMARY FOR SALARY GREATER 150000 SUM:    1142000 TOTAL:    1142000
HERMANSEN            ADMA02         391500
PLOUG                ADMA02         162900

SUMMARY FOR ALL SALARIES          SUM:     554400 TOTAL:    1950400
SUMMARY FOR SALARY GREATER 100000 SUM:     554400 TOTAL:    1950400
SUMMARY FOR SALARY GREATER 150000 SUM:     554400 TOTAL:    1696400
```

48 データ計算

このchapterでは、データ計算に使用する、以下の算術演算ステートメントについて説明します。

- COMPUTE
- ADD
- SUBTRACT
- MULTIPLY
- DIVIDE

また、1つのオペランドの値を1つ以上のフィールドに転送するために使用する、以下のステートメントについて説明します。

- MOVE
- COMPRESS

Important: 最適な処理を行うために、算術演算ステートメントに使用するユーザー定義変数は、フォーマットP（パック型数値）で定義してください。

次のトピックについて説明します。

COMPUTE ステートメント

COMPUTE ステートメントは、算術演算を実行するために使用します。 以下の結合演算子を使用できます。

**	累乗
*	乗算
/	除算
+	加算
-	減算
()	カッコは、論理グループを示すために使用します。

例 **1**：

```
COMPUTE LEAVE-DUE = LEAVE-DUE * 1.1
```

この例では、フィールド LEAVE-DUE の値を 1.1 で乗算して、その結果をフィールド LEAVE-DUE に設定しています。

例 **2**：

```
COMPUTE #A = SQRT (#B)
```

この例では、フィールド #B の値の平方根を算出して、その結果をフィールド #A に割り当てています。

SQRT は、以下の算術演算ステートメントでサポートされている算術関数です。

- COMPUTE
- ADD
- SUBTRACT
- MULTIPLY
- DIVIDE

算術関数の概要については、後述の「算術関数」を参照してください。

例 **3**：

```
COMPUTE #INCOME = BONUS (1,1) + SALARY (1)
```

この例では、当年の最初のボーナスと現在の給与を加算して、その結果をフィールド #INCOME に割り当てています。

MOVE および COMPUTE ステートメント

ステートメント MOVE および COMPUTE は、オペランドの値を 1 つ以上のフィールドに転送するために使用できます。 オペランドには、テキスト項目や数値などの定数、データベースフィールド、ユーザー定義変数、システム変数、または場合によってはシステム関数を使用できます。

これらの 2 つのステートメントの違いは、以下の例のように、MOVE ステートメントでは移動元の値を左側に指定しますが、COMPUTE ステートメントでは割り当て元の値を右側に指定します。

例：

```
MOVE NAME TO #LAST-NAME
COMPUTE #LAST-NAME = NAME
```

ADD、SUBTRACT、MULTIPLY、および DIVIDE ステートメント

ADD、SUBTRACT、MULTIPLY、および DIVIDE ステートメントは、算術演算を実行するために使用します。

例：

```
ADD +5 -2 -1 GIVING #A
SUBTRACT 6 FROM 11 GIVING #B
MULTIPLY 3 BY 4 GIVING #C
DIVIDE 3 INTO #D GIVING #E
```

これらの 4 つのステートメントはすべて ROUNDED オプションを持っています。このオプションは、演算の結果を四捨五入する場合に使用します。

四捨五入のルールについては、「*演算割り当てのルール*」を参照してください。

これらのステートメントの詳細については、『ステートメント』ドキュメントを参照してください。

MOVE、SUBTRACT、および COMPUTE ステートメントの例

以下のプログラムは、算術演算ステートメントでのユーザー定義変数の使用方法を示しています。 3 人の従業員の年齢と給与を計算し、その結果を出力します。

```
** Example 'COMPUX01': COMPUTE
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 BIRTH
  2 JOB-TITLE
  2 SALARY          (1:1)
  2 BONUS           (1:1,1:1)
*
1 #DATE             (N8)
1 REDEFINE #DATE
  2 #YEAR           (N4)
```

```
  2 #MONTH          (N2)
  2 #DAY            (N2)
1 #BIRTH-YEAR       (A4)
1 REDEFINE #BIRTH-YEAR
  2 #BIRTH-YEAR-N   (N4)
1 #AGE              (N3)
1 #INCOME           (P9)
END-DEFINE
*
MOVE *DATN TO #DATE
*
READ (3) MYVIEW BY NAME STARTING FROM 'JONES'
  MOVE EDITED BIRTH (EM=YYYY) TO #BIRTH-YEAR
  SUBTRACT #BIRTH-YEAR-N FROM #YEAR GIVING #AGE
  /*
  COMPUTE #INCOME = BONUS (1:1,1:1) + SALARY (1:1)
  /*
  DISPLAY NAME 'POSITION' JOB-TITLE #AGE #INCOME
END-READ
END
```

プログラム COMPUX01 の出力：

```
Page      1                                                  04-11-11  14:15:54

        NAME                 POSITION          #AGE  #INCOME
-------------------- ------------------------- ---- ----------

JONES                MANAGER                     63      55000
JONES                DIRECTOR                    58      50000
JONES                PROGRAMMER                  48      31000
```

COMPRESS ステートメント

COMPRESS ステートメントは、複数のオペランドの内容を単一の英数字フィールドに転送（結合）するために使用します。

数値フィールドの先行ゼロおよび英数字フィールドの末尾の空白は、フィールド値を受け取りフィールドに転送する前に削除されます。

デフォルトでは、転送される値はそれぞれ1つの空白で区切られ、受け取りフィールドに格納されます。 その他に可能な分割方法については、『ステートメント』ドキュメントに記載されている、COMPRESS ステートメントオプション LEAVING NO SPACE の説明を参照してください。

例：

```
COMPRESS 'NAME:' FIRST-NAME #LAST-NAME INTO #FULLNAME
```

この例では、COMPRESS ステートメントを使用して、テキスト定数（'NAME:'）、データベースフィールド（FIRST-NAME）、およびユーザー定義変数（#LAST-NAME）を、1 つのユーザー定義変数（#FULLNAME）に結合しています。

COMPRESS ステートメントの詳細については、『ステートメント』ドキュメントに記載されている、COMPRESS ステートメントの説明を参照してください。

COMPRESS および MOVE ステートメントの例

以下のプログラムは、ステートメント MOVE および COMPRESS の使用方法を示しています。

```
** Example 'COMPRX01': COMPRESS
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 MIDDLE-I
*
1 #LAST-NAME (A15)
1 #FULL-NAME (A30)
END-DEFINE
*
READ (3) MYVIEW BY NAME STARTING FROM 'JONES'
  MOVE NAME TO #LAST-NAME
  /*
  COMPRESS 'NAME:' FIRST-NAME MIDDLE-I #LAST-NAME INTO #FULL-NAME
  /*
  DISPLAY #FULL-NAME (UC==) FIRST-NAME 'I' MIDDLE-I (AL=1) NAME
END-READ
END
```

プログラム COMPRX01 の出力：

結合されたフィールドの出力形式に注意してください。

```
Page      1                                              04-11-11  14:15:54

          #FULL-NAME                FIRST-NAME      I          NAME
============================== -------------------- - --------------------

NAME: VIRGINIA J JONES         VIRGINIA             J JONES
```

```
NAME: MARSHA JONES              MARSHA                  JONES
NAME: ROBERT B JONES            ROBERT                B JONES
```

複数行表示では、COMPRESS ステートメントを使用して、ユーザー定義変数にフィールド／テキストを結合すると便利です。

COMPRESS ステートメントの例

以下のプログラムでは、#FULL-SALARY、#FULL-NAME、および #FULL-CITY の 3 つのユーザー定義変数を使用しています。 例えば、#FULL-SALARY には、テキスト 'SALARY:' と、データベースフィールドの SALARY および CURR-CODE が格納されます。 その後、WRITE ステートメントでは、結合した変数のみを参照しています。

```
** Example 'COMPRX02': COMPRESS
************************************************************************
DEFINE DATA LOCAL
1 VIEWEMP VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 SALARY       (1:1)
  2 CURR-CODE    (1:1)
  2 CITY
  2 ADDRESS-LINE (1:1)
  2 ZIP
*
1 #FULL-SALARY   (A25)
1 #FULL-NAME     (A25)
1 #FULL-CITY     (A25)
END-DEFINE
*
READ (3) VIEWEMP BY CITY STARTING FROM 'NEW YORK'
  COMPRESS 'SALARY:' CURR-CODE(1) SALARY(1) INTO #FULL-SALARY
  COMPRESS FIRST-NAME NAME                  INTO #FULL-NAME
  COMPRESS ZIP CITY                         INTO #FULL-CITY
  /*
  DISPLAY 'NAME AND ADDRESS' NAME (EM=X^X^X^X^X^X^X^X^X^X^X^X^X)
  WRITE 1/5  #FULL-NAME
        1/37 #FULL-SALARY
        2/5  ADDRESS-LINE (1)
        3/5  #FULL-CITY
  SKIP 1
END-READ
END
```

プログラム COMPRX02 の出力：

```
Page      1                                                  04-11-11  14:15:54

   NAME AND ADDRESS
-----------------------

R U B I N
    SYLVIA RUBIN                    SALARY: USD 17000
    2003 SARAZEN PLACE
    10036 NEW YORK

W A L L A C E
    MARY WALLACE                    SALARY: USD 38000
    12248 LAUREL GLADE C
    10036 NEW YORK

K E L L O G G
    HENRIETTA KELLOGG               SALARY: USD 52000
    1001 JEFF RYAN DR.
    19711 NEWARK
```

算術関数

以下の Natural 算術関数は、算術演算ステートメント（ADD、COMPUTE、DIVIDE、SUBTRACT、MULTIPLY）でサポートされています。

算術の内容	Natural システム関数
field の絶対値。	ABS(*field*)
field アークタンジェント。	ATN(*field*)
field のコサイン。	COS(*field*)
field の指数。	EXP(*field*)
field の小数部分。	FRAC(*field*)
field の整数部分。	INT(*field*)
field の自然対数。	LOG(*field*)
field.の符号。	SGN(*field*)
field のサイン（正弦）。	SIN(*field*)
field の平方根。	SQRT(*field*)
field のタンジェント。	TAN(*field*)
英数字 *field* から抽出された数値。	VAL(*field*)

各算術関数の詳細については、『システム*関数*』ドキュメントも参照してください。

COMPUTE、MOVE、および COMPRESS ステートメントの他の例

次の例のプログラムを参照してください。

- *WRITEX11 - WRITE（nX、n/n および COMPRESS を使用）*
- *IFX03 - IF ステートメント*
- *COMPRX03 - COMPRESS（パラメータ LC および TC を使用）*

49 システム変数とシステム関数

このchapterでは、Natural システム変数と Natural システム関数の用途、および Natural プログラムでの使用方法を説明します。

次のトピックについて説明します。

システム変数

以下では次のトピックについて説明します。

用途

システム変数を使用して、システム情報を表示します。 これらは、Natural プログラム内のどこからでも参照できます。

Natural システム変数は、常に変わる情報、例えば以下のような、現在の Natural セッションに関する情報を提供します。

- 現在のライブラリ
- ユーザーおよび端末 ID
- ループ処理の現在のステータス
- レポート処理の現在のステータス
- 現在の日付と時刻

システム変数の一般的な使用方法については、後述の「*システム変数およびシステム関数の使用例*」、およびライブラリ **SYSEXPG** に含まれている例を参照してください。

システム変数に格納される情報は、Natural プログラムで適切なシステム変数を指定することにより、使用できます。 例えば、日付と時刻のシステム変数は、`DISPLAY`、`WRITE`、`PRINT`、`MOVE`、または `COMPUTE` の各ステートメントで指定できます。

システム変数の特徴

システム変数の名前はすべて、アスタリスク（*）で始まります。

フォーマット／長さ

システム変数のフォーマット／長さの情報については、『システム変数』ドキュメントを参照してください。 次の省略形が使用されます。

フォーマット	
A	英数字
B	バイナリ
D	日付
I	整数
L	論理
N	数値（アンパック）
P	パック型数値
T	時刻

内容変更の可否

それぞれの説明に記載されているこの項目は、Naturalプログラム内で別の値をシステム変数に割り当てることができるかどうか、つまり、Naturalによって生成された内容を上書きできるかどうかを示します。

機能別のシステム変数

Natural システム変数は、以下のように分類されます。

- アプリケーション関連システム変数
- 日時システム変数
- 入出力関連システム変数
- Natural 環境関連システム変数
- システム環境関連システム変数
- XML 関連システム変数

すべてのシステム変数の詳細については、『システム変数』ドキュメントを参照してください。

システム関数

Naturalシステム関数は、レコードを処理した後（ただしブレイク処理の発生前）にデータに適用できる統計関数および算術関数によって構成されます。

システム関数は、AT END OF PAGE、AT END OF DATA、またはAT BREAKの各ステートメントとともに使用されるDISPLAY、WRITE、PRINT、MOVE、またはCOMPUTEの各ステートメントに指定できます。

AT END OF PAGEステートメントの場合、対応するDISPLAYステートメントでGIVE SYSTEM FUNCTIONS節を使用する必要があります（以下の例を参照）。

システム関数は、以下の機能に分類されます。

- 処理ループで使用する Natural システム関数
- 算術関数
- その他の関数

有効なすべてのシステム関数の詳細については、『*システム関数*』ドキュメントを参照してください。

『*システム関数*』ドキュメントの「*処理ループでのシステム関数の使用*」も参照してください。

システム関数の一般的な使用方法については、以下のプログラム例およびライブラリ**SYSEXPG**に格納されている例を参照してください。

システム変数およびシステム関数の使用例

以下のプログラム例は、システム変数およびシステム関数の使用方法を示しています。

```
** Example 'SYSVAX01': System variables and system functions
************************************************************************
DEFINE DATA LOCAL
1 MYVIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 JOB-TITLE
  2 INCOME    (1:1)
    3 CURR-CODE
    3 SALARY
    3 BONUS   (1:1)
END-DEFINE
```

```
*
WRITE TITLE LEFT JUSTIFIED 'EMPLOYEE SALARY REPORT AS OF' *DAT4E /
*
READ (3) MYVIEW BY CITY STARTING FROM 'E'
  DISPLAY GIVE SYSTEM FUNCTIONS
          NAME (AL=15) JOB-TITLE (AL=15) INCOME (1:1)
  AT START OF DATA
    WRITE 'REPORT CREATED AT:' *TIME 'HOURS' /
  END-START
  AT END OF DATA
    WRITE / 'LAST PERSON SELECTED:' OLD (NAME) /
  END-ENDDATA
END-READ
*
AT END OF PAGE
  WRITE 'AVERAGE SALARY:' AVER (SALARY(1))
END-ENDPAGE
END
```

説明：

- システム変数 *DATE は、WRITE TITLE ステートメントで出力されています。
- システム変数 *TIME は、AT START OF DATA ステートメントで出力されています。
- システム関数 OLD は、AT END OF DATA ステートメントで使用されています。
- システム関数 AVER は、AT END OF PAGE ステートメントで使用されています。

プログラム SYSVAX01 の出力：

システム変数およびシステム関数がどのように表示されているかに注意してください。

```
EMPLOYEE SALARY REPORT AS OF 11/11/2004

     NAME           CURRENT                  INCOME
                    POSITION
                                    CURRENCY   ANNUAL     BONUS
                                      CODE     SALARY
--------------- --------------- -------- ---------- ----------

REPORT CREATED AT: 14:15:55.0 HOURS

DUYVERMAN       PROGRAMMER      USD           34000          0
PRATT           SALES PERSON    USD           38000       9000
MARKUSH         TRAINEE         USD           22000          0

LAST PERSON SELECTED: MARKUSH
```

```
AVERAGE SALARY:      31333
```

システム変数の他の例

次の例のプログラムを参照してください。

- *EDITMX05 - 編集マスク（日付および時刻システム変数の EM）*
- *READX04 - READ（FIND およびシステム変数 *NUMBER と *COUNTER を使用）*
- *WTITLX01 - WRITE TITLE（PAGE-NUMBER を使用）*

システム関数の他の例

次の例のプログラムを参照してください。

- *ATBREX06 - AT BREAK OF（NMIN、NAVER、NCOUNT を MIN、AVER、COUNT と比較）*
- *ATENPX01 - AT END OF PAGE（DISPLAY の GIVE SYSTEM FUNCTIONS で有効なシステム関数を使用）*

50 スタック

Natural スタックは、Natural コマンド、ユーザー定義コマンド、および INPUT ステートメントで使用する入力データを格納できる、一種の 「中間ストレージ」 です。

このchapterでは、次のトピックについて説明します。

Natural スタックの使用

スタックには、一連のログオンコマンドのように、頻繁に連続して実行する一連の機能を格納できます。

スタックに格納されたデータ／コマンドは、その一番上に 「スタック」 されます。 格納するデータ／コマンドをスタックの一番上と一番下のどちらに置くかは指定できます。 スタック内のデータ／コマンドは、スタックされている順番（スタックの一番上から）でのみ処理できます。

プログラムでは、システム変数*DATAを参照することにより、スタックの内容を確認できます。詳細については、『システム変数』ドキュメントを参照してください。

スタック処理

スタックに格納されているコマンド／データの処理は、実行する機能によって異なります。

コマンドを実行しようとしている場合、つまり、NEXT プロンプトを表示しようとしている場合、Naturalはまず、スタックの一番上がコマンドかどうかをチェックします。 スタックの一番上がコマンドの場合、NEXT プロンプトは抑制されます。そして、コマンドが読み込まれ、スタックから削除されます。その後、NEXTプロンプトに応答して手動で入力されたかのように、コマンドが実行されます。

入力フィールドが含まれている INPUT ステートメントを実行しようとしている場合、Naturalはまず、スタックの一番上に何らかの入力データがあるかどうかをチェックします。 入力データがある場合、そのデータが INPUT ステートメントにデリミタモードで渡されます。スタックから読み込まれるデータは、INPUT ステートメントの変数と互換性のあるフォーマットである必要があります。その後、データはスタックから削除されます。 INPUT ステートメントの説明の「*Natural スタックデータの処理*」も参照してください。

スタックデータを使用して実行したINPUTステートメントをREINPUTステートメント経由で再実行すると、INPUTステートメントの画面が再表示され、最初に実行したときと同じスタックデータが表示されます。 REINPUT ステートメントでは、スタックから他のデータは読み込まれません。

Naturalプログラムが正常に終了するときは、コマンドがスタックの一番上に来るか、またはスタックがクリアされるまで、スタックは一番上からフラッシュされます。Naturalプログラムが端末コマンド %%またはエラーで終了した場合は、スタックが完全にクリアされます。

スタックへのデータの格納

以下の方法を使用して、データ／コマンドをスタックに格納します。

- STACK パラメータ
- STACK ステートメント
- FETCH および RUN ステートメント

STACK パラメータ

Natural プロファイルパラメータ STACK は、データ／コマンドをスタックに格納するために使用できます。 STACK パラメータ（『パラメータリファレンス』を参照）は、Natural のインストール時に Natural 管理者が Natural パラメータモジュールに指定するか、またはダイナミックパラメータとして Natural の起動時に指定します。

データ／コマンドを STACK パラメータ経由でスタックに格納する場合、複数のコマンドはセミコロン（;）で分割する必要があります。 コマンドを一連のデータまたはコマンド要素に格納して渡す場合、コマンドの前にセミコロンを付ける必要があります。

複数の INPUT ステートメントに対するデータは、コロン（:）で分割する必要があります。 単独の INPUT ステートメントで読み込むデータには、コロンを前に付ける必要があります。パラメータを必要とするコマンドをスタックする場合、コマンドとパラメータの間にはコロンを置きません。

セミコロンとコロンはでセパレータ文字として解釈されるため、入力データ自体にセミコロンとコロンを使用することはできません。

STACK ステートメント

プログラム内で STACK ステートメントを使用すると、データ／コマンドをスタックに格納できます。 1 つの STACK ステートメントで指定したデータ要素は、1 つの INPUT ステートメントで使用します。これは、複数の INPUT ステートメントに対するデータをスタックに格納するには、複数の STACK ステートメントを使用する必要があることを意味します。

スタックには、フォーマットされていないデータもフォーマットされているデータも格納できます。

- フォーマットされていないデータがスタックから読み込まれると、データ文字列はデリミタモードで解釈され、セッションパラメータ IA（INPUT 割り当て文字）および ID（INPUT 区切り文字）で指定されている文字が、割り当てとデータ分割のキーワードに対する制御文字として処理されます。
- フォーマットされているデータがスタックに格納されている場合、フィールドの個々の内容が分割され、対応する INPUT ステートメントの 1 つの入力フィールドに渡されます。 スタックに格納するデータにデリミタ文字、制御文字、または DBCS 文字が含まれている場合、これ

らの文字が不適切に解釈されないように、フォーマットしてスタックに格納する必要があります。

STACK ステートメントの詳細については、『ステートメント』ドキュメントを参照してください。

FETCH および RUN ステートメント

呼び出すプログラムに渡すパラメータを持つ FETCH または RUN ステートメントを実行すると、これらのパラメータはスタックの一番上に格納されます。

スタックのクリア

スタックの内容は、RELEASE ステートメントで削除できます。 RELEASE ステートメントの詳細については、『ステートメント』ドキュメントを参照してください。

Note: Natural プログラムが端末コマンド %%またはエラーで終了した場合は、スタックが完全にクリアされます。

51 日付情報の処理

このchapterでは、Naturalアプリケーションでの日付情報の処理について、さまざまな面から説明します。

次のトピックについて説明します。

日付フィールドの編集マスクおよび日付システム変数

日付フィールドの値を特定の表現で出力する場合は、通常、フィールドに編集マスクを指定します。 編集マスクを使用して、1 文字ずつどのように出力するかを指定します。

現在の日付を特定の表現で使用する場合、日付フィールドを定義して編集マスクを指定する必要はありません。代わりに、*日付システム変数*を使用します。Naturalには、さまざまな日付システム変数が用意されています。その中には、表現の異なる現在日付も含まれています。 これらの表現には2桁の年コンポーネントを持つものもあれば、4桁の年コンポーネントを持つものもあります。

すべての日付システム変数のリストと詳細については、『システム変数』ドキュメントを参照してください。

デフォルトの日付編集マスク - DTFORM パラメータ

`DTFORM`プロファイルパラメータによって、Naturalレポートのデフォルトタイトルの日付部分、日付定数、および日付入力に使用されるデフォルトのフォーマットが決まります。

この日付フォーマットを使用して、日付の年、月、日の各コンポーネントの順序、およびこれらのコンポーネント間で使用するデリミタ文字を指定します。

有効な`DTFORM`の設定は、以下のとおりです。

設定	日付フォーマット*	例
DTFORM=I	*yyyy-mm-dd*	2005-12-31
DTFORM=G	*dd.mm.yyyy*	31.12.2005
DTFORM=E	*dd/mm/yyyy*	31/12/2005
DTFORM=U	*mm/dd/yyyy*	12/31/2005

* *dd* = 日、*mm* = 月、*yyyy* = 年

`DTFORM`パラメータは、Naturalパラメータモジュール／ファイル内で設定できます。また、Naturalを起動するときにダイナミックに設定することもできます。デフォルトでは、`DTFORM=I`が適用されます。

英数字表現の日付フォーマット - DF パラメータ

編集マスクが指定されている場合、フィールド値の表現は編集マスクによって決まります。 編集マスクが指定されていない場合、フィールド値の表現は、セッションパラメータ DF とプロファイルパラメータ DTFORM の組み合せによって決まります。

DF パラメータでは、以下の日付表現の 1 つを選択できます。

DF=S	2 桁の年コンポーネントとデリミタを使用する 8 バイトの表記です（*yy-mm-dd*）。
DF=I	デリミタを使用しない、4 桁の年コンポーネントの 8 バイトの表記です（*yyyymmdd*）。
DF=L	4 桁の年コンポーネントとデリミタを使用する 10 バイトの表記です（*yyyy-mm-dd*）。

各表現に対する、年、月、日の各コンポーネントの順序、および使用するデリミタ文字は、DTFORM パラメータによって決まります。

デフォルトでは、DF=S が適用されます（INPUT ステートメントを除く。下記参照）。

セッションパラメータ DF は、コンパイル時に評価されます。

このパラメータは、以下のステートメントで指定できます。

- FORMAT
- INPUT、DISPLAY、WRITE、および PRINT のステートメントレベルおよび要素（フィールド）レベル
- MOVE、COMPRESS、STACK、RUN、および FETCH の要素（フィールド）レベル

上記のステートメントのいずれかで指定すると、DF パラメータは以下のように適用されます。

ステートメント	DF パラメータの効果
DISPLAY、WRITE、PRINT	これらのステートメントの1つを使用して日付変数の値を出力すると、出力する前に値が英数字表現に変換されます。 DF パラメータにより、使用される表現が決まります。
MOVE、COMPRESS	MOVE または COMPRESS の各ステートメントを使用して日付変数の値を英数字フィールドに転送すると、転送する前に値が英数字表現に変換されます。 DF パラメータにより、使用される表現が決まります。
STACK、RUN、FETCH	日付変数の値をスタックに格納すると、スタックに格納する前に値が英数字表現に変換されます。 DF パラメータにより、使用される表現が決まります。 FETCH ステートメントまたは RUN ステートメントのパラメータとして日付変数を指定する場合も同様です（これらのパラメータもスタック経由で渡されるため）。

ステートメント	DF パラメータの効果
INPUT	INPUT ステートメントでデータ変数を使用する場合、DF パラメータによって、値をどのようにフィールドに入力する必要があるかが決まります。 ただし、DF パラメータを*指定せずに*日付変数を INPUT ステートメントで使用すると、デリミタ付きの2桁の年コンポーネント、またはデリミタなしの4桁の年コンポーネントのいずれかで日付を入力できます。 この場合も、年、月、日の各コンポーネントの順序、および使用するデリミタ文字は、DTFORMパラメータによって決まります。

Note: DF=S で提供される年情報は2桁のみです。これは、日付の値に世紀が含まれていると、変換時にこの情報が失われることを意味します。 世紀の情報を保持するには、DF=I または DF=L を設定します。

WRITE ステートメントでの DF パラメータの例

これらの例では、DTFORM=G が適用されると仮定します。

```
/* DF=S (default)
WRITE *DATX   /* Output has this format: dd.mm.yy
END
```

```
FORMAT DF=I
WRITE *DATX   /* Output has this format: ddmmyyyy
END
```

```
FORMAT DF=L
WRITE *DATX   /* Output has this format: dd.mm.yyyy
END
```

MOVE ステートメントでの DF パラメータの例

この例では、DTFORM=E が適用されると仮定します。

```
DEFINE DATA LOCAL
  1 #DATE  (D) INIT <D'31/12/2005'>
  1 #ALPHA (A10)
END-DEFINE
...
MOVE #DATE        TO #ALPHA /* Result: #ALPHA contains 31/12/05
MOVE #DATE (DF=I) TO #ALPHA /* Result: #ALPHA contains 31122005
```

```
MOVE #DATE (DF=L) TO #ALPHA /* Result: #ALPHA contains 31/12/2005
...
```

STACK ステートメントでの DF パラメータの例

この例では、DTFORM=I が適用されると仮定します。

```
DEFINE DATA LOCAL
  1 #DATE  (D) INIT <D'2005-12-31'>
  1 #ALPHA1(A10)
  1 #ALPHA2(A10)
  1 #ALPHA3(A10)
END-DEFINE
...
STACK TOP DATA #DATE (DF=S) #DATE (DF=I) #DATE (DF=L)
...
INPUT #ALPHA1 #ALPHA2 #ALPHA3
...
/* Result: #ALPHA1 contains 05-12-31
/*         #ALPHA2 contains 20051231
/*         #ALPHA3 contains 2005-12-31
...
```

INPUT ステートメントでの DF パラメータの例

この例では、DTFORM=I が適用されると仮定します。

```
DEFINE DATA LOCAL
  1 #DATE1 (D)
  1 #DATE2 (D)
  1 #DATE3 (D)
  1 #DATE4 (D)
END-DEFINE
...
INPUT #DATE1 (DF=S) /* Input must have this format: yy-mm-dd
      #DATE2 (DF=I) /* Input must have this format: yyyymmdd
      #DATE3 (DF=L) /* Input must have this format: yyyy-mm-dd
```

```
      #DATE4         /* Input must have this format: yy-mm-dd or yyyymmdd
...
```

出力用の日付フォーマット - DFOUT パラメータ

セッション／プロファイルパラメータ DFOUT は、編集マスクが指定されておらず、DF パラメータも適用されていない、INPUT、DISPLAY、WRITE、および PRINT の各ステートメントの日付フィールドにのみ適用されます。

INPUT、DISPLAY、PRINT、および WRITE の各ステートメントによって表示され、編集マスクが指定されておらず、DF パラメータも適用されてない日付フィールドの場合は、プロファイル／セッションパラメータ DFOUT によって、表示するフィールド値のフォーマットが決まります。

有効な DFOUT の設定は、以下のとおりです。

DFOUT=S	日付変数は、2 桁の年コンポーネント、およびパラメータ DTFORM で指定されているデリミタを使用して表示されます（*yy-mm-dd*）。
DFOUT=I	日付変数は、デリミタなしの 4 桁の年コンポーネントで表示されます（*yyyymmdd*）。

デフォルトでは、DFOUT=S が適用されます。 どちらの DFOUT 設定でも、日付値の年、月、日の各コンポーネントの順序は、DTFORM パラメータによって決まります。

どちらの日付値表現でも 8 バイトのフィールドに収まるため、日付フィールドの長さは DFOUT 設定の影響を受けません。

DFOUT パラメータは、Natural パラメータモジュール／ファイル内で、または Natural を起動するときにダイナミックに、あるいはシステムコマンド GLOBALS を使用して設定できます。 これは、ランタイム時に評価されます。

例：

この例では、DTFORM=I が適用されると仮定します。

```
DEFINE DATA LOCAL
1 #DATE  (D) INIT <D'2005-12-31'>
END-DEFINE
...
WRITE #DATE        /* Output if DFOUT=S is set ...: 05-12-31
                   /* Output if DFOUT=I is set ...: 20051231
```

```
WRITE #DATE (DF=L) /* Output (regardless of DFOUT): 2005-12-31
...
```

スタック用の日付フォーマット - DFSTACK パラメータ

セッション／プロファイルパラメータ DFSTACK は、DF パラメータが指定されていない、STACK、FETCH、および RUN の各ステートメントで使用されている日付フィールドにのみ適用されます。

DFSTACK パラメータによって、STACK、RUN、FETCH の各ステートメント経由でスタックに格納される日付変数の値のフォーマットが決まります。

有効な DFSTACK の設定は、以下のとおりです。

DFSTACK=S	日付変数は、2 桁の年コンポーネント、およびプロファイルパラメータ DTFORM で指定されているデリミタを使用してスタックに格納されます（*yy-mm-dd*）。
DFSTACK=C	DFSTACK=S と同じです。 ただし、世紀の変更はランタイム時にインターセプトされます。
DFSTACK=I	日付変数は、デリミタなしの 4 桁の年コンポーネントでスタックに格納されます（*yyyymmdd*）。

デフォルトでは、DFSTACK=S が適用されます。 DFSTACK=S は、世紀の情報なし（つまり、失われます）で日付値がスタックに格納されることを意味します。 その後、値をスタックから読み込んで別の日付変数に格納すると、世紀情報は現在のものであるとみなされるか、または YSLW パラメータ（下記参照）の設定によって決まります。 この操作によって、元の日付値と異なる世紀が設定される可能性があります。ただし、この場合、Natural はエラーを発行しません。

世紀の情報なしで日付値がスタックに格納されるという点では、DFSTACK=C は DFSTACK=S と同じ動作をします。 ただし、（YSLW パラメータ、または元の世紀が現在の世紀でなかったために）スタックから読み込んだ値の世紀情報が元の日付値と異なった場合、Naturalはランタイムエラーを発行します。

Note: このランタイムエラーは、値をスタックに格納する時点ですでに発行されています。

DFSTACK=I を使用すると、世紀の情報を失わずに日付値を8バイトの長さでスタックに格納できます。

DFSTACK パラメータは、Natural パラメータモジュール／ファイル内で、または Natural を起動するときにダイナミックに、あるいはシステムコマンド GLOBALS を使用して設定できます。 これは、ランタイム時に評価されます。

例：

以下の例では、DTFORM=I および YSLW=0 が適用されているものとします。

```
DEFINE DATA LOCAL
  1 #DATE  (D) INIT <D'2005-12-31'>
  1 #ALPHA1(A8)
  1 #ALPHA2(A10)
END-DEFINE
...
STACK TOP DATA #DATE #DATE (DF=L)
...
INPUT #ALPHA1 #ALPHA2
...
/* Result if DFSTACK=S or =C is set: #ALPHA1 contains 05-12-31
/* Result if DFSTACK=I is set .....: #ALPHA1 contains 20051231
/* Result (regardless of DFSTACK) .: #ALPHA2 contains 2005-12-31
...
```

年スライディングウィンドウ - YSLW パラメータ

プロファイルパラメータ YSLW を使用すると、2 桁の年の値に対する世紀を指定できます。

YSLW パラメータは、Natural パラメータモジュール／ファイル内で設定できます。また、Natural を起動するときにダイナミックに設定することもできます。2桁の年コンポーネントを持つ英数字の日付値を日付変数に移すと、このパラメータはランタイム時に評価されます。 これは、以下のデータ値に適用されます。

- 算術関数 VAL(*field*) で使用するデータ値
- 論理条件の IS(D) オプションで使用するデータ値
- 入力データとしてスタックから読み込まれたデータ値
- 入力データとして入力フィールドに入力されたデータ値

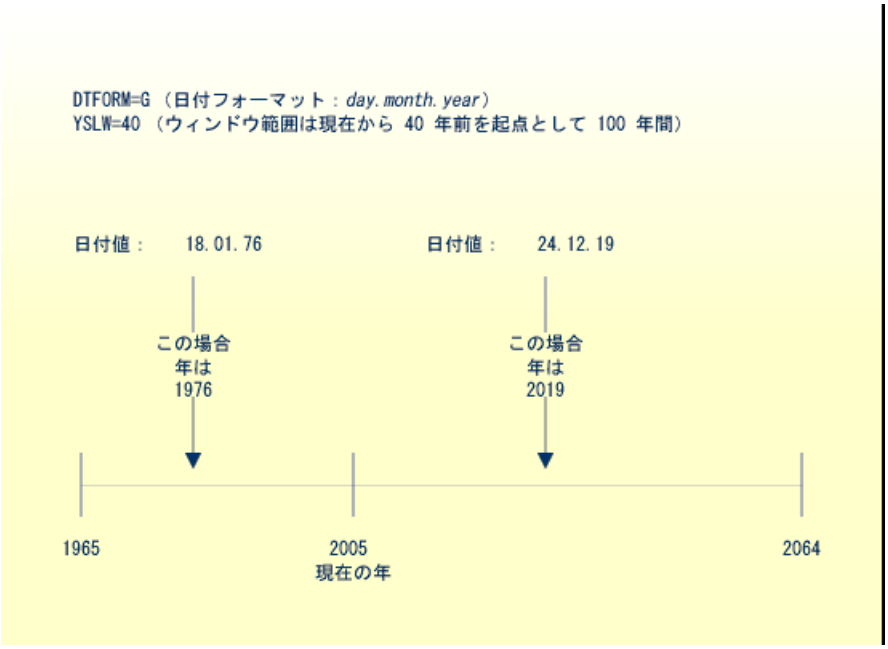
YSLW パラメータによって、「年スライディングウィンドウ」 でカバーする年の範囲を特定します。 スラインディングウィンドウのメカニズムは、2 桁の年の日付が 100 年の 「ウィンドウ」 内にあるものと仮定します。 この 100 年の範囲で、すべての 2 桁の年の値を特定の世紀に一意に関連付けることができます。

YSLW パラメータを使用して、その 100 年の範囲を何年前から開始するかを指定します。現在の年から YSLW の値を引くことにより、ウィンドウ範囲の最初の年が決定します。

YSLW パラメータの有効な値は、0～99 です。 デフォルト値は YSLW=0 です。これは、スライディングウィンドウメカニズムを使用しないことを意味します。つまり、2桁の年を持つ日付は、現在の世紀にあるとみなされます。

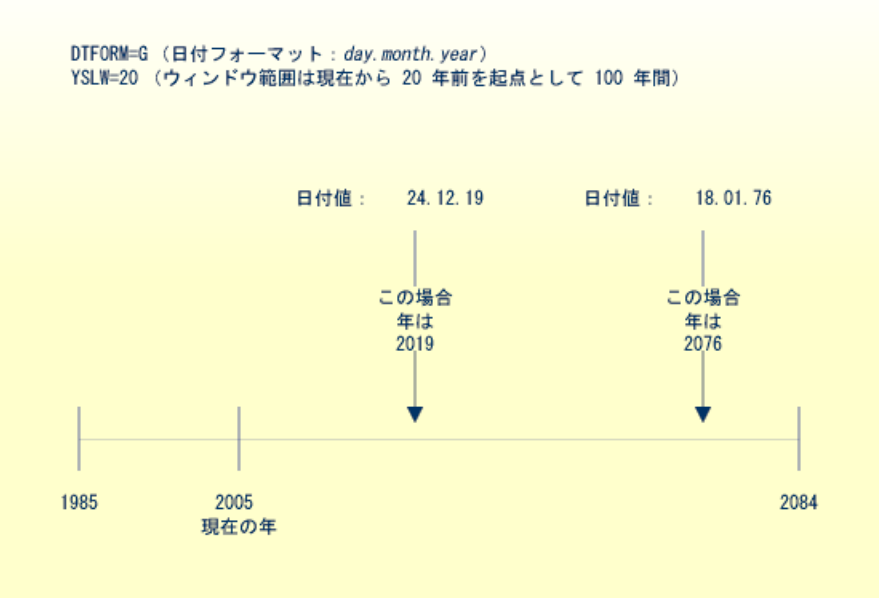
例 **1**：

現在の年が 2005 年の場合に YSLW=40 を指定すると、スラインディングウィンドウは 1965～2064 年をカバーします。 65～99 の 2 桁の年の値 *nn* は、19*nn* と解釈されます。一方、00～64 の 2 桁の年の値 *nn* は、20*nn* と解釈されます。



例 **2**：

現在の年が 2005 年の場合に YSLW=20 を指定すると、スラインディングウィンドウは 1985～2084 年をカバーします。 85～99 の 2 桁の年の値 *nn* は、19*nn* と解釈されます。一方、00～84 の 2 桁の年の値 *nn* は、20*nn* と解釈されます。



DFSTACK と YSLW の組み合わせ

以下の例で、パラメータ DFSTACK と YSLW をさまざまに組み合わせて使用することによる効果を説明します。

Note: これらのすべての例において、DTFORM=I が適用されているものとします。

例 **1**：

以下の例では、現在の年は 2005 年で、パラメータの設定には DFSTACK=S（デフォルト値）および YSLW=20 が適用されているものとします。

```
DEFINE DATA LOCAL
  1 #DATE1 (D) INIT <D'1956-12-31'>
  1 #DATE2 (D)
END-DEFINE
...
STACK TOP DATA #DATE1 /* century information is lost (year 56 is stacked)
...
INPUT #DATE2          /* year sliding window determines 56 to be 2056
...
```

```
/* Result: #DATE2 contains 2056-12-31
           even if #DATE1 is set to <D'2156-12-31'>
```

この場合、年スライディングウィンドウは適切に設定されないため、世紀の情報は（意図しない値に）変更されます。

例 **2**：

以下の例では、現在の年は 2005 年で、パラメータの設定には DFSTACK=S（デフォルト値）および YSLW=60 が適用されているものとします。

```
DEFINE DATA LOCAL
  1 #DATE1 (D) INIT <D'1956-12-31'>
  1 #DATE2 (D)
END-DEFINE
...
STACK TOP DATA #DATE1 /* century information is lost (year 56 is stacked)
...
INPUT #DATE2          /* year sliding window determines 56 to be 1956
...
/* Result: #DATE2 contains 1956-12-31
           even if #DATE1 is set to <D'2056-12-31'>
```

この場合、年スライディングウィンドウは適切に設定されるため、世紀の情報は正しく復元されます。

例 **3**：

以下の例では、現在の年は 2005 年で、パラメータの設定には DFSTACK=C および YSLW=0（デフォルト値）が適用されているものとします。

```
DEFINE DATA LOCAL
  1 #DATE1 (D) INIT <D'1956-12-31'>
  1 #DATE2 (D)
END-DEFINE
...
STACK TOP DATA #DATE1 /* century information is lost (year 56 is stacked)
...
INPUT #DATE2          /* 56 is assumed to be in current century -> 1956
...
/* Result: RUNTIME ERROR (UNINTENDED CENTURY CHANGE)
```

この場合、世紀情報は（意図しない値に）変更されます。 ただし、この変更は DFSTACK=C 設定によってインターセプトされます。

例 **4**：

以下の例では、現在の年は2005年で、パラメータの設定にはDFSTACK=CおよびYSLW=60が適用されているものとします。

```
DEFINE DATA LOCAL
  1 #DATE1 (D) INIT <D'2056-12-31'>
  1 #DATE2 (D)
END-DEFINE
...
STACK TOP DATA #DATE1 /* century information is lost (year 56 is stacked)
...
INPUT #DATE2          /* year sliding window determines 56 to be 1956
...
/* Result: RUNTIME ERROR (UNINTENDED CENTURY CHANGE)
```

この場合、世紀情報は年スライディングウィンドウに従って変更されます。 ただし、この変更はDFSTACK=C設定によってインターセプトされます。

年固定ウィンドウ

このトピックの詳細については、プロファイルパラメータYSLWの説明を参照してください。

デフォルトページタイトル用の日付フォーマット - DFTITLE パラメータ

セッション／プロファイルパラメータDFTITLEによって、デフォルトのページタイトル（DISPLAY、WRITE、PRINTの各ステートメントの出力）の日付フォーマットが決まります。

DFTITLE=S	日付は、デリミタ付きの2桁の年コンポーネントで出力されます（*yy-mm-dd*）。
DFTITLE=L	日付は、デリミタ付きの4桁の年コンポーネントで出力されます（*yyyy-mm-dd*）。
DFTITLE=I	日付は、デリミタなしの4桁の年コンポーネントで出力されます（*yyyymmdd*）。

これらの各出力フォーマットに対する、年、月、日の各コンポーネントの順序、および使用するデリミタ文字は、DTFORMパラメータによって決まります。

DFTITLEパラメータは、Naturalパラメータモジュール／ファイル内で、またはNaturalを起動するときにダイナミックに、あるいはシステムコマンドGLOBALSを使用して設定できます。 これは、ランタイム時に評価されます。

例：

この例では、DTFORM=I が適用されると仮定します。

```
WRITE 'HELLO'
END
/*
/* Date in page title if DFTITLE=S is set ...: 05-10-31
/* Date in page title if DFTITLE=L is set ...: 2005-10-31
/* Date in page title if DFTITLE=I is set ...: 20051031
```

Note: DFTITLE パラメータは、WRITE TITLE ステートメントで指定されているユーザー定義のページタイトルには効果がありません。

52 テキスト表記

INPUT、DISPLAY、WRITE、WRITE TITLE、およびWRITE TRAILERの各ステートメントでは、テキスト表記を使用して、これらのステートメントとともに使用するテキストを定義できます。

このchapterでは、次のトピックについて説明します。

ステートメントで使用するテキストの定義 - '*text*' 表記

ステートメントで使用するテキスト（プロンプトメッセージなど）は、アポストロフィ（'）または引用符（"）で囲む必要があります。 アポストロフィ2つ（''）と引用符（"）を混同しないようにしてください。

引用符で囲んだテキストは、小文字から大文字に自動変換できます。 自動変換を無効にするには、エディタプロファイルの設定を変更します。 詳細については、（『*Natural スタジオの使用*』ドキュメントの）「*プログラムエディタオプション*」に記載されている、［テキスト定数の無視］および［大文字変換］の説明を参照してください。

テキスト自体には1～72文字を使用できますが、複数行にわたって記述することはできません。

テキスト要素は、ハイフンを使用して連結できます。

例：

```
DEFINE DATA LOCAL
1 #A(A10)
END-DEFINE

INPUT 'Input XYZ' (CD=BL) #A
WRITE '=' #A
WRITE 'Write1 ' - 'Write2 ' - 'Write3' (CD=RE)
END
```

テキスト文字列の一部としてのアポストロフィの使用

Naturalプロファイルパラメータ TQ（引用符の変換）がONに設定されていると、以下の処理が適用されます。 これはデフォルト設定です。

アポストロフィで囲まれたテキスト文字列内で1つのアポストロフィを表すには、2つのアポストロフィ（''）または1つの引用符（"）を使用する必要があります。 どちらの表記を使用しても、単一のアポストロフィが出力されます。

引用符で囲まれたテキスト文字列内で1つのアポストロフィを表すには、1つのアポストロフィを指定します。

アポストロフィの例：

```
#FIELDA = 'O''CONNOR'
#FIELDA = 'O"CONNOR'
#FIELDA = "O'CONNOR"
```

上記の 3 例すべてで、以下の結果が出力されます。

```
O'CONNOR
```

テキスト文字列の一部としての引用符の使用

Natural プロファイルパラメータ TQ（引用符の変換）が OFF に設定されていると、以下の処理が適用されます。 デフォルトの設定は TQ=ON です。

1 つのアポストロフィで囲まれたテキスト文字列内で 1 つの引用符を表すには、1 つの引用符を指定します。

引用符で囲まれたテキスト文字列内で 1 つの引用符を表すには、2 つの引用符（""）を指定します。

引用符の例：

```
#FIELDA = 'O"CONNOR'
#FIELDA = "O""CONNOR"
```

上記の 2 例すべてで、以下の結果が出力されます。

```
O"CONNOR
```

フィールド値の前に n 回出力する文字の定義 - '*c*'(*n*) 表記

単一の文字をテキストとして複数回出力するには、以下の表記を使用します。

```
'c'(n)
```

c には出力する文字を指定し、*n* には文字を出力する回数を指定します。 *n* の最大値は、249 です。

例：

```
WRITE '*'(3)
```

文字 *c* の前後には、アポストロフィの代わりに引用符も使用できます。

53 ユーザーコメント

ユーザーコメントとは、ソースコードのステートメント間に追加または挿入する記述や説明メモのことです。これらの情報は、自分以外のプログラマによって作成または変更されたソースコードの理解や管理に特に有効です。 また、コメント開始を示す文字列は、テストのためにステートメントの機能や複数のソースコード行を一時的に無効にするために使用することもできます。

Natural でソースコードにコメントを入力するには、以下の 2 つの方法があります。

ソースコード行全体をコメントとして使用する方法

ソースコード行全体をユーザーコメントとして使用する場合、行の先頭に以下の1つを入力します。

- アスタリスクと空白（* ）
- 2 つのアスタリスク（**）
- スラッシュとアスタリスク（/*）

```
*  USER COMMENT
** USER COMMENT
/* USER COMMENT
```

例：

以下の例のように、コメント行は、ソースコードの構造を明確にするために使用することもできます。

```
** Example 'LOGICX03': BREAK option in logical condition
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 BIRTH
*
1 #BIRTH (A8)
END-DEFINE
*
LIMIT 10
READ EMPLOY-VIEW BY BIRTH
  MOVE EDITED BIRTH (EM=YYYYMMDD) TO #BIRTH
  /*
  IF BREAK OF #BIRTH /6/
    NEWPAGE IF LESS THAN 5 LINES LEFT
    WRITE / '-' (50) /
  END-IF
```

```
  /*
  DISPLAY NOTITLE BIRTH (EM=YYYY-MM-DD) NAME FIRST-NAME
END-READ
END
```

ソースコード行の途中からコメントとして使用する方法

ソースコード行の途中からのみをユーザーコメントとして使用する場合、空白、スラッシュ、およびアスタリスク（/*）を入力すると、行内のこの表記以降の部分がコメントとしてマークされます。

```
ADD 5 TO #A          /* USER COMMENT
```

例：

```
** Example 'LOGICX04': IS option as format/length check
************************************************************************
DEFINE DATA LOCAL
1 #FIELDA (A10)          /* INPUT FIELD TO BE CHECKED
1 #FIELDB (N5)           /* RECEIVING FIELD OF VAL FUNCTION
1 #DATE   (A10)          /* INPUT FIELD FOR DATE
END-DEFINE
*
INPUT #DATE #FIELDA
IF #DATE IS(D)
  IF #FIELDA IS (N5)
    COMPUTE #FIELDB = VAL(#FIELDA)
    WRITE NOTITLE 'VAL FUNCTION OK' //  '=' #FIELDA '=' #FIELDB
  ELSE
    REINPUT 'FIELD DOES NOT FIT INTO N5 FORMAT'
            MARK *#FIELDA
  END-IF
ELSE
  REINPUT 'INPUT IS NOT IN DATE FORMAT (YY-MM-DD) '
          MARK *#DATE
END-IF
*
END
```

54 論理条件基準

このchapterでは、ステートメント FIND、READ、HISTOGRAM、ACCEPT／REJECT、IF、DECIDE FOR、および REPEAT で使用できる、論理条件基準の用途と使用方法について説明します。

次のトピックについて説明します。

はじめに

基本の条件は、1 つの関係式です。 複数の関係式を論理演算子（AND、OR）と組み合わせて、複合条件を構成することができます。

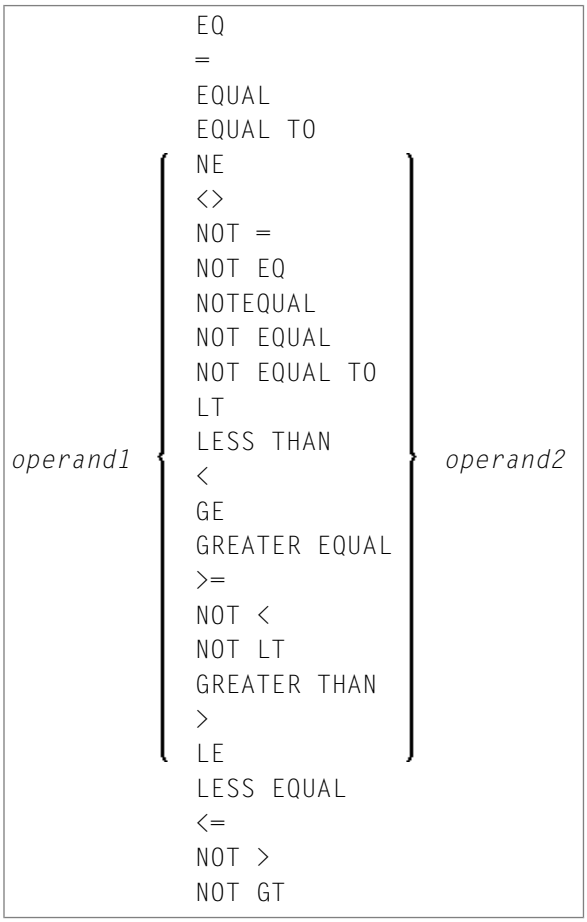
また、演算式を使用して、1 つの関係式を構成することもできます。

論理条件基準は、以下のステートメントで使用できます。

ステートメント	使用方法
FIND	論理条件基準を持つ WHERE 節を使用して、WITH 節で指定されている基本選択条件に、さらに条件を追加します。 WHERE 節で指定されている論理条件基準は、レコードの選択および読み込み後に評価されます。 WITH 節では、論理条件基準ではなく、（FIND ステートメントで記述されている）「基本検索条件」 が使用されます。
READ	論理条件基準を持つ WHERE 節を使用して、読み込んだレコードを処理するかどうかを指定します。 論理条件基準は、レコードの読み込み後に評価されます。
HISTOGRAM	論理条件基準を持つ WHERE 節を使用して、読み込んだ値を処理するかどうかを指定します。 論理条件基準は、値の読み込み後に評価されます。
ACCEPT/REJECT	FIND、READ、または HISTOGRAM ステートメントでレコードが選択され、読み込まれる場合、IF 節を ACCEPT または REJECT ステートメントで使用して、指定された条件に加えて論理条件の基準を指定することもできます。論理条件の基準は、レコードが読み込まれ、レコードの処理が開始した後に評価されます。
IF	論理条件の基準を使用して、ステートメントの実行を制御します。
DECIDE FOR	論理条件の基準を使用して、ステートメントの実行を制御します。
REPEAT	REPEAT ステートメントの UNTIL 節または WHILE 節に、いつ処理ループを終了するのかを決定する論理条件基準を指定します。

関係式

構文：



オペランド定義テーブル：

オペランド	構文要素						フォーマット														参照	ダイナミック定義
operand1	C	S	A		N	E	A	U	N	P	I	F	B	D	T	L		G	O		可	可
operand2	C	S	A		N	E	A	U	N	P	I	F	B	D	T	L		G	O		可	不可

上記オペランド定義テーブルの詳細については、『ステートメント』ドキュメントの「*構文記号およびオペランド定義テーブル*」を参照してください。

上記の表の "構文要素" の "E" は、算術式を表します。つまり、関係式のオペランドとして算術式を指定できることを意味します。 算術式の詳細については、COMPUTE ステートメントの説明にある *arithmetic-expression* を参照してください。

関係式の例：

```
IF NAME = 'SMITH'
IF LEAVE-DUE GT 40
IF NAME = #NAME
```

関係式における配列の比較については、「*配列の処理*」を参照してください。

Note: 浮動小数点のオペランドを使用すると、浮動小数点で比較が実行されます。浮動小数点数自体の精度に制限があるため、数値と浮動小数点形式との変換を行うときに、切り上げ／切り捨てエラーを回避することはできません。

論理条件における算術式

以下の例は、論理条件で算術式をどのように使用するかを示しています。

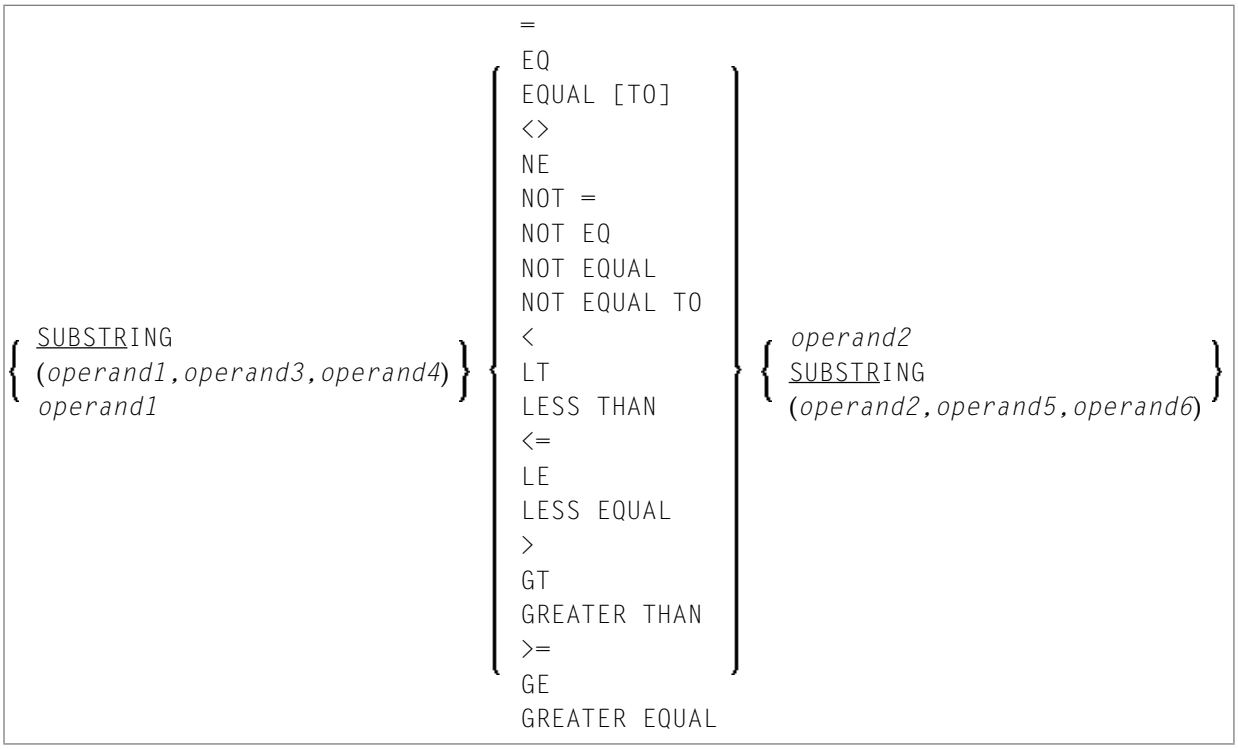
```
IF #A + 3 GT #B - 5 AND #C * 3 LE #A + #B
```

論理条件におけるハンドル

関係式のオペランドがハンドルの場合、演算子 EQUAL および NOT EQUAL のみを使用できます。

関係式における **SUBSTRING** オプション

構文：



オペランド定義テーブル：

オペランド	構文要素						フォーマット												参照	ダイナミック定義
operand1	C	S	A		N		A	U				B							可	可
operand2	C	S	A		N		A	U				B							可	不可
operand3	C	S							N	P	I	B							可	不可
operand4	C	S							N	P	I								可	不可
operand5	C	S							N	P	I								可	不可
operand6	C	S							N	P	I								可	不可

SUBSTRING オプションを使用すると、英数字、バイナリ、または Unicode の各フィールドの一部を比較できます。 フィールド名（*operand1*）の後にまず開始位置（*operand3*）を指定し、次に、フィールドの比較する部分の長さ（*operand4*）を指定します。

また、フィールド値を別のフィールド値の一部と比較することもできます。 フィールド名（*operand2*）の後にまず開始位置（*operand5*）を指定し、次に、フィールド *operand1* の比較する部分の長さ（*operand6*）を指定します。

両方の形式を組み合わせることもできます。つまり、SUBSTRING を *operand1* と *operand2* の両方に指定できます。

例：

以下の式は、フィールド #A の値の 5～12 桁目とフィールド #B の値を比較しています。

```
SUBSTRING(#A,5,8) = #B
```

- 5 が開始位置で、8 が長さです。

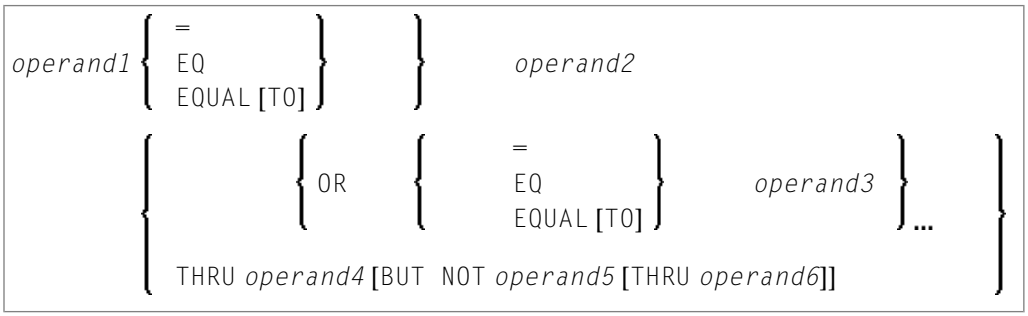
以下の式は、フィールド #A の値とフィールド #B の値の 3～6 桁目を比較しています。

```
#A = SUBSTRING(#B,3,4)
```

Note: *operand3*／*operand5* を省略すると、開始位置は "1" とみなされます。 *operand4*／*operand6* を省略すると、長さはフィールドの開始位置から終わりまでとみなされます。

拡張関係式

構文：



オペランド定義テーブル：

オペランド	構文要素						フォーマット													参照	ダイナミック定義
operand1	C	S	A		N*	E	A	U	N	P	I	F	B	D	T			G	O	可	不可
operand2	C	S	A		N*	E	A	U	N	P	I	F	B	D	T			G	O	可	不可
operand3	C	S	A		N*	E	A	U	N	P	I	F	B	D	T			G	O	可	不可
operand4	C	S	A		N*	E	A	U	N	P	I	F	B	D	T			G	O	可	不可
operand5	C	S	A		N*	E	A	U	N	P	I	F	B	D	T			G	O	可	不可
operand6	C	S	A		N*	E	A	U	N	P	I	F	B	D	T			G	O	可	不可

* 算術関数およびシステム変数は使用できますが、 ブレイク関数は使用できません。

operand3 は、以下のように、MASK オプションまたは SCAN オプションを使用して指定することもできます。

```
MASK (mask-definition) [operand]
MASK operand
SCAN operand
```

これらのオプションの詳細については、「*MASK オプション*」および「*SCAN オプション*」を参照してください。

例：

```
IF #A = 2 OR = 4 OR = 7
IF #A = 5 THRU 11 BUT NOT 7 THRU 8
```

MASK オプション

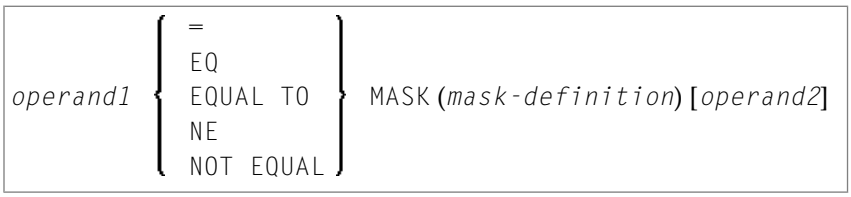
MASK オプションを使用すると、フィールドの特定の位置を特定の内容でチェックできます。

以下では次のトピックについて説明します。

- 定数マスク
- 変数マスク
- マスク内の文字
- マスク長
- 日付チェック
- 定数または変数の内容に対するチェック
- 範囲チェック
- パック型数値データまたはアンパック型数値データのチェック

定数マスク

構文：



オペランド定義テーブル：

オペランド	構文要素						フォーマット											参照	ダイナミック定義
operand1	C	S	A		N		A	U	N	P								可	不可
operand2	C	S					A	U	N	P			B					可	不可

Operand2 は、*mask-definition* に少なくとも 1 つの "X" が指定されている場合にのみ使用できます。 *Operand1* および *operand2* は、互換性のあるフォーマットである必要があります。

- *operand1* がフォーマット A の場合、*operand2* はフォーマット A、B、N、または U である必要があります。
- *operand1* がフォーマット U の場合、*operand2* はフォーマット A、B、N、または U である必要があります。

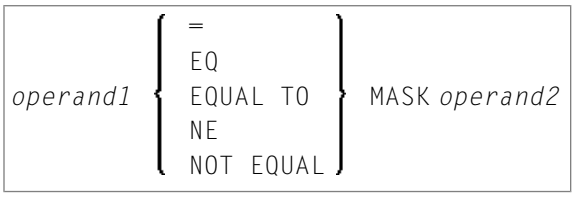
- *operand1* がフォーマット N または P の場合、*operand2* はフォーマット N または P である必要があります。

mask-definition 内の "X" は、*operand1* および *operand2* の対応する位置を選択して比較します。

変数マスク

定数 *mask-definition*（上記参照）の他に、変数マスク定義を使用することもできます。

構文：



オペランド定義テーブル：

オペランド	構文要素						フォーマット											参照	ダイナミック定義
operand1	C	S	A		N		A	U	N	P								可	不可
operand2		S					A	U										可	不可

operand2 の内容はマスク定義とみなされます。 *operand2* 内の末尾の空白は無視されます。

- *operand1* がフォーマット A、N、または P の場合、*operand2* はフォーマット A である必要があります。
- *operand1* がフォーマット U の場合、*operand2* はフォーマット U である必要があります。

マスク内の文字

マスク定義（定数マスクのマスク定義、および変数マスクの *operand2*）には、以下の文字を使用できます。

文字	意味
. または ? または _	ピリオド、疑問符、または下線は、該当する 1 桁をチェックしないことを示します。
* または %	アスタリスクまたはパーセント記号は、任意の桁数をチェックしないことを示すために使用します。

文字	意味
/	スラッシュは、値が特定の文字（または文字列）で終わっているかどうかをチェックするために使用します。 例えば、フィールドの最後が "E" であるか、または最後の "E" の後が空白以外に何もない場合、以下の条件は真になります。 `IF #FIELD = MASK (*'E'/)`
A	該当する桁が英字（大文字または小文字）かどうかをチェックします。
'*c*'	1つ以上の桁が、アポストロフィで囲まれている文字列かどうかをチェックします。アポストロフィを2つ使用すると、単一のアポストロフィをチェックできます。*operand1*がUnicode フォーマットの場合、'*c*' には Unicode 文字を使用する必要があります。
C	該当する桁が英字（大文字または小文字）、数字、または空白かどうかをチェックします。
DD	該当する 2 桁が有効な日の表記（01～31。MM および YY/YYYY が指定されていれば、その値に依存。「*日付チェック*」も参照）かどうかをチェックします。
H	該当する桁が 16 進数表記（A～F、0～9）かどうかをチェックします。
JJJ	該当する桁が有効なユリウス日、つまり年における日数（001～366。YY/YYYY が指定されていれば、その値に依存。「*日付チェック*」も参照）かどうかをチェックします。
L	該当する桁が小文字の英字（a～z）かどうかをチェックします。
MM	該当する桁が有効な月（01～12）かどうかをチェックします。「*日付チェック*」も参照してください。
N	該当する桁が数値かどうかをチェックします。
n...	1 つ（以上）の桁が 0～*n* の範囲内の数値かどうかをチェックします。
*n*1-*n*2 または *n*1:*n*2	該当する桁が *n*1～*n*2 の範囲内の数値かどうかをチェックします。 *n*1 および *n*2 は同じ長さである必要があります。
P	該当する桁が表示可能な文字（U、L、N、または S）かどうかをチェックします。
S	該当する桁が特殊文字かどうかをチェックします。『*オペレーション*』ドキュメントの「*NATCONV.INI での異なる文字セットのサポート*」も参照してください。
U	該当する桁が大文字の英字（A～Z）かどうかをチェックします。
X	該当する桁が、マスク定義に続く値（*operand2*）の同じ位置の値と同じかどうかをチェックします。 "X" は、変数マスク定義では意味も持たないため、使用できません。
YY	該当する 2 桁が有効な年（00～99）かどうかをチェックします。「*日付チェック*」も参照してください。
YYYY	該当する 4 桁が有効な年（0000～2699）かどうかをチェックします。

文字	意味
Z	該当する桁の左側のハーフバイトが 16 進数の 3 または 7で、右側のハーフバイトが 16 進数の 0〜9 かどうかをチェックします。 これは、負数の数値に対しても正しくチェックできます。 数値の符号は最後の桁に格納されるため、その桁は 16 進数の非数値とみなされます。したがって、（該当する桁が数値かどうかをチェックする）"N" では、負数の数値を正しくチェックできません。 マスク内では "Z" は、チェック対象の一連の数字ごとに一度だけ使用します。

マスク長

マスクの長さにより、チェックが必要な位置の数が決まります。

例：

```
DEFINE DATA LOCAL
1 #CODE (A15)
END-DEFINE
...
IF #CODE = MASK (NN'ABC'....NN)
...
```

上記の例では、#CODE の最初の 2 桁に対し、数値かどうかをチェックしています。 その後の 3 桁は、定数 "ABC" かどうかをチェックしています。 その次の 4 桁はチェックしていません。 10 桁目および 11 桁目は、数値かどうかをチェックしています。 12〜15 桁目はチェックしていません。

日付チェック

指定するマスク内でチェックできる日付は 1 つのみです。

日付に対するチェックでマスクに日（DD）を指定して月（MM）を指定しない場合、現在の月を想定してチェックが行われます。

日付に対するチェックでマスクに日（DD）またはユリウス日（JJJ）を指定して年（YY または YYYY）を指定しない場合、現在の年を想定してチェックが行われます。

日付を 2 桁の年（YY）でチェックするときにスライディングウィンドウも固定ウィンドウも設定されていない場合、現在の世紀を想定してチェックが行われます。 スライディングウィンドウまたは固定ウィンドウの詳細については、『パラメータリファレンス』に記載されている、プロファイルパラメータ YSLW の説明を参照してください。

例 **1**：

```
MOVE 1131 TO #DATE (N4)
IF #DATE = MASK (MMDD)
```

この例では、月と日の有効性をチェックしています。 月に対する値（11）は有効とみなされますが、11 月には 30 日しかないため、日に対する値（31）は無効とみなされます。

例 **2**：

```
IF #DATE(A8) = MASK (MM'/'DD'/'YY)
```

この例では、フィールド #DATE の内容がフォーマット MM/DD/YY（月／日／年）の日付として有効かどうかをチェックしています。

例 **3**：

```
IF #DATE (A4) = MASK (19-20YY)
```

この例では、フィールド #DATE の内容が 19～20 の範囲内の 2 桁の数字とそれに続く有効な 2 桁の年（00～99）かどうかをチェックしています。 Natural では、上記の方法で世紀を割り当てています。

Note: 明白なことですが、上記のマスクでは、年の整合性とは関係なく数値範囲 19～20 をチェックしているため、1900～2099 の範囲内で有効な年をチェックすることはできません。 年の範囲をチェックするには、日付の整合性をチェックするコードと範囲の有効性をチェックするコードを別個に記述します。

```
IF #DATE (A10) = MASK (YYYY'-'MM'-'DD) AND #DATE = MASK (19-20)
```

定数または変数の内容に対するチェック

マスクチェックに対する値として定数または変数を使用する場合、この値（*operand2*）は *mask-definition* の直後に指定する必要があります。

Operand2 は、少なくともマスクと同じ長さである必要があります。

マスクでは、チェックする桁には "X"、チェックしない桁には "."（または "?" または "_"）を指定します。

例：

```
DEFINE DATA LOCAL
1 #NAME (A15)
END-DEFINE
...
IF #NAME = MASK (..XX) 'ABCD'
...
```

上記の例では、フィールド #NAME の 3～4 桁目が "CD" かどうかをチェックしていいます。 1 桁目および 2 桁目はチェックされません。

マスクの長さにより、チェックが必要な位置の数が決まります。 マスク操作で使用する任意のフィールドまたは定数に対し、マスクは左詰めで適用されます。 式の右側のフィールド（または定数）のフォーマットは、式の左側のフィールドのフォーマットと同じである必要があります。

チェックするフィールド（*operand1*）がフォーマット A の場合、使用する定数（*operand2*）はアポストロフィで囲む必要があります。 フィールドが数値の場合、使用する値は数値定数、数値データベースフィールドの値、またはユーザー定義変数である必要があります。

どちらの場合でも、マスク内の "X" インジケータが示す位置に該当しない文字／数値は無視されます。

指定された位置の値が両方とも同じ場合、MASK 操作の結果は真になります。

例：

```
** Example 'LOGICX01': MASK option in logical condition
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
END-DEFINE
*
HISTOGRAM EMPLOY-VIEW CITY
  IF CITY =
MASK (....XX) '....NN'


    DISPLAY NOTITLE CITY *NUMBER
  END-IF
END-HISTOGRAM
*
END
```

上記の例では、フィールド CITY の 5～6 桁目が両方とも文字 "N" の場合、当該レコードは受け入れられます。

範囲チェック

範囲チェックを実行する場合、指定した変数を検証する桁数は、マスクに指定する値の精度で定義します。 例えば、"(...193...)" というマスクを指定すると、4〜6 桁目が 000〜193 の範囲内の 3 桁の数値かどうかが検証されます。

マスク定義の他の例

- 以下の例では、#NAME の各文字が英字かどうかがチェックされます。

```
IF #NAME (A10) = MASK (AAAAAAAAAA)
```

- 以下の例では、#NUMBER の 4〜6 桁目が数値かどうかがチェックされます。

```
IF #NUMBER (A6) = MASK (...NNN)
```

- 以下の例では、#VALUE の 4〜6 桁目が値 "123" かどうかがチェックされます。

```
IF #VALUE(A10) = MASK (...'123')
```

- 以下の例では、#LICENSE が "NY-" で始まるライセンス番号で、最後の 5 文字が #VALUE の最後の 5 文字と同じかどうかがチェックされます。

```

DEFINE DATA LOCAL
1 #VALUE(A8)
1 #LICENSE(A8)
END-DEFINE
INPUT 'ENTER KNOWN POSITIONS OF LICENSE PLATE:' #VALUE
IF #LICENSE = MASK ('NY-'XXXXX) #VALUE
```

- 以下の条件は、"NAT" で始まり "AL" で終わる値は、"NAT" と "AL" の間に他の文字が何文字あっても、すべて真になります。これには、"NATAL" だけでなく、"NATURAL" や "NATIONALITY" などの値も含まれます。

```
MASK('NAT'*'AL')
```

パック型数値データまたはアンパック型数値データのチェック

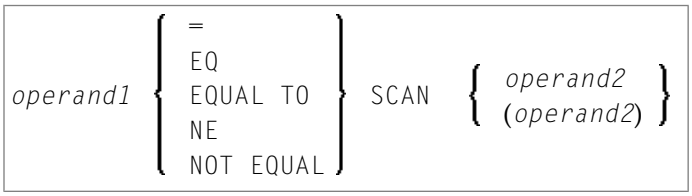
レガシアプリケーションでは、英数字フィールドやバイナリフィールドを再定義したパック型またはアンパック型の数値変数を頻繁に使用しています。 パック型変数またはアンパック型変数を割り当てや計算で使用すると、エラーや予測できない結果が生じる可能性があるため、このような再定義はしないことをお勧めします。再定義した変数を使用する前に内容を検証するには、桁数 - 1 個の N オプション（「マスク内の文字」を参照）を使用し、その後に一度だけ単一の Z オプションを指定します。

例：

```
IF #P1 (P1) = MASK (Z)
IF #N4 (N4) = MASK (NNNZ)
IF #P5 (P5) = MASK (NNNNZ)
```

SCAN オプション

構文：



オペランド定義テーブル：

オペランド	構文要素						フォーマット												参照	ダイナミック定義
operand1	C	S	A		N		A	U	N	P									可	不可
operand2	C	S					A	U					B*						可	不可

* *operand1* がフォーマット A または U の場合にのみ、*operand2* にバイナリを使用できます。*operand1* がフォーマット U で、*operand2* がフォーマット B の場合、*operand2* の長さは偶数である必要があります。

SCAN オプションは、フィールド内の特定の値をスキャンするために使用します。

SCAN オプション（*operand2*）に使用する文字は、英数字定数または Unicode 定数（アポストロフィで区切られた文字列）、英数字データベースフィールドまたは Unicode データベースフィールドの値、あるいはユーザー定義変数として指定する必要があります。

Caution: *operand1* および *operand2* の末尾の空白は、自動的に除去されます したがって、SCAN オプションは、末尾に空白のある値のスキャンには使用できません。 *operand1* および *operand2* の先頭または途中の空白は保持されます。 *operand2* がすべて空白の場合、*operand1* の値に関係なく、スキャンは成功とみなされます。スキャン操作で末尾の空白をスキャン対象とする場合は、EXAMINE FULL ステートメントの使用を検討してください。

スキャンするフィールド（*operand1*）には、フォーマット A、N、P、または U を使用できます。 SCAN 操作には、EQ（等しい）または NE（等しくない）の各演算子を指定できます。

SCAN 操作に使用する文字列の長さは、スキャンされるフィールドの長さ未満である必要があります。 指定した文字列の長さが、スキャンするフィールドの長さと同じ場合、SCAN ではなく EQUAL 演算子を使用する必要があります。

SCAN オプションの例：

```
** Example 'LOGICX02': SCAN option in logical condition
************************************************************************
DEFINE DATA
LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
*
1 #VALUE   (A4)
1 #COMMENT (A10)
END-DEFINE
*
INPUT 'ENTER SCAN VALUE:' #VALUE
LIMIT 15
*
HISTOGRAM EMPLOY-VIEW FOR NAME
  RESET #COMMENT
  IF NAME = SCAN #VALUE
    MOVE 'MATCH' TO #COMMENT
  END-IF
  DISPLAY NOTITLE NAME *NUMBER #COMMENT
END-HISTOGRAM
*
END
```

プログラム LOGICX02 の出力：

```
ENTER SCAN VALUE:
```

15 件の名前を "LL" でスキャンした結果、3 件一致した例：

```
        NAME           NMBR      #COMMENT
-------------------- --------- ----------

ABELLAN                      1 MATCH
ACHIESON                     1
ADAM                         1
ADKINSON                     8
AECKERLE                     1
AFANASSIEV                   2
AHL                          1
AKROYD                       1
ALEMAN                       1
ALESTIA                      1
ALEXANDER                    5
ALLEGRE                      1 MATCH
ALLSOP                       1 MATCH
ALTINOK                      1
ALVAREZ                      1
```

論理条件基準における BREAK

BREAKオプションでは、フィールドの現在の値または値の一部と、処理ループで前回通過した、同じフィールドの値とを比較できます。

構文：

```
BREAK [OF] operand1 [/n/]
```

オペランド定義テーブル：

オペランド	構文要素					フォーマット												参照	ダイナミック定義
operand1		S				A	U	N	P	I	F	B	D	T	L			可	不可

構文要素の説明：

operand1	チェックするコントロールフィールドを指定します。 配列の特定のオカレンスをコントロールフィールドとして使用することもできます。
lnl	-表記 /*n*/ を使用すると、値の変化を調べるためにチェックされるのは、コントロールフィールドの（左から右へ数えて）最初の *n* 個の位置だけであることを示すことができます。 この表記は、フォーマット A、B、N、または P のオペランドに対してのみ使用できます。 指定した位置のフィールドの値が変わると、BREAK 操作の結果は真になります。 AT END OF DATA 条件が発生すると、BREAK 操作の結果は偽になります。 例： 以下の例では、フィールド FIRST-NAME の最初の文字が変わっているかどうかがチェックされます。 `BREAK FIRST-NAME /1/` このオプションには、Natural システム関数を使用できません（AT BREAK ステートメントでは使用可能）。

BREAK オプションの例：

```
** Example 'LOGICX03': BREAK option in logical condition
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 BIRTH
*
1 #BIRTH (A8)
END-DEFINE
*
LIMIT 10
READ EMPLOY-VIEW BY BIRTH
  MOVE EDITED BIRTH (EM=YYYYMMDD) TO #BIRTH
  /*
  IF BREAK OF #BIRTH /6/
    NEWPAGE IF LESS THAN 5 LINES LEFT
    WRITE / '-' (50) /
  END-IF
  /*
  DISPLAY NOTITLE BIRTH (EM=YYYY-MM-DD) NAME FIRST-NAME
END-READ
END
```

プログラム LOGICX03 の出力：

```
  DATE            NAME              FIRST-NAME
   OF
 BIRTH
---------- -------------------- --------------------

1940-01-01 GARRET               WILLIAM
1940-01-09 TAILOR               ROBERT
1940-01-09 PIETSCH              VENUS
1940-01-31 LYTTLETON            BETTY

-------------------------------------------------

1940-02-02 WINTRICH             MARIA
1940-02-13 KUNEY                MARY
1940-02-14 KOLENCE              MARSHA
1940-02-24 DILWORTH             TOM

-------------------------------------------------

1940-03-03 DEKKER               SYLVIA
1940-03-06 STEFFERUD            BILL
```

IS オプション - 値のフォーマットおよび長さのチェック

構文：

operand1 IS (*format*)

このオプションは、英数字フィールドまたは Unicode フィールド（*operand1*）の値を特定の別のフォーマットに変換できるかどうかをチェックするために使用します。

オペランド定義テーブル：

オペランド	構文要素						フォーマット											参照	ダイナミック定義
operand1	C	S	A		N		A	U										可	不可

チェックできる *format* は、以下のとおりです。

N*ll.ll*	長さが *ll.ll* の数値
F*ll*	長さが *ll* の浮動小数点
D	日付。 *dd-mm-yy*、*dd-mm-yyyy*、*ddmmyyyy*（*dd* = 日、*mm* = 月、*yy* または *yyyy* = 年）の各日付フォーマットを使用できます。年、月、日の各コンポーネントの順序、およびコンポーネント間のデリミタ文字は、プロファイルパラメータ DTFORM（『パラメータリファレンス』を参照）によって決まります。
T	時刻（デフォルトの時刻表示フォーマットに依存）
P*ll.ll*	長さが *ll.ll* のパック型数値
I*ll*	長さが *ll* の整数値

チェック時は、*operand1* の先頭および末尾の空白は無視されます。

IS オプションは、例えば、算術関数 VAL（英数字フィールドから数値を抽出）を実行する前にフィールドの内容をチェックして、ランタイムエラーが発生しないようにするために使用できます。

Note: IS オプションで可能なのは、英数字フィールドの値が特定の「フォーマット」かどうかのチェックではなく、その「フォーマット」に*変換*できるかどうかのチェックです。 値が特定のフォーマットかどうかをチェックするには、MASK オプションを使用します。

IS オプションの例：

```
** Example 'LOGICX04': IS option as format/length check
************************************************************************
DEFINE DATA LOCAL
1 #FIELDA (A10)          /* INPUT FIELD TO BE CHECKED
1 #FIELDB (N5)           /* RECEIVING FIELD OF VAL FUNCTION
1 #DATE   (A10)          /* INPUT FIELD FOR DATE
END-DEFINE
*
INPUT #DATE #FIELDA
IF #DATE IS(D)
  IF #FIELDA IS (N5)
    COMPUTE #FIELDB = VAL(#FIELDA)
    WRITE NOTITLE 'VAL FUNCTION OK' //  '=' #FIELDA '=' #FIELDB
  ELSE
    REINPUT 'FIELD DOES NOT FIT INTO N5 FORMAT'
            MARK *#FIELDA
  END-IF
ELSE
```

```
  REINPUT 'INPUT IS NOT IN DATE FORMAT (YY-MM-DD) '
          MARK *#DATE
END-IF
*
END
```

プログラム LOGICX04 の出力：

```
#DATE 150487    #FIELDA









INPUT IS NOT IN DATE FORMAT (YY-MM-DD)
```

論理変数の評価

構文：

operand1

このオプションは、論理変数（フォーマットL）はとともに使用します。論理変数の値は、"TRUE" または "FALSE" になります。 *operand1* には、使用する論理変数名を指定します。

オペランド定義テーブル：

オペランド	構文要素			フォーマット	参照	ダイナミック定義
operand1	C	S	A	L	不可	不可

論理変数の例：

```
** Example 'LOGICX05': Logical variable in logical condition
************************************************************************
DEFINE DATA LOCAL
1 #SWITCH (L)  INIT <true>
1 #INDEX  (I1)
END-DEFINE
*
```

```
FOR #INDEX 1 5
  WRITE NOTITLE #SWITCH (EM=FALSE/TRUE) 5X 'INDEX =' #INDEX
  WRITE NOTITLE #SWITCH (EM=OFF/ON) 7X 'INDEX =' #INDEX
  IF #SWITCH
    MOVE FALSE TO #SWITCH
  ELSE
    MOVE TRUE TO #SWITCH
  END-IF
  /*
  SKIP 1
END-FOR
END
```

プログラム LOGICX05 の出力：

```

TRUE     INDEX =    1
ON       INDEX =    1

FALSE    INDEX =    2
OFF      INDEX =    2

TRUE     INDEX =    3
ON       INDEX =    3

FALSE    INDEX =    4
OFF      INDEX =    4

TRUE     INDEX =    5
ON       INDEX =    5
```

MODIFIED オプション

構文：

operand1 [NOT] MODIFIED

このオプションは、属性がダイナミックに割り当てられているフィールドの内容が、INPUT ステートメントの実行中に変更されているかどうかを判断するために使用します。

オペランド定義テーブル：

オペランド	構文要素	フォーマット	参照	ダイナミック定義
operand1	S A	C	不可	不可

INPUT ステートメント内で参照される属性制御変数には、端末にマップが転送される時点では常にステータス "NOT MODIFIED" が割り当てられています。

属性制御変数を参照しているフィールドの内容が変更されると、属性制御変数にステータス "MODIFIED" が割り当てられます。複数のフィールドが同じ属性制御変数を参照している場合、これらのフィールドのどれかが変更されると、この変数は "MODIFIED" に設定されます。

operand1 が配列の場合、配列要素の少なくとも 1 つにステータス "MODIFIED" が割り当てられていると、結果は真になります（OR 演算）。

MODIFIED オプションの例：

```
** Example 'LOGICX06': MODIFIED option in logical condition
************************************************************************
DEFINE DATA LOCAL
1 #ATTR (C)
1 #A    (A1)
1 #B    (A1)
END-DEFINE
*
MOVE (AD=I) TO #ATTR
*
INPUT (CV=#ATTR) #A #B
IF #ATTR NOT MODIFIED
  WRITE NOTITLE 'FIELD #A OR #B HAS NOT BEEN MODIFIED'
END-IF
*
IF #ATTR MODIFIED
  WRITE NOTITLE 'FIELD #A OR #B HAS BEEN MODIFIED'
END-IF
*
END
```

プログラム LOGICX06 の出力：

```
#A    #B
```

任意の値を入力して Enter キーを押すと、以下のような出力が表示されます。

```
FIELD #A OR #B HAS BEEN MODIFIED
```

SPECIFIED オプション

構文：

operand1 [NOT] SPECIFIED

このオプションは、呼び出されたオブジェクト（サブプログラム、外部サブルーチン、ダイアログ、または ActiveX コントロール）のオプションパラメータが、呼び出し元オブジェクトから値を受け取っているかどうかをチェックするために使用します。

オプションパラメータとは、呼び出されるオブジェクトの DEFINE DATA PARAMETER ステートメント内でキーワード OPTIONAL を使用して定義されているフィールドのことです。フィールドが OPTIONAL として定義されている場合、呼び出し元オブジェクトからこのフィールドに値を渡すことはできますが、必須ではありません。

呼び出し元ステートメントでは、表記 *n*X を使用して、パラメータに値を渡さないことを示します。

値を受け取っていないオプションパラメータを処理しようとすると、ランタイムエラーが発生します。 このようなエラーを回避するには、呼び出されたオブジェクト内で SPECIFIED オプションを使用してオプションパラメータに値が渡されているかどうかを確認し、値が渡されている場合にのみ処理するようにします。

Parameter-name は、呼び出されたオブジェクトの DEFINE DATA PARAMETER ステートメントで指定されているパラメータ名です。

フィールドが OPTIONAL として定義されていない場合、SPECIFIED 条件は常に "TRUE" になります。

SPECIFIED オプションの例：

呼び出し元プログラム：

```
** Example 'LOGICX07': SPECIFIED option in logical condition
************************************************************************
DEFINE DATA LOCAL
1 #PARM1 (A3)
1 #PARM3 (N2)
END-DEFINE
*
#PARM1 := 'ABC'
#PARM3 := 20
*
CALLNAT 'LOGICX08' #PARM1 1X #PARM3
*
END
```

呼び出されたサブプログラム：

```
** Example 'LOGICX08': SPECIFIED option in logical condition
************************************************************************
DEFINE DATA PARAMETER
1 #PARM1 (A3)
1 #PARM2 (N2) OPTIONAL
1 #PARM3 (N2) OPTIONAL
END-DEFINE
*
WRITE '=' #PARM1
*
IF #PARM2 SPECIFIED
  WRITE '#PARM2 is specified'
  WRITE '=' #PARM2
ELSE
  WRITE '#PARM2 is not specified'
* WRITE '=' #PARM2                /* would cause runtime error NAT1322
END-IF
*
IF #PARM3 NOT SPECIFIED
  WRITE '#PARM3 is not specified'
ELSE
  WRITE '#PARM3 is specified'
  WRITE '=' #PARM3
END-IF
END
```

プログラム LOGICX07 の出力：

```
Page      1                                                  04-12-15  11:25:41

#PARM1: ABC
#PARM2 is not specified
#PARM3 is specified
#PARM3:  20
```

論理条件基準内で使用するフィールド

データベースフィールドおよびユーザー定義変数は、論理条件基準を構成するために使用できます。 マルチプルバリューフィールドであるデータベースフィールド、またはピリオディックグループに含まれているデータベースフィールドも使用できます。 マルチプルバリューフィールドに値の範囲が指定されている、またはピリオディックグループにオカレンスの範囲が指定されている場合、検索値がその範囲の値／オカレンス内で検出されると、条件は真になります。

使用する値はそれぞれ、式の反対側で使用されているフィールドと互換性がある必要があります。10進数表記は、数値フィールドに対する値にのみ指定します。また、値の小数部の桁数は、フィールドに定義されている小数部の桁数と一致させる必要があります。

オペランドのフォーマットが異なる場合、2番目のオペランドが最初のオペランドのフォーマットに変換されます。

Note: 小数点表記のない数値定数は、値が -2147483648～+2147483647 の場合はフォーマットIで格納されます。「*数値定数*」を参照してください。したがって、このような整数定数を *operand1* とする比較は、*operand2* を整数値に変換することにより実行されます。これは、*operand2* の小数点以下の値は切り捨てられるため、比較の対象とみなされないことを意味します。

例：

```
 IF 0 = 0.5    /* is true because 0.5 (operand2) is converted to 0 (format I of
operand1)
 IF 0.0 = 0.5  /* is false
```

```
IF 0.5 = 0    /* is false
IF 0.5 = 0.0  /* is false
```

以下の表に、論理条件で一緒に使用できるオペランドのフォーマットを示します。

operand1	*operand2*												
	A	**U**	**B*n* (*n*=<4)**	**B*n* (*n*>=5)**	**D**	**T**	**I**	**F**	**L**	**N**	**P**	**GH**	**OH**
A	○	○	○	○									
U	○	○	[2]	[2]									
B*n* (*n*=<4)	○	○	○	○	○	○	○	○		○	○		
B*n* (*n*>=5)	○	○	○	○									
D			○		○	○	○	○		○	○		
T			○		○	○	○	○		○	○		
I			○		○	○	○	○		○	○		
F			○		○	○	○	○		○	○		
L													
N			○		○	○	○	○		○	○		
P			○		○	○	○	○		○	○		
GH [1]												○	
OH [1]													○

Notes:

1. 1) ここで、GH は GUI ハンドルを表し、OH はオブジェクトハンドルを表します。
2. 2) バイナリ値には Unicode コードポイントが格納されているものとします。また、比較は、2 つの Unicode 値の比較として実行されます。 バイナリフィールドの長さは偶数である必要があります。

2つの値を英数字値として比較する場合、長い方の値と同じ長さにするために、末尾に空白を追加して短い方の値を拡張するものとします。

2つの値をバイナリ値として比較する場合、長い方の値と同じ長さにするために、先頭にバイナリゼロを挿入して短い方の値を拡張するものとします。

2 つの値を Unicode 値として比較する場合、両方の値の末尾の空白を削除してから、ICU 照合アルゴリズムを使用して 2 つの値を比較します。『*Unicode とコードページのサポート*』ドキュメントの「*論理条件の基準*」も参照してください。

比較の例：

```
A1(A1) := 'A'
A5(A5) := 'A    '
B1(B1) := H'FF'
B5(B5) := H'00000000FF'
U1(U1) := UH'00E4'
U2(U2) := UH'00610308'
IF A1 = A5 THEN ...                  /* TRUE
IF B1 = B5 THEN ...                  /* TRUE
IF U1 = U2 THEN ...                  /* TRUE
```

配列とスカラ値を比較する場合、配列の各要素がスカラ値と比較されます。 少なくとも 1 つの配列要素が条件を満たしていると、結果は真になります（OR 演算）。

配列と配列を比較する場合、配列の各要素が、もう一方の配列の対応する要素と比較されます。すべての要素の比較が条件を満たしている場合にのみ、結果は真になります（AND 演算）。

「*配列の処理*」も参照してください。

Note: Adabas フォネティックディスクリプタは、論理条件には使用できません。

論理条件基準の例：

```
FIND EMPLOYEES-VIEW WITH CITY = 'BOSTON' WHERE SEX = 'M'
READ EMPLOYEES-VIEW BY NAME WHERE SEX = 'M'
ACCEPT IF LEAVE-DUE GT 45
IF #A GT #B THEN COMPUTE #C = #A + #B
REPEAT UNTIL #X = 500
```

複雑な論理式における論理演算子

論理条件基準は、ブール演算子 AND、OR、および NOT を使用して組み合わせることができます。カッコを使用して論理グループを示すこともできます。

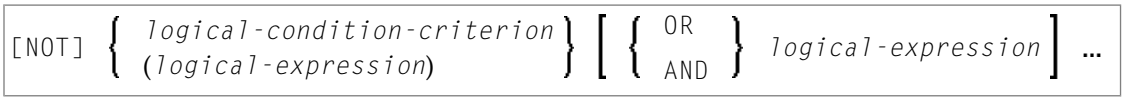
演算子は、以下の順序で評価されます。

優先順位	演算子	意味
1	( )	カッコ
2	NOT	否定
3	AND	AND 操作
4	OR	OR 操作

以下の*論理条件基準*を論理演算子で組み合わせることにより、複雑な*論理式*を作成できます。

- 関係式
- 拡張関係式
- MASK オプション
- SCAN オプション
- BREAK オプション

*論理式*の構文は以下のとおりです。



論理式の例：

```
FIND STAFF-VIEW WITH CITY = 'TOKYO'
     WHERE BIRTH GT 19610101 AND SEX = 'F'
IF NOT (#CITY = 'A' THRU 'E')
```

論理式における配列の比較については、「*配列の処理*」を参照してください。

Note: 複数の論理条件基準が AND で組み合わされている場合、それらの条件のいずれかが偽になると即座に評価は終了します。

55 演算割り当てのルール

このchapterでは、次のトピックについて説明します。

フィールドの初期化

算術演算のオペランドとして使用するフィールド（ユーザー定義変数またはデータベースフィールド）は、以下のフォーマットのいずれかで定義する必要があります。

フォーマット	
N	アンパック型数値
P	パック型数値
I	整数
F	浮動小数点
D	日付
T	時刻

Note: レポーティングモードの場合、算術演算のオペランドとして使用するフィールドをあらかじめ定義しておく必要があります。算術演算の結果フィールドとして使用するユーザー定義変数またはデータベースフィールドについては、あらかじめ定義しておく必要はありません。

DEFINE DATA ステートメントで定義されているユーザー定義変数およびデータベースフィールドはすべて、実行するためにプログラムが呼び出されるときに、ゼロまたは空白で適切に初期化されます。

データ転送

データ転送は、MOVE ステートメントまたは COMPUTE ステートメントを使用して実行します。 次の表は、オペランドで使用可能な、データ転送で互換性のあるフォーマットをまとめたものです。

送出側のフィールドフォーマット	受け取り側のフィールドフォーマット													
	N または P	A	U	B*n* (*n*<5)	B*n* (*n*>4)	I	L	C	D	T	F	G	O
N または P	○	[2]	[14]	[3]	-	○	-	-	-	○	○	-	-
A	-	○	[13]	[1]	[1]	-	-	-	-	-	-	-	-
U	-	[11]	○	[12]	[12]	-	-	-	-	-	-	-	-
B*n* (*n*<5)	[4]	[2]	[14]	[5]	[5]	○	-	-	-	○	○	-	-

B*n* (*n*>4)	-	[6]	[15]	[5]	[5]	-	-	-	-	-	-	-	-	
I	○	[2]	[14]	[3]	-	○	-	-	-	○	○	-	-	
L	-	[9]	[16]	-	-	-	○	-	-	-	-	-	-	
C	-	-	-	-	-	-	-	○	-	-	-	-	-	
D	○	[9]	[16]	○	-	○	-	-	○	[7]	○	-	-	
T	○	[9]	[16]	○	-	○	-	-	[8]	○	○	-	-	
F	○	[9] [10]	[10] [16]	[3]	-	○	-	-	-	○	○	-	-	
G	-	-	-	-	-	-	-	-	-	-	-	○	-	
O	-	-	-	-	-	-	-	-	-	-	-	-	○	

上記の意味は次に示すとおりです。

○	データ転送の互換性があることを示します。
-	データ転送の互換性がないことを示します。
[ ]	角カッコ [ ] は、下記の対応するデータ転送ルールを参照します。

データ変換

データ値の変換には、以下のルールが適用されます。

1. 英数字からバイナリ
   値は左から右に 1 バイトずつ移動します。 定義されている長さおよび指定されているバイト数に応じて、変換後の値は切り捨てられるか、または末尾に空白文字が追加されます。
2. （**N**、**P**、**I**）およびバイナリ（長さ **1**～**4**）から英数字
   値はアンパック形式に変換され、左詰めで英数字フィールドに転送されます。つまり、先行ゼロは除去され、フィールドには末尾の空白が充填されます。 負の数値に関しては、記号は 16 進法 "D*x*" に変換されます。 数値の小数点は無視されます。 小数点の前および後のすべての桁は、1 つの整数値として扱われます。
3. （**N**、**P**、**I**）からバイナリ（**1**～**4** バイト）
   数値はバイナリ（4 バイト）に変換されます。 数値の小数点は無視され、小数点の前後の値は 1 つの整数値として扱われます。 変換後のバイナリ値は、符号の値に応じて、正数または 2 の補数になります。
4. バイナリ（**1**～**4** バイト）から数値
   変換された値は右詰めで数値に割り当てられます。つまり、先行ゼロを含みます 長さが 1～3 バイトのバイナリ値は、常に正の符号を使用しているとみなされます。 長さが 4 バイトのバイナリ値は、最も左側のビットによって数値の符号が決まります（1 = 負数、0 = 正数）。受け取り側の数値の小数点は無視されます。 小数点の前および後のすべての桁は、1 つの整数値として扱われます。

5. バイナリからバイナリ
   値は右から左に 1 バイトずつ転送されます。 受け取りフィールドには、先頭にバイナリゼロが挿入されます。
6. バイナリ（**5** バイト以上）から英数字
   値は左から右に 1 バイトずつ移動します。 定義されている長さおよび指定されているバイト数に応じて、変換後の値は切り捨てられるか、または末尾に空白文字が追加されます。
7. 日付（**D**）から時刻（**T**）
   日付を時刻に転送する場合、時刻は 00:00:00:0 として変換されます。
8. 時刻（**T**）から日付（**D**）
   時刻を日付に転送する場合、時刻情報は切り捨てられ、日付情報のみが残されます。
9. **L**、**D**、**T**、**F** から **A**
   値は表示形式に変換され、左詰めで割り当てられます。
10. **F**
    短い英数字フィールドまたは Unicode フィールドに F を割り当てる場合、必要に応じて仮数が縮小されます。
11. **Unicode** から英数字
    Unicode 値は、International Components for Unicode（ICU）ライブラリを使用して、現在のコードページに基づいた英数字に変換されます。 定義された長さおよび指定されたバイト数に応じて、結果が切り捨てられるか、結果の末尾に空白文字が追加されます。
12. **Unicode** からバイナリ
    値は左から右にコード単位で転送されます。 定義された長さおよび指定されたバイト数に応じて、結果が切り捨てられるか、結果の末尾に空白文字が追加されます。 受け取り側のバイナリフィールドの長さは偶数である必要があります。
13. 英数字から **Unicode**
    英数字値は、International Components for Unicode（ICU）ライブラリを使用して、現在のコードページから Unicode 値に変換されます。 定義された長さおよび指定されたコード単位数に応じて、結果が切り捨てられるか、結果の末尾に空白文字が追加されます。
14. （**N**、**P**、**I**）およびバイナリ（長さ **1**～**4**）から **Unicode**
    値はアンパック形式に変換され、その値から先行ゼロを除去した英数字値が取得されます。 負の数値に関しては、記号は 16 進法 `Dx` に変換されます。 数値の小数点は無視されます。 小数点の前および後のすべての桁は、1 つの整数値として扱われます。 変換後の値が、英数字から Unicode に変換されます。 定義された長さおよび指定されたコード単位数に応じて、結果が切り捨てられるか、結果の末尾に空白文字が追加されます。
15. バイナリ（**5** バイト以上）から **Unicode**
    値は左から右に 1 バイトずつ移動します。 定義されている長さおよび指定されているバイト数に応じて、変換後の値は切り捨てられるか、または末尾に空白文字が追加されます。 送出側のバイナリフィールドの長さは偶数である必要があります。

16. **L**、**D**、**T**、**F** から **U**
    値は英数字の表示形式に変換されます。 変換後の値が、英数字から Unicode に変換され、左詰めで割り当てられます。

ソースとターゲットのフォーマットが同じ場合、定義されている長さと、指定されているバイト数（フォーマット A および B）またはコード単位（フォーマット U）に応じて、変換後の値は切り捨てられるか、あるいは末尾の空白（フォーマット A および U）または先頭のバイナリゼロ（フォーマット B）が追加されます。

「*ダイナミック変数の使用*」も参照してください。

フィールドの切り捨てと切り上げ

フィールドの切り捨ておよび切り上げには、以下のルールが適用されます。

- 数値フィールドの上位の切り捨ては、切り捨てる桁が先行ゼロの場合にのみ実行できます。 小数点以下の桁は、小数点が明示的に指定されているかどうかにかかわらず、切り捨てられます。
- 英数字フィールドの末尾の空白は切り捨てられます。
- オプション `ROUNDED` を指定すると、切り捨てられる値の一番上の桁が 5 以上の場合、結果の一番下の桁に切り上げられます。 除算の結果の精度については、「*算術演算結果の精度*」も参照してください。

算術演算結果のフォーマットと長さ

次の表は、算術演算の結果のフォーマットと長さを示しています。

	I1	I2	I4	N または P	F4	F8
I1	I1	I2	I4	P*	F4	F8
I2	I2	I2	I4	P*	F4	F8
I4	I4	I4	I4	P*	F4	F8
N または P	P*	P*	P*	P*	F4	F8
F4	F4	F4	F4	F4	F4	F8
F8	F8	F8	F8	F8	F8	F8

メインフレームコンピュータでは、算術演算結果の精度を向上させるため、フォーマット／長さ F4 の代わりに F8 が使用されます。

P* は、「*算術演算結果の精度*」に示されているように、各オペランドの整数の長さと精度に基づいて個別に決定されます。

フォーマット I に適用される、10 進数での桁数と有効な値を以下に示します。

フォーマット／長さ	10 進数での桁数	有効値
I1	3	-128～127
I2	5	-32,768～32,767
I4	10	-2,147,483,648～2,147,483,647

浮動小数点数を使用した算術演算

以下では次のトピックについて説明します。

- 全般的な考慮事項
- 浮動小数点数の精度
- 浮動小数点表現への変換
- プラットフォーム依存

全般的な考慮事項

浮動小数点数（フォーマット F）は整数（フォーマット I）と同様に 2 の累乗の和として表されますが、アンパック型およびパック型の数値（フォーマット N および P）は 10 の累乗の和として表されます。

アンパック型またはパック型の数値では、小数点の位置は固定です。一方、浮動小数点数では、小数点の位置は（その名が示すとおり）「浮動」 です。つまり、小数点の位置は固定されず、実際の値によって変わります。

浮動小数点数は、正弦関数や対数関数などの三角関数や算術関数の計算に不可欠です。

浮動小数点数の精度

浮動小数点数の性質から、精度には以下の制限があります。

- フォーマット／長さが F4 の変数の場合、精度は約 7 桁に制限されます。
- フォーマット／長さが F8 の変数の場合、精度は約 15 桁に制限されます。

これ以上大きな桁を持つ値は、浮動小数点数では正確に表せません。 小数点の前後に何桁あるかに関係なく、浮動小数点数で対応できるのはそれぞれ先頭の 7 桁または 15 桁のみです。

整数値は、絶対値が $2^{23} - 1$ を超えていなければ、フォーマット／長さが F4 の変数で正確に表すことができます。

浮動小数点表現への変換

英数字、アンパック型数値、またはパック型数値を浮動小数点フォーマットに変換する場合（割り当て操作など）、表現を変更、つまり、10 の累乗の和から 2 の累乗の和に変換する必要があります。

したがって、2の累乗の有限和で表現できる数値のみが正確に表されます。それ以外の数値は近似値としてのみ表されます。

例：

以下の数値は、正確な浮動小数点表現です。

$$1.25 = 2^{0} + 2^{-2}$$

以下の数値は、正確に表現できない浮動小数点表現です。

$$1.2 = 2^{0} + 2^{-3} + 2^{-4} + 2^{-7} + 2^{-8} + 2^{-11} + 2^{-12} + \ldots$$

このように、英数字、アンパック型数値、またはパック型数値から浮動小数点値への変換、およびその逆の変換では、小規模なエラーが発生する場合があります。

プラットフォーム依存

ハードウェアアーキテクチャが異なるため、浮動小数点数の表現はプラットフォームによって変わります。浮動小数点演算を行う同じアプリケーションを別のプラットフォームで実行すると、結果がわずかに異なるのはこのためです。 それぞれの表現によって、浮動小数点変数の有効な値の範囲も決まります。（おおよその）有効な値の範囲は、次のとおりです。

- $\pm 1.17 * 10^{-38}$ ～$\pm 3.40 * 10^{38}$ （F4 変数の場合）
- $\pm 2.22 * 10^{-308}$ ～$\pm 1.79 * 10^{308}$ （F8 変数の場合）

Note: 電卓で使用されている表現も、コンピュータで使用されている表現と異なっている可能性があります。したがって、同じ計算をしても違う結果が出る可能性があります。

日付および時刻を使用した算術演算

フォーマットD（日付）およびT（時刻）は、加算、減算、乗算、および除算でのみ使用できます。 乗算および除算は、加算および減算の中間結果にのみ使用できます。

日付／時刻値は、相互に加算／減算できます。また、整数値（小数なし）を日付／時刻に加算／減算することもできます。 これらの整数値は、フォーマット N、P、I、D、または T のフィールドに格納できます。

加算／減算の中間結果は、その後の演算で被乗数／被除数として使用することもできます。

日付値に加算／減算する整数値は、日数とみなされます。 時刻値に加算／減算する整数値は、1/10 秒とみなされます。

日付と時刻を使用した算術演算には、以下のようなNaturalの内部処理に基づく制限が適用されます。

Natural では内部的に、日付／時刻変数を使用した算術演算は、以下のように処理されます。

```
COMPUTE result-field = operand1 +/- operand2
```

上のステートメントは次のように解決されます。

1. *intermediate-result = operand1 +/- operand2*
2. *result-field = intermediate-result*

つまり、Natural は、最初の手順で加算／減算の結果を計算し、2 番目の手順でその結果を結果フィールドに割り当てます。

さらに複雑な算術演算も、次のように、同じパターンに従って解決されます。

```
COMPUTE result-field = operand1 +/- operand2 +/- operand3 +/- operand4
```

上のステートメントは次のように解決されます。

1. *intermediate-result1 = operand1 +/- operand2*
2. *intermediate-result2 = intermediate-result1 +/- operand3*
3. *intermediate-result3 = intermediate-result2 +/- operand4*
4. *result-field = intermediate-result3*

乗算および除算も、加算および減算と同様に解決されます。

このような *intermediate-result* の内部フォーマットは、以下の表に示すとおり、オペランドのフォーマットに依存します。

加算

以下の表は、加算（*intermediate-result* = *operand1* + *operand2*）の中間結果のフォーマットを示しています。

operand1 のフォーマット	*operand2* のフォーマット	*intermediate-result* のフォーマット
D	D	Di
D	T	T
D	Di、Ti、N、P、I	D
T	D、T、Di、Ti、N、P、I	T
Di、Ti、N、P、I	D	D
Di、Ti、N、P、I	T	T
Di、N、P、I	Di	Di
Ti、N、P、I	Ti	Ti
Di	Ti、N、P、I	Di
Ti	Di、N、P、I	Ti

減算

以下の表は、減算（*intermediate-result* = *operand1* - *operand2*）の中間結果のフォーマットを示しています。

operand1 のフォーマット	*operand2* のフォーマット	*intermediate-result* のフォーマット
D	D	Di
D	T	Ti
D	Di、Ti、N、P、I	D
T	D、T	Ti
T	Di、Ti、N、P、I	T
Di、N、P、I	D	Di
Di、N、P、I	T	Ti
Di	Di、Ti、N、P、I	Di
Ti	D、T、Di、Ti、N、P、I	Ti
N、P、I	Di、Ti	P12

乗算または除算

以下の表は、乗算（*intermediate-result* = *operand1* * *operand2*）または除算（*intermediate-result* = *operand1* / *operand2*）の中間結果のフォーマットを示しています

operand1 のフォーマット	*operand2* のフォーマット	*intermediate-result* のフォーマット
D	D、Di、Ti、N、P、I	Di
D	T	Ti
T	D、T、Di、Ti、N、P、I	Ti
Di	T	Ti
Di	D、Di、Ti、N、P、I	Di
Ti	D	Di
Ti	Di、T、Ti、N、P、I	Ti
N、P、I	D、Di	Di
N、P、I	T、Ti	Ti

内部割り当て

Di は内部日付フォーマットの値です。Ti は内部時刻フォーマットの値です。これらの値は、その後の日付／時刻の算術演算に使用できますが、フォーマット D の結果フィールドに割り当てることはできません（以下の割り当て表を参照）。

内部フォーマット Di または Ti の中間結果をその後の加算／減算／乗算／除算でオペランドとして使用する複雑な算術演算では、これらのフォーマットはそれぞれ D または T とみなされます。

以下の表は、どの中間結果をどの結果フィールドに内部的に割り当て（*result-field* = *intermediate-result*）できるのかを示しています。

result-field のフォーマット	*intermediate-result* のフォーマット	割り当て可能
D	D、T	○
D	Di、Ti、N、P、I	×
T	D、T、Di、Ti、N、P、I	○
N、P、I	D、T、Di、Ti、N、P、I	○

フォーマット D または T の結果フィールドに、負数を格納することはできません。

例 **1** および **2**（無効）：

```
COMPUTE DATE1 (D) = DATE2 (D) + DATE3 (D)
COMPUTE DATE1 (D) = DATE2 (D) - DATE3 (D)
```

加算／減算の中間結果はフォーマット D*i* ですが、フォーマット D*i* の値をフォーマット D の結果フィールドに割り当てることはできないため、これらの演算は実行できません。

例 **3** および **4**（無効）：

```
COMPUTE DATE1 (D) = TIME2 (T) - TIME3 (T)
COMPUTE DATE1 (D) = DATE2 (D) - TIME3 (T)
```

加算／減算の中間結果はフォーマット T*i* ですが、フォーマット T*i* の値をフォーマット D の結果フィールドに割り当てることはできないため、これらの演算は実行できません。

例 **5**（有効）：

```
COMPUTE DATE1 (D) = DATE2 (D) - DATE3 (D) + TIME3 (T)
```

この演算は可能です。まず、DATE3 が DATE2 から減算され、フォーマット D*i* の中間結果が生成されます。次に、この中間結果が TIME3 に加算され、フォーマット T の中間結果が生成されます。最後に、この 2 番目の中間結果が結果フィールド DATE1 に割り当てられます。

例 **6** および **7**（無効）：

```
COMPUTE DATE1 (D) = DATE2 (D) + DATE3 (D) * 2
COMPUTE TIME1 (T) = TIME2 (T) - TIME3 (T) / 3
```

中間結果ではなく日付／時刻フィールドを使用して乗算／除算を実行しようとしているため、これらの演算は実行できません。

例 **8**（有効）：

```
COMPUTE DATE1 (D) = DATE2 (D) + (DATE3(D) - DATE4 (D)) * 2
```

この演算は可能です。まず、DATE4 が DATE3 から減算され、フォーマット Di の中間結果が生成されます。次に、この中間結果が 2 で乗算され、フォーマット Di の中間結果が生成されます。この中間結果が DATE2 に加算され、フォーマット D の中間結果が生成されます。最後に、この 3 番目の中間結果が結果フィールド DATE1 に割り当てられます。

フォーマット T の値をフォーマット D のフィールドに割り当てる場合、時刻値に有効な日付コンポーネントが含まれていることを確認する必要があります。

フォーマット表現の混在に対するパフォーマンスの考慮事項

算術演算を実行する場合、フィールドフォーマットの選択はパフォーマンスに大きく影響します。

業務演算では、可能であれば、フォーマット I（整数）のフィールドのみを使用します。

科学演算では、可能であれば、フォーマット F（浮動小数点）のフィールドを使用します。

数値（N、P）と浮動小数点（F）のフォーマットが混在する式では、浮動小数点フォーマットへの変換が実行されます。 この変換は CPU にとってかなりの負荷となります。 したがって、算術演算ではフォーマット表現を混在させないことをお勧めします。

算術演算結果の精度

演算	整数部の桁数	小数部の桁数
加算／減算	Fi + 1 または Si + 1（どちらか大きい方）	Fd または Sd（どちらか大きい方）
乗算	Fi + Si + 2	Fd + Sd（最大 7）
除算	Fi + Sd	（下記参照）
累乗	15 - Fd	Fd
平方根	Fi	Fd

各項目の意味を次に示します。

F	第 1 オペランド
S	第 2 オペランド
R	結果
i	整数部の桁数
d	小数部の桁数

除算の結果に対する小数部の桁数

除算の結果の精度は、結果フィールドを使用できるかどうかによって変わります。

- 結果フィールドを使用できる場合、精度は Rd または Fd（どちらか大きい方）になります*。
- 結果フィールドを使用できない場合、精度は Fd または Sd（どちらか大きい方）になります。*。

* ROUNDED オプションを使用すると、実際に結果を四捨五入する前に、結果の精度が内部的に 1 桁増やされます。

結果フィールドは、COMPUTE ステートメント、DIVIDE ステートメント、および比較演算子の後に除算が使用されている論理条件（IF #A = #B / #C THEN ... など）では使用できる、または使用できるとみなされます。

結果フィールドは、比較演算子の前に除算が使用されている論理条件（IF #B / #C = #A THEN ... など）では使用できない、または使用できないとみなされます。

例外：

被除数と除数がともに整数フォーマットで、少なくともその1つが変数の場合、結果フィールドの精度および ROUNDED オプションが使用されているかどうかに関係なく、除算の結果は常に整数フォーマットになります。

算術式の結果の精度

算術式、例えば #A / (#B * #C) + #D * (#E - #F + #G) の精度は、処理順序に従って算術演算の結果を評価することにより導き出されます。 算術式の詳細については、COMPUTE ステートメントの説明にある *arithmetic-expression* を参照してください。

算術演算のエラー条件

加算、減算、乗算、および除算では、結果の総桁数（整数部と小数部）が 31 を超えるとエラーになります。

累乗では、以下の条件のいずれかに該当するとエラーが発生します。

- 基数がパック型で、結果が 16 桁を超えたか、または任意の中間結果が 15 桁を超えた場合
- 基数が浮動小数点で、結果が概算で次の値を超えた場合：$7 * 10^{75}$

配列の処理

通常、次の規則が適用されます。

- スカラ演算はすべて、単一のオカレンスを構成する配列要素に適用されます。
- 定数値を使用して変数を定義すると（#FIELD (I2) CONSTANT <8> など）、コンパイル時に値が変数に割り当てられ、その変数は定数として処理されます。 したがって、このような変数を配列のインデックスとして使用すると、その次元は特定の数のオカレンスを持つことになります。
- 割り当て／比較操作で次元数の異なる2つの配列を使用する場合、次元数の少ない配列の「足りない」 次元は (1:1) とみなされます。

  例：#ARRAY1 (1:2) を #ARRAY2 (1:2,1:2) に割り当てる場合、#ARRAY1 は #ARRAY1 (1:1,1:2) とみなされます。

以下では次のトピックについて説明します。

配列の次元の定義

1 次元、2 次元、および 3 次元の配列は、以下のように定義します。

次元数	プロパティ
3	#a3(第 3 次元, 第 2 次元, 第 1 次元)
2	#a2(第 2 次元, 第 1 次元)
1	#a1(第 1 次元)

配列の割り当て操作

配列の範囲を別の配列の範囲に割り当てる場合、割り当ては要素ごとに実行されます。

例：

```
DEFINE DATA LOCAL
1 #ARRAY(I4/1:5) INIT <10,20,30,40,50>
END-DEFINE
*
MOVE #ARRAY(2:4) TO #ARRAY(3:5)
/* is identical to
/* MOVE #ARRAY(2) TO #ARRAY(3)
/* MOVE #ARRAY(3) TO #ARRAY(4)
/* MOVE #ARRAY(4) TO #ARRAY(5)
/*
/* #ARRAY contains  10,20,20,20,20
```

単一のオカレンスを配列の範囲に割り当てる場合、範囲の各要素に単一のオカレンスの値が割り当てられます。 算術関数を使用する場合、範囲の各要素に関数の結果が割り当てられます。

割り当て操作を実行する前に、配列の各次元を相互に比較して、以下にリストされている条件に該当するかどうかをチェックします。 次元は個別に比較されます。つまり、配列の第 1 次元はもう 1 つの配列の第 1 次元と、配列の第 2 次元はもう 1 つの配列の第 2 次元と、そして配列の第 3 次元はもう 1 つの配列の第 3 次元と比較されます。

ある配列から別の配列への値の割り当ては、以下の条件のいずれかに該当する場合にのみ実行できます。

- 比較する次元のオカレンス数が一致している場合
- 比較する次元の両方のオカレンス数が無限の場合
- 別の次元に割り当てられる次元が単一のオカレンスで構成されている場合

例 - 配列の割り当て：

以下のプログラムは、実行可能な配列の割り当て操作を示しています。

```
DEFINE DATA LOCAL
1 A1   (N1/1:8)
1 B1   (N1/1:8)
1 A2   (N1/1:8,1:8)
1 B2   (N1/1:8,1:8)
1 A3   (N1/1:8,1:8,1:8)
1 I    (I2)          INIT <4>
1 J    (I2)          INIT <8>
1 K    (I2)          CONST <8>
END-DEFINE
*
COMPUTE A1(1:3) = B1(6:8)                                    /* allowed
COMPUTE A1(1:I) = B1(1:I)                                    /* allowed
```

```
 COMPUTE A1(*)   = B1(1:8)                                    /* allowed
 COMPUTE A1(2:3) = B1(I:I+1)                                  /* allowed
 COMPUTE A1(1)   = B1(I)                                      /* allowed
 COMPUTE A1(1:I) = B1(3)                                      /* allowed
 COMPUTE A1(I:J) = B1(I+2)                                    /* allowed
 COMPUTE A1(1:I) = B1(5:J)                                    /* allowed
 COMPUTE A1(1:I) = B1(2)                                      /* allowed
 COMPUTE A1(1:2) = B1(1:J)                                    /* NOT ALLOWED
(NAT0631)
 COMPUTE A1(*)   = B1(1:J)                                    /* NOT ALLOWED
(NAT0631)
 COMPUTE A1(*)   = B1(1:K)                                    /* allowed
 COMPUTE A1(1:J) = B1(1:K)                                    /* NOT ALLOWED
(NAT0631)
 *
 COMPUTE A1(*)       = B2(1,*)                                /* allowed
 COMPUTE A1(1:3)     = B2(1,I:I+2)                            /* allowed
 COMPUTE A1(1:3)     = B2(1:3,1)                              /* NOT ALLOWED
(NAT0631)
 *
 COMPUTE A2(1,1:3)   = B1(6:8)                                /* allowed
 COMPUTE A2(*,1:I)   = B1(5:J)                                /* allowed
 COMPUTE A2(*,1)     = B1(*)                                  /* NOT ALLOWED
(NAT0631)
 COMPUTE A2(1:I,1)   = B1(1:J)                                /* NOT ALLOWED
(NAT0631)
 COMPUTE A2(1:I,1:J) = B1(1:J)                                /* allowed
 *
 COMPUTE A2(1,I)     = B2(1,1)                                /* allowed
 COMPUTE A2(1:I,1)   = B2(1:I,2)                              /* allowed
 COMPUTE A2(1:2,1:8) = B2(I:I+1,*)                            /* allowed
 *
 COMPUTE A3(1,1,1:I)   = B1(1)                                /* allowed
 COMPUTE A3(1,1,1:J)   = B1(*)                                /* NOT ALLOWED
(NAT0631)
  COMPUTE A3(1,1,1:I)   = B1(1:I)                             /* allowed
 COMPUTE A3(1,1:2,1:I) = B2(1,1:I)                            /* allowed
 COMPUTE A3(1,1,1:I)   = B2(1:2,1:I)                          /* NOT ALLOWED
```

```
(NAT0631)
 END
```

配列の比較操作

一般的に、複数の次元を持つ配列を比較する場合、各次元は個別に処理されます。つまり、配列の第 1 次元はもう 1 つの配列の第 1 次元と、配列の第 2 次元はもう 1 つの配列の第 2 次元と、そして配列の第 3 次元はもう 1 つの配列の第 3 次元と比較されます。

2 つの配列の次元の比較は、以下の条件のいずれかに該当する場合にのみ実行できます。

- 比較する配列の次元のオカレンス数が同じ場合
- 比較する配列の次元のオカレンス数が無限の場合
- どちらかの配列の全次元が単一のオカレンスの場合

例 - 配列の比較：

以下のプログラムは、実行可能な配列の比較操作を示しています。

```
DEFINE DATA LOCAL
1 A3  (N1/1:8,1:8,1:8)
1 A2  (N1/1:8,1:8

 1 A1  (N1/1:8)
1 I   (I2)   INIT <4>
1 J   (I2)   INIT <8>
1 K   (I2)   CONST <8>
END-DEFINE
*
IF A2(1,1)     = A1(1)              THEN IGNORE END-IF /* allowed
IF A2(1,1)     = A1(I)              THEN IGNORE END-IF /* allowed
IF A2(1,*)     = A1(1)              THEN IGNORE END-IF /* allowed
IF A2(1,*)     = A1(I)              THEN IGNORE END-IF /* allowed
IF A2(1,*)     = A1(*)              THEN IGNORE END-IF /* allowed
IF A2(1,*)     = A1(I -3:I+4)       THEN IGNORE END-IF /* allowed
IF A2(1,5:J)   = A1(1:I)            THEN IGNORE END-IF /* allowed
IF A2(1,*)     = A1(1:I)            THEN IGNORE END-IF /* NOT ALLOWED(NAT0629)
IF A2(1,*)     = A1(1:K)            THEN IGNORE END-IF /* allowed
*
IF A2(1,1)     = A2(1,1)            THEN IGNORE END-IF /* allowed
IF A2(1,1)     = A2(1,I)            THEN IGNORE END-IF /* allowed
IF A2(1,*)     = A2(1,1:8)          THEN IGNORE END-IF /* allowed
IF A2(1,*)     = A2(I,I -3:I+4)     THEN IGNORE END-IF /* allowed
IF A2(1,1:I)   = A2(1,I+1:J)        THEN IGNORE END-IF /* allowed
IF A2(1,1:I)   = A2(1,I:I+1)        THEN IGNORE END-IF /* NOT ALLOWED(NAT0629)
IF A2(*,1)     = A2(1,*)            THEN IGNORE END-IF /* NOT ALLOWED(NAT0629)
IF A2(1,1:I)   = A1(2,1:K)          THEN IGNORE END-IF /* NOT ALLOWED(NAT0629)
*
```

```
IF A3(1,1,*)   = A2(1,*)             THEN IGNORE END-IF /* allowed
IF A3(1,1,*)   = A2(1,I -3:I+4)      THEN IGNORE END-IF /* allowed
IF A3(1,*,I:J) = A2(*,1:I+1)         THEN IGNORE END-IF /* allowed
IF A3(1,*,I:J) = A2(*,I:J)           THEN IGNORE END-IF /* allowed
END
```

2 つの配列の範囲を比較する場合、以下の 2 つの表現では結果が異なることに注意してください。

```
#ARRAY1(*) NOT EQUAL #ARRAY2(*)
NOT #ARRAY1(*) = #ARRAY2(*)
```

例：

- 条件 **A**：

  ```
  IF #ARRAY1(1:2) NOT EQUAL #ARRAY2(1:2)
  ```

  これは、以下と等しくなります。

  ```
  IF (#ARRAY1(1) NOT EQUAL #ARRAY2(1)) AND (#ARRAY1(2) NOT EQUAL #ARRAY2(2))
  ```

  したがって、#ARRAY1 の最初のオカレンスと #ARRAY2 の最初のオカレンスが異なり、かつ、#ARRAY1 の 2 番目のオカレンスと #ARRAY2 の 2 番目のオカレンスが異なっている場合に、条件 A は真になります。

- 条件 **B**：

  ```
  IF NOT #ARRAY1(1:2) = #ARRAY2(1:2)
  ```

  これは、以下と等しくなります。

```
IF NOT (#ARRAY1(1)= #ARRAY2(1) AND #ARRAY1(2) = #ARRAY2(2))
```

また、以下とも等しくなります。

```
IF (#ARRAY1(1) NOT EQUAL #ARRAY2(1)) OR (#ARRAY1(2) NOT EQUAL #ARRAY2(2))
```

したがって、#ARRAY1 の最初のオカレンスと #ARRAY2 の最初のオカレンスが異なっているか、*または*、#ARRAY1 の 2 番目のオカレンスと #ARRAY2 の 2 番目のオカレンスが異なっている場合に、条件 B は真になります。

配列での算術演算

配列を使用する算術演算の一般的なルールは、対応する次元のオカレンス数が一致している必要があるということです。

以下の例は、このルールに従っています。

```
#c(2:3,2:4) := #a(3:4,1:3) + #b(3:5)
```

つまり、次のようになります。

配列	次元	オカレンス数	範囲
#c	第 2 次元	2	2:3
#c	第 1 次元	3	2:4
#a	第 2 次元	2	3:4
#a	第 1 次元	3	1:3
#b	第 1 次元	3	3:5

演算は要素ごとに実行されます。

Note: 次元数の異なる算術演算を実行することもできます。

上記の例では、以下の演算が実行されます。

```
#c(2,2) := #a(3,1) + #b(3)

#c(2,3) := #a(3,2) + #b(4)

#c(2,4) := #a(3,3) + #b(5)

#c(3,2) := #a(4,1) + #b(3)

#c(3,3) := #a(4,2) + #b(4)

#c(3,4) := #a(4,3) + #b(5)
```

以下のリストは、（COMPUTE、ADD、およびMULTIPLYの各ステートメントにおける）算術演算で配列の範囲がどのように使用されるかを例示したものです。例1～4では、対応する次元のオカレンス数が一致している必要があります。

1. *range* + *range* = *range*

   加算は要素ごとに実行されます。

2. *range* * *range* = *range*

   乗算は要素ごとに実行されます。

3. *scalar* + *range* = *range*

   スカラが範囲の各要素に加算されます。

4. *range* * *scalar* = *range*

   範囲の各要素がスカラで乗算されます。

5. *range* + *scalar* = *scalar*

   範囲の各要素がスカラに加算され、その結果がスカラに割り当てられます。

6. *scalar* * *range* = *scalar2*

   スカラが配列の各要素で乗算され、その結果が *scalar2* に割り当てられます。

上記の例に示されているように、算術演算では中間結果が生成されるため、重複するインデックス範囲は要素ごとに計算され、その結果が中間結果配列に格納されます。そして最後に、中間結果配列が結果フィールドに割り当てられます。

例：

```
DEFINE DATA LOCAL
1 #ARRAY(I4/1:5) INIT <10,20,30,40,50>
END-DEFINE

#ARRAY(3:5) := #ARRAY(2:4) + 1

/* A temporary array for the
/* intermediate result values is
/* generated implicitly: #temp(1:3).
/* The following operations are
/* performed internally:
/* #temp(1) := #ARRAY(2) + 1
/* #temp(2) := #ARRAY(3) + 1
/* #temp(3) := #ARRAY(4) + 1
/* #ARRAY(3:5) := #temp(1:3)
/*
/* #ARRAY contains  10,20,21,31,41
```

56 3GL プログラムからの Natural サブプログラムの呼び出し

Natural サブプログラムは、第 3 世代プログラミング言語（3GL）で作成されたプログラミングオブジェクトから呼び出すことができます。呼び出し元のプログラムは、標準の CALL インターフェイスをサポートする任意のプログラミング言語で作成できます。

この目的のために、Natural にはインターフェイス ncxr_callnat が用意されています。 3GL プログラムでは、必要なサブプログラム名を指定してこのインターフェイスを呼び出します。

Note: あらかじめ Natural を有効化しておく必要があります。つまり、Natural オブジェクトで CALL ステートメントを使用して、呼び出し元の 3GL プログラムを呼び出しておく必要があります。

サブプログラムは、CALLNAT ステートメントで別の Natural オブジェクトから呼び出されたかのように実行されます。

（END ステートメントまたは ESCAPE ROUTINE ステートメントで）サブプログラムの処理が終了すると、3GL プログラムに制御が戻ります。

このchapterでは、次のトピックについて説明します。

3GL プログラムからサブプログラムへのパラメータ渡し

呼び出し元の 3GL プログラムから Natural サブプログラムにパラメータを渡すことができます。パラメータを渡す場合、CALL ステートメントを使用してパラメータを渡すときと同じルールが適用されます。

3GL プログラムでは、以下の 4 つのパラメータとともに Natural インターフェイス ncxr_callnat を呼び出します。

- 1 番目のパラメータは、呼び出す Natural サブプログラムの名前です。
- 2 番目のパラメータは、サブプログラムに渡すパラメータの数です。
- 3 番目のパラメータは、サブプログラムに渡すパラメータのアドレスが格納されたテーブルのアドレスです。
- 4 番目のパラメータは、サブプログラムに渡すパラメータのフォーマット／長さ指定が格納されたテーブルのアドレスです。

呼び出し元プログラムのパラメータの順序、フォーマット、および長さは、サブプログラムの DEFINE DATA PARAMETER ステートメント内のフィールドの順序、フォーマット、および長さと正確に一致している必要があります。 呼び出し元プログラムのフィールド名と呼び出されるサブプログラムのフィールド名は、同一である必要はありません。

3GL プログラムからの Natural サブプログラムの呼び出し例

3GL プログラムから Natural サブプログラムを呼び出す方法の例については、Natural ルートディレクトリのサブディレクトリ *samples\sysexuex* にある以下の例を参照してください。

- *MY3GL.NSP*（メインプログラム）
- *MY3GLSUB.NSN*（サブプログラム）
- *MYC3GL.C*（"C" の関数）

57 Natural プログラム内からのオペレーティングシステムコマンドの発行

Natural ユーザー出口 SHCMD は、Natural プログラム内からオペレーティングシステムコマンドの発行を行うために使用できます。

このchapterでは、次のトピックについて説明します。

構文

```
CALL 'SHCMD' 'command' ['option']
```

パラメータ

command	*command* は、オペレーティングシステムによって実行されます。 コマンド（ディレクトリ表示の DIR や削除の DEL など）を実行するには、システムコマンドインタープリタも指定する必要があります。 詳細については、以下の「*例*」を参照してください。
option	*option* には、コマンドを実行する方法を指定します。 このパラメータの指定は任意です。次のオプションを使用できます。 ■ ASYNCH ■ NOSCREENIO ■ SYNCH（デフォルト） 以下の「パラメータオプション」を参照してください。

パラメータオプション

以下のコマンドオプションを使用できます。

オプション	説明
ASYNCH	Natural は、コマンドの実行が完全に終了するまで待ちません。 このような種類の処理を非同期処理といいます。
NOSCREENIO	このオプションは、コマンドによって生成される出力を抑制するために使用します。 抑制された出力は、NULL デバイスにリダイレクトされます。
SYNCH	Natural は、コマンドの実行が完全に終了するまで待ちます。 このような種類の処理を同期処理といいます。デフォルトはこの設定です。

Note: オプション ASYNCH と SYNCH は、同時に設定できません。

リターンコード

以下のリターンコードの値を使用できます。

リターンコード	説明
0	コマンドが正常に実行されました。
4	不正な SHCMD パラメータが指定されました。
その他のすべてのコード	オペレーティングシステム依存のエラーコード

例

オペレーティングシステムコマンド DIR を実行して、ディレクトリを表示します。

```
CALL 'SHCMD''CMD.EXE /C DIR'
```

Natural 関数 RET を使用してリターンコードを取得します。

```
RESET rc (I4)
CALL 'SHCMD''CMD.EXE /C DIR'
rc = RET( 'SHCMD' )                    /* retrieve return code
IF rc <> 0 THEN                        /* in case of an error
DISPLAY "Error occurred during SHCMD" /* display an error message
```

引用符でコマンドを囲むことにより、空白を含むコマンドを実行できます。

以下の例は Microsoft Excel を実行します。

```
RESET #cmd (A253)
MOVE '"C:\Program Files\Microsoft Office\Office\EXCEL.EXE"' to #cmd
CALL "SHCMD" #cmd "ASYNCH"
```

この場合、パラメータ TQ（引用符の変換）を OFF にする必要があります。そうしないと、引用符が削除されます。

TQ パラメータに依存しないようにするには、引用符の 16 進数の ASCII コード（H'22'）をコマンドの前後に付加します。 以下に例を示します。

```
RESET #cmd (A253)
MOVE H'22' - "C:\Program Files\Microsoft Office\Office\EXCEL.EXE" - H'22' to #cmd
CALL "SHCMD" #cmd "ASYNCH"
```

58 インターネットおよび XML アクセス用のステートメント

このchapterでは、インターネットおよび XML にアクセスするためのステートメントの概要、およびその他の参照情報について説明します。 これらのステートメントを最大限に活用するには、通信規格に関する深い知識が必要です。

このchapterでは、次のトピックについて説明します。

使用可能なステートメント

インターネットおよび XML へのアクセスには、以下のステートメントを使用できます。

- REQUEST DOCUMENT
- PARSE XML

REQUEST DOCUMENT

このステートメントを使用すると、HTTP プロトコルを使用できます。

以下の例は、このステートメントを使用して、外部に保管されているドキュメントにアクセスする方法を示しています。

```
REQUEST DOCUMENT FROM
"http://bolsap1:5555/invoke/sap.demo/handle_RFC_XML_POST"
WITH
USER #User PASSWORD #Password
DATA
NAME 'XMLData' VALUE #Queryxml
NAME 'repServerName' VALUE 'NT2'
RETURN
PAGE #Resultxml
RESPONSE #rc
```

詳細については、『ステートメント』ドキュメントの「REQUEST DOCUMENT」を参照してください。

PARSE XML

PARSE XML ステートメントを使用すると、Natural プログラムから XML ドキュメントを解析できるようになります。

詳細については、『ステートメント』ドキュメントの「PARSE XML」を参照してください。

その他の参照情報

役に立つリソースのリストを以下に示します。

サンプルプログラム

各ステートメントの説明の最後に用意されているサンプルプログラムの他に、Naturalライブラリ SYSEXV にサンプルプログラムが含まれています。

技術論文

インターネットおよび XML にアクセスするための Natural ステートメントの使用に関する技術論文は、ナレッジセンターエリアの ServLine24 で確認できます。

訓練コース

Software AG のコーポレートユニバーシティでは、このテーマの特別訓練コースを用意しています。 コーポレートユニバーシティの訓練コースについては、
http://servline24.softwareag.com/public/ の ServLine24 を参照してください。

また、お近くの地域におけるオンサイトの特別訓練コースについて、担当地域の Software AG にお問い合わせください。

有効なリンク

役に立つと思われるリンクを以下に示します。

- World Wide Web Consortium（W3C）：*http://www.w3.org/*
- Extensible Markup Language（XML）：*http://www.w3.org/XML/*
- HyperText Markup Language（HTML）ホームページ：*http://www.w3.org/MarkUp/*
- W3 Schools：*http://www.w3schools.com/*

59 ポータブル Natural 生成プログラム

Natural バージョン 5 以降では、Natural 生成プログラム（GP）を UNIX、OpenVMS、および Windows のプラットフォーム間で移植可能です。

このchapterでは、次のトピックについて説明します。

互換性

Natural バージョン 5 以降では、Natural でサポートされる UNIX、OpenVMS、および Windows の任意のプラットフォームでカタログされたソースは、これらすべてのオープンシステムプラットフォームで再コンパイルなしで実行できます。 この機能によって、オープンシステムプラットフォーム間でのアプリケーションの展開が容易になります。

Natural バージョン 4 または Natural バージョン 3 で生成された Natural アプリケーションは、再カタログしなくても Natural バージョン 5 以降で実行できます（上位互換性）。 この場合には、ポータブル GP 機能は有効ではありません。 ポータブル GP およびその他の改良点を利用するには、Natural バージョン 5 以降でカタログする必要があります。

コマンドプロセッサ GP は移植できません。 ポータブル GP 機能は、メインフレームプラットフォームでは使用できません。 このことは、メインフレームコンピュータで生成された Natural GP は、アプリケーションを再コンパイルしない場合、UNIX、OpenVMS、および Windows プラットフォームで実行できないことを意味します。逆の場合も同様です。

エンディアンモードについて

Natural バージョン 5 以降では、UNIX、OpenVMS、または Windows プラットフォームのどれが実行中であるかに応じて、マルチバイトの数値がGPに保存されるときのバイト順序が考慮されます。 2 つのバイト順モードがあり、「リトルエンディアン」 と 「ビッグエンディアン」 と呼ばれています。

- 「リトルエンディアン」 とは、数値の下位バイトがメモリの最下位アドレスに格納され、上位バイトが最上位アドレスに格納されることを意味します（小さな桁が最初に来ます）。
- 「ビッグエンディアン」 とは、数値の上位バイトがメモリの最下位アドレスに格納され、下位バイトが最上位アドレスに格納されることを意味します（大きな桁が最初に来ます）。

UNIX、OpenVMS、および Windows プラットフォームでは、両方のエンディアンモードが使用されます。Intel プロセッサおよび AXP コンピュータのバイト順は 「リトルエンディアン」 であり、HP-UX、Sun Solaris、RS6000 などその他のプロセッサでは "ビッグエンディアン" モードが使用されています。

Natural では、必要に応じて、ポータブル GP を実行プラットフォームのエンディアンモードに自動的に変換します。 GP がすでにプラットフォームのエンディアンモードで 生成されている場合、このエンディアン変換は実行されません。

ENDIAN パラメータ

ポータブル GP の実行パフォーマンスを拡張するために、プロファイルパラメータ ENDIAN が導入されました。 ENDIAN は、コンパイル中に GP が生成されるエンディアンモードを決定します。

DEFAULT	GP が生成されるマシンのエンディアンモード
BIG	ビッグエンディアンモード（上位バイトが最初に来ます）
LITTLE	リトルエンディアンモード（下位バイトが最初に来ます）

DEFAULT、BIG、および LITTLE の値のうち 1 つを選択します。デフォルト値は DEFAULT です。

ENDIAN モードパラメータは次の方法で設定できます。

- Natural コンフィグレーションユーティリティを使用して、プロファイルパラメータとして設定
- 開始パラメータとして設定
- セッションパラメータとして、または GLOBALS コマンドを使用して設定

Natural 生成プログラムの転送

異なるプラットフォーム（UNIX、OpenVMS、および Windows）でポータブル GP を利用するには、生成された Natural オブジェクトをターゲットプラットフォームに転送するか、または NFS などを経由してターゲットプラットフォームからアクセス可能にする必要があります。

Natural 生成オブジェクト、または Natural アプリケーション全体を配布するには、Natural オブジェクトハンドラの使用をお勧めします。 手順としては、ソース環境のオブジェクトをワークファイルにアンロードし、このワークファイルをターゲット環境に転送して、ワークファイルからオブジェクトをロードします。

手順 59.1. オープンシステムプラットフォーム間で Natural 生成オブジェクトを展開するには

1 Natural オブジェクトハンドラを開始します。 必要なすべてのカタログオブジェクトを PORTABLE タイプのワークファイルにアンロードします。

必要であれば、エラーメッセージもワークファイルにアンロードできます。

> ⚠ **Important:** 指定するワークファイルタイプは PORTABLE でなければなりません。 PORTABLE は、ワークファイルが異なるマシンに転送されるときに、ワークファイルの自動エンディアン変換を実行します。『ステートメント』ドキュメントの「DEFINE

WORK FILE」ステートメントの説明にあるワークファイルタイプの情報も参照してください。

2 ワークファイルをターゲット環境に転送します。 ネットワーク、CD、フロッピーディスク、テープ、電子メール、ダウンロードなど使用する転送メカニズムによっては、ZIP ファイルなどの圧縮アーカイブの使用や、UUENCODE/UUDECODE などによるエンコードが有効な場合もあります。 FTP 経由のコピーでは、バイナリ転送タイプが必要です。

Note: 使用する転送メソッドによっては、ロード機能を続行する前に、転送するワークファイルのレコードフォーマットと属性またはブロックサイズを、特定のターゲットプラットフォームに応じて調整する必要があります。 ターゲットプラットフォームでのワークファイルのフォーマットと属性は、そこで生成された同じタイプのワークファイルと同じものにする必要があります。 適応が必要な場合は、オペレーティングシステムツールを使用してください。

3 ターゲット環境で Natural オブジェクトハンドラを開始します。 ワークファイルタイプとして PORTABLE を選択します。 Natural オブジェクトおよびエラーメッセージをワークファイルからロードします。

Naturalオブジェクトハンドラの使用法の詳細については、『*ユーティリティ*』ドキュメントの「*オブジェクトハンドラ*」を参照してください。

Natural 開発ワークステーションから Natural ランタイムワークステーションへアプリケーションを移植する方法の詳細については、『*オペレーション*』ドキュメントの「*移行手順の概要*」を参照してください。

上述の推奨される方法に加えて、オペレーティングシステムツールや別の転送方法を使用することにより、単一のNatural生成オブジェクトだけでなく、ライブラリ全体やその一部を「移動」またはコピーするためのさまざまな方法があります。 これらすべての場合について、オブジェクトをNaturalで実行できるようにするには、オブジェクトをNaturalシステムファイルFUSERにインポートして *FILEDIR.SAG* 構造に適合させる必要があります。 FNAT または FUSER ディレクトリの詳細については、『*オペレーション*』ドキュメントの「*システムファイル FNAT と FUSER*」を参照してください。

この作業には、次のいずれかの方法を使用します。

- SYSMAIN ユーティリティのインポート機能を使用する。
- FTOUCH ユーティリティを使用する。 このユーティリティは、Natural に入らずに使用できます。
- ドラッグ & ドロップまたはメニューコマンドの［切り取り］、［コピー］、および［貼り付け］を使用して、Windows エクスプローラから Natural 環境にファイルをインポートすることもできます。 このことは、Windows エクスプローラ経由でインポートする Natural オブジェクトにアクセスできる場合は、ドラッグ & ドロップや［切り取り］、［コピー］、および［貼り付け］が使用可能であり、*FILEDIR.SAG* ファイルは自動的に更新されることを意味

します。オブジェクトのコピー、移動、およびインポートの詳細については、『*Natural スタジオの使用*』ドキュメントの「*Natural オブジェクトの管理*」を参照してください。

同じことが、NSFやネットワークファイルサーバーなどを経由して、ターゲットプラットフォームからソース環境で生成されたオブジェクトへの直接アクセスが可能な場合にも当てはまります。この場合は、オブジェクトをインポートする必要があります。

移植可能な FILEDIR.SAG ファイルとエラーメッセージファイル

Natural バージョン 6.2 以降では、*FILEDIR.SAG* ファイルおよびエラーメッセージファイルはプラットフォームに依存しません。 したがって、共通の FUSER システムファイルをさまざまなオープンシステムプラットフォームで共有できます。例えば、1つのオープンシステムプラットフォームから一連のNaturalライブラリをコピーし、オペレーティングシステムのコピー手順を使用して別のプラットフォームにコピーすることができます。 ただし、FNAT システムファイルの共有はお勧めしません。 移植可能な *FILEDIR.SAG* に関する詳細については、『*オペレーション*』ドキュメントの「*Natural の移植可能なシステムファイル*」を参照してください。

60 イベントドリブンプログラミングについて

このセクションでは、次のトピックについて説明します。

- イベントドリブンアプリケーションとは
- **GUI** 開発環境
- **GUI** 設計のヒント
- アプリケーション作成に関連するタスク
- チュートリアル
- 基本的な用語

イベントドリブンプログラミングの詳細については、「*イベントドリブンプログラミングの手法*」を参照してください。

61 イベントドリブンアプリケーションとは

このchapterでは、次のトピックについて説明します。

はじめに

イベントドリブンアプリケーションは、プログラムドリブンアプローチ以外の新しい開発アプローチです。Naturalは、この両方を提供しています。イベントドリブンプログラミングでは、グラフィカルユーザーインターフェイスを通した入力に従って、アプリケーションが実行されます。

プログラムドリブンアプリケーションでは、イベントではなくアプリケーションが、実行するコードを制御します。 実行可能なコードの最初の行から実行が開始され、定義されたパスウェイに従ってアプリケーションを通過し、あらかじめ設定された順序で指示どおりに追加のプログラムを呼び出します。

イベントドリブンプログラミングでは、ユーザーアクションまたはシステムイベントに関連付けられているコードが起動されます。 したがって、コードの実行順序は、発生するイベント、つまりユーザーの行動に依存します。 これが、グラフィカルユーザーインターフェイスおよびイベントドリブンプログラミングの基本です。主体はユーザーで、コードはユーザーの行動に応答します。 イベントドリブンプログラムはキャラクタ指向のインターフェイスでも実行できますが、グラフィカルユーザーインターフェイスで実行する方がより一般的です。

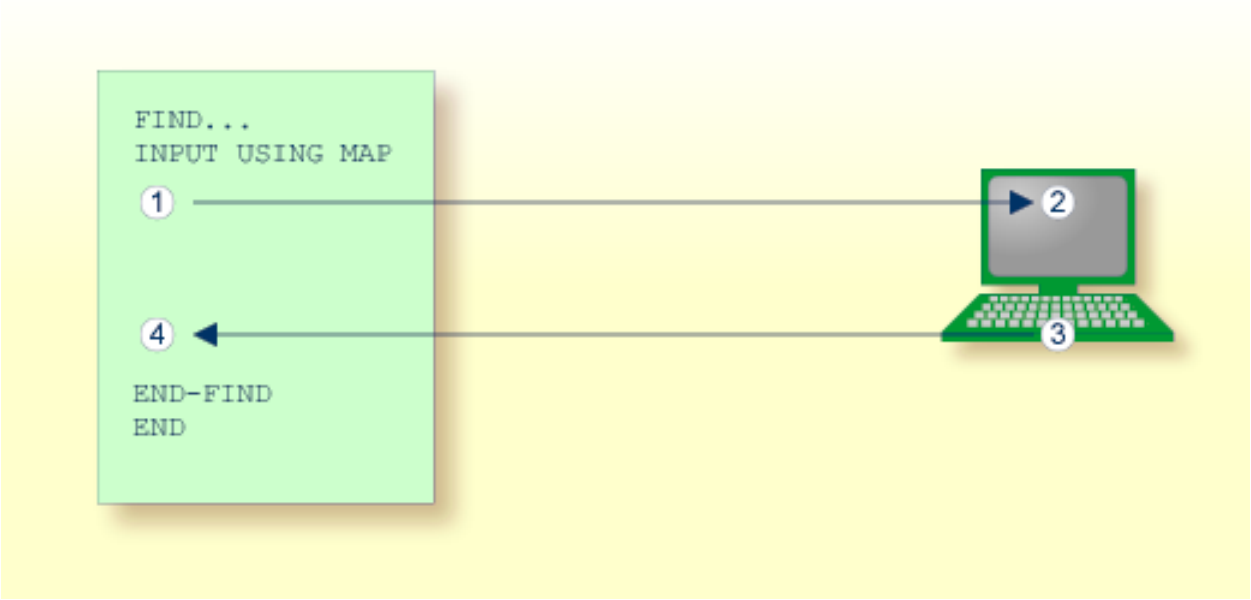
ユーザーが何をするかは予測できないため、コードは実行時に状況を想定する必要があります。例えば、アプリケーションでは、［**OK**］ボタンを押す前にユーザーは編集エリアコントロールにテキストを追加している、などと想定します。

想定をする場合、これらの想定が常に有効となるようにアプリケーションを構築する必要があります。 例えば、ユーザーにテキストを確実に追加させるために、あらかじめボタンを無効にしておき、編集エリアコントロールのCHANGEイベントが発生した場合にのみボタンを有効にします。

コードでは、特定の操作を実行しながら別のイベントを起動できます。例えば、スクロールバーコントロールのスライダを動かすと、CHANGE イベントが起動されます。

以下の図は、プログラムドリブンアプリケーションとイベントドリブンアプリケーションの違いを示しています。

プログラムドリブンアプリケーション

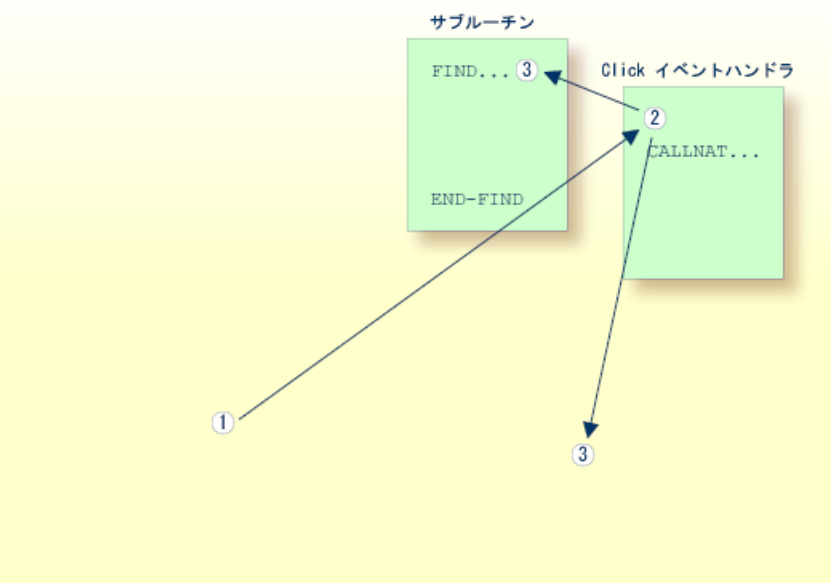


典型的なプログラムドリブンアプリケーションでは、以下の一連の手順が適用されます。

1. プログラムが端末に画面を送信します。
2. ユーザーは画面のデータフィールドに入力します。
3. ユーザーが Enter キーまたはファンクションキーを押します。
4. プログラムはユーザーの入力が有効かどうかを判断します。

データが有効な場合、`END` ステートメントに達するまで処理が続けられます。

イベントドリブンアプリケーション



典型的なイベントドリブンアプリケーションでは、以下の一連の手順が適用されます。

1. ユーザーが画面上でアクションを要求します。
2. イベントハンドラコードはコンテキストに従ってバックグラウンドで対応します。
3. 一定の条件が満たされると、実行されたイベントハンドラコードは他の Natural コード（ここではサブルーチン）を起動するか、または画面に制御を戻します。

プログラムドリブンアプローチでは、ユーザーは Enter キーとファンクションキーを通してコードと対話します。一方、イベントドリブンアプリケーションでは、ユーザーが特定のコード（イベントハンドラ）を起動します。 一般に、イベントドリブンアプリケーションでは、ユーザーの入力を待っている間はどのようなコードも実行されていません。同じ状況でもプログラムドリブンアプリケーションでは、INPUT ステートメントが処理されている可能性があります。

ここで起きていること

グラフィカルユーザーインターフェイスプログラムでは、ユーザーによって開始される個々のイベントに対応するプログラムを作成する必要があります。

イベントとは、ダイアログまたはダイアログエレメントによって認識されるアクションのことです。 イベントドリブンアプリケーションは、イベントに応答してコードが実行されます。 各ダイアログまたは各ダイアログエレメントには、一連のイベントがあらかじめ定義されています。

これらのイベントの 1 つが発生すると、Natural は、関連付けられているイベントハンドラのコードを呼び出します。

プログラマは、アプリケーションのダイアログおよびダイアログエレメントが特定のイベントにどのように応答するかを指定します。 プログラムをイベントに応答させる場合は、そのイベントに対するイベントコードを作成します。

イベントドリブンコードの作成

作成する各ダイアログまたは各ダイアログエレメントには、プログラム（イベントハンドラ）が応答できる一連のイベントが Natural によってあらかじめ定義されています。 イベントに応答することは簡単です。ダイアログとダイアログエレメントには、ユーザーアクションを認識し、それらに関連付けられているコードを実行する組み込み機能が備わっています。

すべてのイベントのコードを作成する必要はありません。 ダイアログオブジェクトをイベントに応答させる場合は、そのイベントに応答して Natural が実行するイベントコードを作成します。

典型的なイベントドリブンアプリケーションでは、以下の一連のアクションが実行されます。

- ダイアログまたはダイアログエレメントがアクションをイベントとして認識します。アクションはユーザーが起こすことができます（クリックやキーストロークなど）。
- イベントに対応するイベントコードがある場合は、そのコードが実行されます。
- アプリケーションは次のイベントを待ちます。

イベントに応答するために作成するイベントコードでは、計算の実行、入力データの取得、およびインターフェイスの一部の操作を実行できます。Naturalを使用して属性の設定値を変更することにより、ダイアログまたはダイアログエレメントを操作します。

Caution: 繰り返し発生するイベントによって生じるカスケードイベントを、コードで作成しないようにしてください。 例えば、ユーザーがスクロールバーコントロールのスライダをドラッグすると、現在の `SLIDER` 属性の設定が自動的に変更され、CHANGE イベントが起動されます。 CHANGE イベントに関連付けられているコードでも現在の `SLIDER` 属性の設定が変更されていると、CHANGE イベントが再び起動され、現在の `SLIDER` 属性設定が再度調整され、CHANGEイベントがもう一度起動される、というふうに繰り返されます。 この速度では、あっという間にメモリが不足します。

イベントドリブンアプリケーションのコンポーネント

以下では次のトピックについて説明します。

- ダイアログ
- ダイアログエレメント
- 属性
- イベントハンドラ
- データエリア - グローバル、ローカル、パラメータ
- インラインサブルーチン

ダイアログ

ダイアログは、イベントドリブンアプリケーションの中心的な Natural オブジェクトです。 イベントドリブンアプリケーションは、基本ダイアログを実行することによって開始されます。 `OPEN DIALOG` ステートメントを指定すると、基本ダイアログから他の従属ダイアログを開くことができます。 プログラムドリブンアプリケーションとは異なり、これらのダイアログは通常はモードレス、つまり、エンドユーザーはすべての開いているダイアログを同時に処理することができます。 アプリケーションは、基本ダイアログが閉じられると終了します。

ダイアログは、ダイアログエディタで作成します。マップエディタと同様に、ダイアログエディタは、ダイアログウィンドウとそのダイアログエレメント、グローバルデータエリア（GDA）、ローカルデータエリア（LDA）、パラメータデータエリア（PDA）、サブルーチン、および特定のイベントハンドラセクションの指定に基づいて、Natural オブジェクトを構築します。

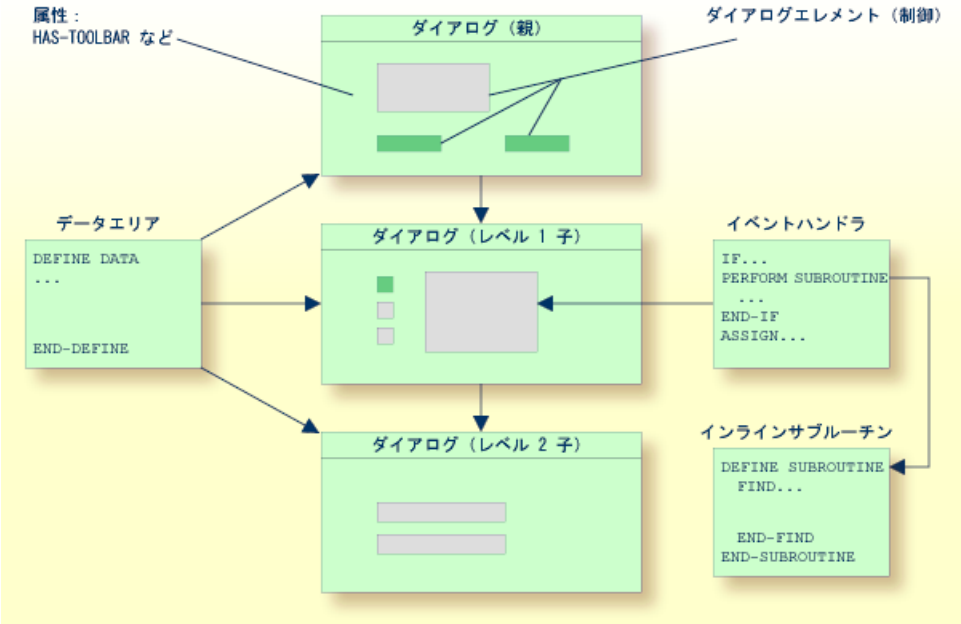
ダイアログの実行時における、システム変数 `*DIALOG-ID` によって識別されるランタイムインスタンスと、ダイアログウィンドウの GUI インスタンス（ハンドル。デフォルトのハンドル名は `#DLG$WINDOW`）との間には違いがあります。

アプリケーションで複数のダイアログを使用する場合は、基本ダイアログウィンドウと他のダイアログをどのように関連付けるのかを指定する必要があります。まず、アプリケーションを MDI（マルチドキュメントインターフェイス）にする必要があるかどうかを決める必要があります。

MDI アプリケーションを選択した場合、基本ダイアログは 「MDI フレームウィンドウ」 タイプに、従属ダイアログは 「MDI 子ウィンドウ」 タイプおよび 「標準ウィンドウ」 タイプにする必要があります。

非 MDI を選択した場合、アプリケーションは 「標準ウィンドウ」 タイプのダイアログのみを使用できます。

「標準ウィンドウ」 タイプのダイアログには、"ポップアップ"、"モーダル"、または "ダイアログボックス" の各スタイルがあります。



ダイアログエレメント

ほとんどすべてのダイアログエレメントは、エンドユーザーとイベントドリブンアプリケーションとの対話を可能にする、ダイアログ内のグラフィック要素です。 ダイアログをダイアログエディタで開いて属性を設定すると（下記参照）、プログラマはウィンドウ内のダイアログエレメントを「作成」できます。通常、これは、メニューコントロール、ツールバー、およびその他の要素（プッシュボタンコントロールや入力フィールドコントロールなど）から構成されます。

ダイアログエレメントの「作成」とは、ダイアログエディタのメニューまたはツールバーからダイアログエレメントのタイプを選択し、マウスを使用して必要な位置に配置することを意味します。 また、グリッドを定義することもできます。グリッドを定義すると、ダイアログエレメントをグリッドに対して位置合わせできるので、配置操作が簡単になります。

属性

属性はダイアログおよびダイアログエレメントのプロパティです。 ダイアログまたはダイアログエレメントの作成後にマウスでダブルクリックすると、対応する属性ウィンドウが表示されます。 その後、属性に値を設定できます。設定しない場合は、システムデフォルト値のままになります。 属性ウィンドウには、イベントハンドラウィンドウを開くプッシュボタンコントロールも含まれています。

イベントハンドラ

イベントハンドラとは、イベントが発生すると起動されるNaturalコードのことです。例えば、エンドユーザーがプッシュボタンコントロールをクリックすると、CLICKイベントが発生します。イベントハンドラウィンドウでは、まず（属性を設定したばかりの）ダイアログまたはダイアログエレメントで有効なイベントのリストからイベントのタイプを選択する必要があります。その後、コードウィンドウが有効になり、Naturalコードを入力できます。

データエリア - グローバル、ローカル、パラメータ

- グローバルデータエリア（GDA）は、アプリケーション内のNaturalオブジェクト間でデータフィールドを共有するために使用します。アプリケーションごとに1つのGDAを指定できます。
- ローカルデータエリア（LDA）には、ダイアログ専用のデータフィールドを格納します。
- パラメータデータエリア（PDA）は、常にダイアログに存在します。PDAは、`OPEN DIALOG`ステートメントまたは`SEND EVENT`ステートメントでパラメータをダイアログに渡すために使用します。これらのステートメントでは、`WITH`節でパラメータの名前を指定するか、またはパラメータを連続してリストすることによって、パラメータを渡します。ダイアログのパラメータデータエリアウィンドウを使用すると、自由形式でパラメータデータを入力することができます。

インラインサブルーチン

インラインサブルーチンは、多くのイベントハンドラによって呼び出される使用頻繁の高いタスクに使う標準コードを定義します。ダイアログウィンドウの［インラインサブルーチン］プッシュボタンコントロール経由で、インラインサブルーチンウィンドウにアクセスします。

62 GUI 開発環境

Natural の機能を理解するには、最初に、それを実行する環境を理解する必要があります。

グラフィカルユーザーインターフェイス（GUI）環境は、少なくとも以下の2つの重要な点で、従来のメインフレーム環境と異なっています。

- アプリケーションは画面スペースを共有します。Naturalアプリケーションは1つ以上のウィンドウのグループで実行されます。画面全体を占有することはほとんどありません。
- アプリケーションはコンピュータの時間を共有します。 アプリケーションを継続的に実行することはできません。継続して実行する場合は、バックグラウンドで実行する必要があります。

Naturalを使用すると、アプリケーションはコンピュータの時間および他のリソース（クリップボードなど）を共有します。 イベントドリブンアプリケーションは、特定のイベントの発生を待つダイアログおよびダイアログエレメントから構成されます。

イベントの実行を待っている間、（ユーザーがアプリケーションを閉じない限り）アプリケーションはデスクトップに残ります。 その間、ユーザーは、他のアプリケーションの実行、ウィンドウのサイズ変更、またはシステム設定（色など）のカスタマイズなどを実行できます。 しかし、コードは常に存在し、ユーザーがアプリケーションに戻るとアクティブになれるよう待機しています。

63 GUI 設計のヒント

このchapterでは、次のトピックについて説明します。

はじめに

GUI アプリケーションの画面の設計には、メインフレームの 3270 画面の設計とは別の知識が必要です。 それはなぜでしょうか？

それは、GUI アプリケーションはユーザーが制御するからです。これらのアプリケーションは非モーダルで、構造化されていません。 ウィンドウおよびウィンドウ内のフィールドにアクセスする順番を決めるのは、ユーザーです。 典型的なデータベースアプリケーションは、特定の順番でオペレーションを実行するようユーザーに要求します。これらのアプリケーションはフォーム指向で、構造化されています。

GUI インターフェイスは従来のメインフレームインターフェイスとは異なる機能を持っているため、GUI画面の設計も異なります。プッシュボタンコントロールやリストボックスコントロールなどのダイアログエレメントが組み込まれたウィンドウを設計できます。 イベントドリブン Natural で呼び出されるダイアログである GUI ウィンドウを設計するときは、テキストのフォントタイプとサイズ、背景色と前景色、および各ウィンドウのサイズを定義します。

以下のセクションでは、効果的な GUI 設計のためのヒントについて説明します。

調査

- プロトタイプを作成する前に、ユーザーとともに数時間過ごしてください。

  ユーザーのニーズ、好み、不満に対処するためにユーザーと話し合うことは、設計を開始するためのよい基盤となります。

- 既存の GUI 設計からアイデアを受け継ぎます。

  GUI を再開発しないことで時間を節約します。 何が動作して何が動作しないかを念頭において、他の GUI を試してください。 GUI の一貫性は、ユーザーの新規アプリケーション使用の学習を支援し、効率を高めて、トレーニングコストを削減します。 既存の GUI アプリケーションに対するユーザーのフィードバックを取得します。不十分な GUI 設計を複製したプロトタイプを開発するよりも、ユーザーの好みや不満に耳を傾けます。

- オンラインでのアプリケーション開発に時間を費やす前に、紙の上でアイデアを練ってください。

  オンラインで複数のプロトタイプを作成して時間を費やすより、紙の上でメインウィンドウの多くの画面設計オプションに目を通す方がより迅速に作業できます。 これはコーディングより簡単で、しかも、不十分な設計にいつまでも煩わされずに済みます。

ユーザーが開発プロセスにかかわっている場合、アプリケーションをシステムにインストールする前に、ユーザーのニーズと好みについて迅速にコメントをもらうことができます。 オンラインの開発ツールを使用する前に、紙のプロトタイプを使用してみてください。

画面設計

- 関連する主題に合わせて、複数のウィンドウを設計します。

  できるだけ多くのフィールドを1つの画面に入れようとする3270モニタに対する設計と違い、GUI 画面では、サブウィンドウを使用することでより優れた設計ができます。 例えば、メインウィンドウには必須フィールドのみを配置し、任意または補足の情報はすべて1つ以上のサブウィンドウに格納します。 サブウィンドウには、メインウィンドウに入力する情報がわからない場合に参照する選択肢（ドロップダウンリストなど）を組み込むことができます。 メッセージおよびフィールド依存情報は、メインウィンドウよりも補足的なウィンドウに表示した方がより効果的です。

- 見やすい、整ったウィンドウを設計します。

  3色以上の色、複数のグラフィック、および不統一な形状の使用は避け、ウィンドウが乱雑にならないようにしてください。 乱雑な画面を見てユーザーが困惑したり、気を散らしたりすることのないよう、余白を多くして画面のオブジェクトのバランスを取ります。 異なる形状やオブジェクトの使用、および色数は最小限にすることをお勧めします。

- 使いやすい、すっきりしたウィンドウを設計します。

  多数のフォント、フォントサイズ、フォントタイプまたはファミリ、および色を使用したアプリケーションは、ユーザーを困惑させる、使いにくいアプリケーションになります。1ウィンドウ当たりのフォント、フォントサイズ、フォントタイプの使用は、最大3つまでに抑えてください。 画面上で分割されることがよくあるため、斜体およびセリフフォントは使用しないようにします。 色は控えめに使用します。 目に最も優しい色は中間色です。 原色の赤や緑は非常に目立ちますが、ユーザーはこのウィンドウの前で長時間作業するのだということを忘れないでください。

- キーボードとマウスの使用について設計します。

  キーボードの使用を好み、ショートカットコマンドを記憶するユーザーもいれば、マウスを使用する方をより快適に感じるユーザーもいます。 個々のアクションは、マウスとキーボードの両方でアクセスできるようにする必要があります。

- ユーザーのニーズに従ってウィンドウを設計します。

  たくさんの機能を持つ見た目にもすばらしい画面を作成することには心を引かれますが、ユーザーが使用しなければ何の価値もありません。 設計者は、自分が利用できるすべての機能を実験するためにではなく、ユーザーが仕事をするためにアプリケーションを開発しているのだということを忘れないでください。 最初に、ユーザーに何が必要であるかを把握します。そして、彼らのニーズに合わせて設計を調整します。 さまざまな目的を持つ画面をさまざまな

方法で設計します。ユーザーに入力を促す必要があれば、会話型スタイルを使用します。ユーザーにフォームから値を入力させる場合は、データ入力スタイルを使用します。

会話型画面

- フィールドプロンプトを備えた会話型画面を設計します。

  会話ベースのスタイルでは、ユーザーは会話型でデータを入力します（例：旅行の予約）。画面のプロンプトに従ってユーザーが入力する会話ベースのスタイルでは、ラベル、ヒント、指示、さらにはユーザーがクライアントに尋ねるべき質問までをも使用して、画面を充実させます。

データ入力画面

- 簡潔なラベルを備えたデータ入力画面を設計します。

  フォームベースのスタイルでは、ユーザーはフォームからデータを入力します。 入力画面の各行はフォームの行と一致している必要があります。また、これらの行は同じ順番にする必要があります。 行対行の一致を維持するためにラベルを簡略化できます。 ヘッダーおよび指示は最小限に抑えることをお勧めします。 ラベルの唯一の目的は、作業中断後にユーザーが元の位置を見つけられるようにすることです。

メニュー設計

以下の 3 つの基準に基づいてメニューの設計をすることをお勧めします。

- ユーザーがアプリケーションを実行するオペレーティングシステムによって定義されている規則を使用してメニューを編成します。例えば、Microsoft Windows の場合は、特定のメニュー（［ファイル］、［編集］、［表示］など）、メニューオプション（［編集］メニューの［切り取り］、［コピー］、［貼り付け］など）、およびメニューバー上のメニューの特定の順序（［ヘルプ］は常に一番右に表示する、など）を使用することをお勧めします。
- 監視または有用性テストによって判断した使用頻度に基づいて、メニューを配置します。

  ユーザーがより熟練することにより使用法が変わるかどうかを予測します。 使用法の変化がオペレーティングシステムの規則に違反しないよう注意してください。
- メニュー項目をアルファベット順にリストします。 オペレーティングシステムの規則とユーザーの使用頻度を考慮してください。

色の使用法

- 色の使用は控えめにします。

  人間は一度に 5 色（プラス／マイナス 2 色）の意味しか記憶できません。

- 主要な警告としてではなく、付加的な警告として色を使用します。

  ユーザーに警告するには、明るい赤のテキストを使用するだけでは不十分です。警告音を追加してください。 全男性の 8 パーセントは赤緑色盲であるため、赤いテキストに気付かない可能性があります。

- グラフでは、点線や下線などの補助キーなしでは、色を使用しないようにしてください。

  白黒モニタのユーザーが色の効果なしでも理解できるようにする必要があります。 また、ほとんどのユーザーはカラープリンタを持っていません。

整合性チェック

- アプリケーション全体に一貫性を持たせてください。

  関連する主題のフォント、色、または形状を変更しないでください。例えば、アプリケーションの［**OK**］ボタンはすべて同じ形、サイズ、色、およびフォントで設計します。関連するオブジェクトが異なる方法で表示されると、ユーザーは視覚的な手掛かりを使用できないため、アプリケーションに慣れるまでにより長い時間がかかります。 関連するボタンに同じフォント、色、およびサイズを使用して、類似したアクションは類似した方法で表示します。

- 命名規則を採用し、アプリケーション全体で順守します。

  従来のプログラムで名前を変えるには1つの大規模なプログラムを修正すればよかったのですが、オブジェクト指向のプログラムでは、名前を変更するたびに多数のイベントコードを個別に編集する必要があります。 GUI アプリケーションを設計するときは、より厳密に命名規則を順守する必要があります。 これにより、後で整理するための時間を大幅に節約できます。

64 アプリケーション作成に関連するタスク

イベントドリブンNaturalのアプリケーションを作成するには、多数の主要なタスクを実行する必要があります。 このセクションでは、これらのタスクを典型的な実行順序で説明します。 ただし、この順番は柔軟性がないわけではありません。 例えば、ダイアログを設計する過程で何度も念入りにテストしたり、開発の過程で頻繁に作業を保存したりできます。

- アプリケーションを「マルチドキュメントインターフェイス」と「単一ドキュメントインターフェイス」のどちらにするかを決定します。
- 1つ以上のダイアログを作成します。
- ダイアログ（複数可）の属性を設定します。
- ダイアログエレメントを作成し、ダイアログ（複数可）に配置します。
- ダイアログエレメントの属性を設定します。
- （［ダイアログ］メニューの［操作順序設定］を選択して）各ダイアログのタブオーダーを定義します。
- ダイアログ（複数可）の名前を保存します。
- グローバルデータエリアを定義します。
- ローカルデータエリア（複数可）を定義します。
- ダイアログ（複数可）のイベントハンドラコードを作成します。
- ダイアログ（複数可）のインラインサブルーチンを作成します。
- ダイアログエレメントのイベントハンドラコードを作成します。
- ダイアログ（複数可）をSTOWします。
- ダイアログ（複数可）をテスト（チェックおよび実行）します。
- アプリケーションを実行します。

次のチュートリアルでは、最も頻繁に実行されるタスクを紹介します。

65 チュートリアル

このchapterでは、イベントドリブンアプリケーションのコンポーネントを連続して追加する方法を説明する簡単なチュートリアルを紹介します。このチュートリアルでは、1つのダイアログで構成される小さなサンプルアプリケーションをどのように開発するかを説明します。 作成するアプリケーションは、逓減の減価償却計算プログラムです。

この計算プログラムは、例えば、車を購入したときの値段、所有年数、1年ごとの償却率を入力して、車の価値を算出するために使用できます。

任意の時点でチュートリアルを中断してアプリケーションを保存できるため、後で中断したところからチュートリアルを再開できます。

手順 65.1. サンプルアプリケーションを開発するには

1 （ウィンドウに表示する）新規ダイアログを作成します。
2 ダイアログに属性を割り当てます（ウィンドウ設定の決定）。
3 ダイアログ内のダイアログエレメントを作成します（ユーザーとの対話方法の決定）。
4 ダイアログエレメントに属性を割り当てます（属性設定の決定）。
5 アプリケーションのローカルデータエリアを作成します（エンドユーザーが入力した数値をイベントハンドラで使用できるようにする変数を定義）。
6 イベントハンドラコードをダイアログエレメントに関連付けます（ランタイム時のユーザーの行動に対する処理の決定）。
7 アプリケーションをチェック、STOW、および実行します。

ローカルデータエリアを作成することを除き、この手順は、あらゆるイベントドリブンアプリケーションを作成するために必要な最小の手順です。

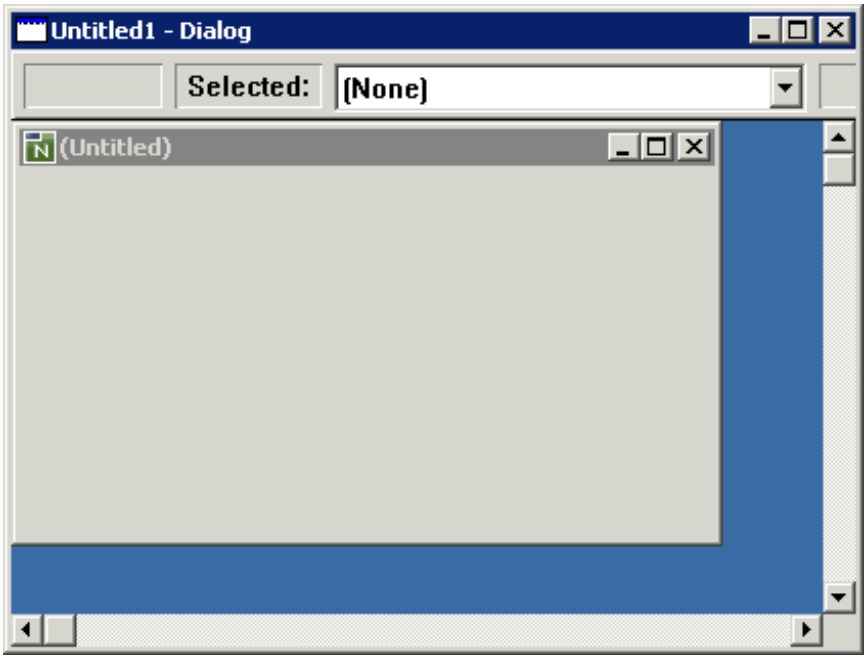
上記の手順の詳細については、以下のトピックで説明します。

ダイアログの作成

手順 65.2. 新規ダイアログを作成するには

1 Natural を呼び出します。
2 ［オブジェクト］メニューを［新規作成］、［ダイアログ］の順に選択します。

Naturalウィンドウによって、ダイアログエディタのメニューおよびツールバーが表示されます。［無題 1 - ダイアログ］という編集ウィンドウが表示されます。この編集ウィンドウは、サイズ変更できます。



編集ウィンドウには、［（無題）］というタイトルの新規ダイアログウィンドウが含まれています。この新規ダイアログウィンドウもサイズ変更できます。また、編集ウィンドウのスクロールバーも使用できます。

ダイアログへの属性の割り当て

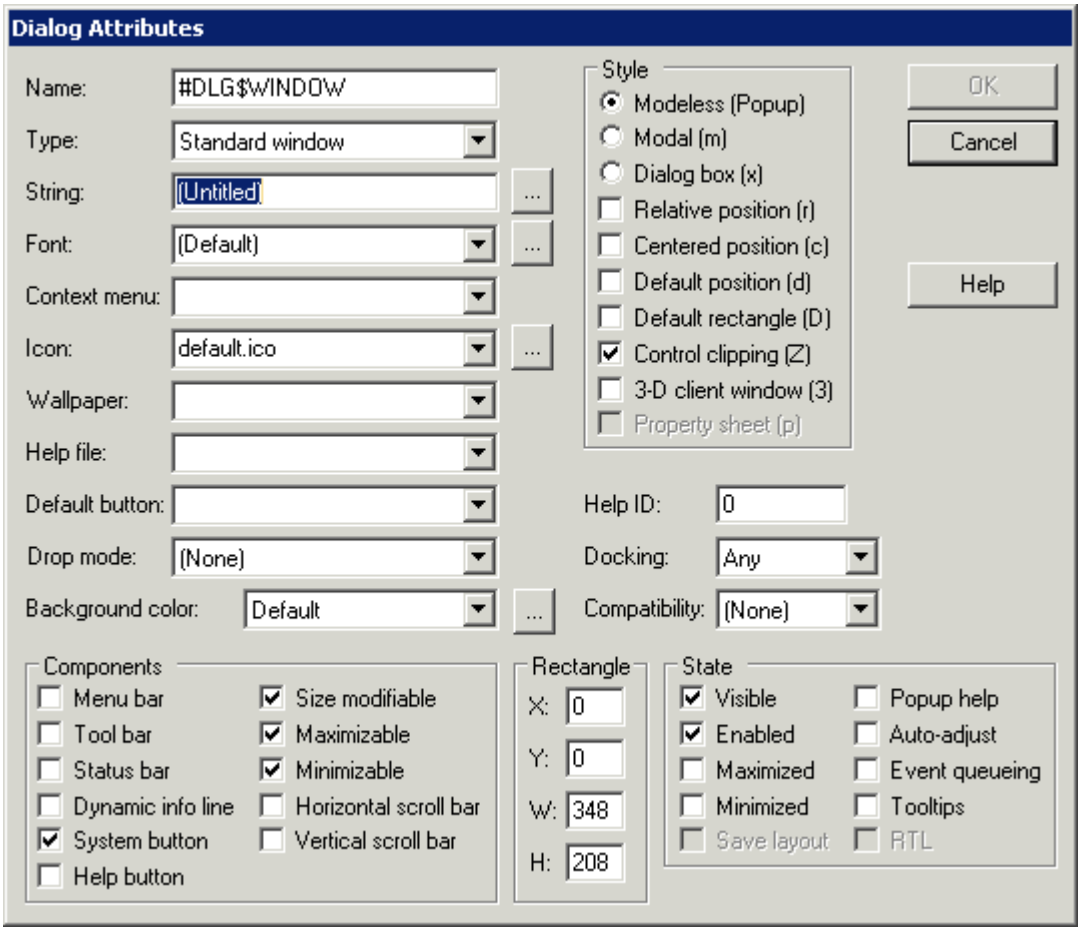
手順 65.3. 属性をダイアログに割り当てるには

1 ［ダイアログ］メニューの［属性］を選択します。

Or:

ダイアログウィンドウ内でダブルクリックします。

［ダイアログ属性］ダイアログボックスが表示されます。



2 ［文字列］テキストボックスにカーソルを移動し、新規ダイアログウィンドウのタイトルを「Degressive Depreciation」と入力します。

3 ［背景色］ドロップダウンリストボックスから、希望する色、例えば［グレー］を選択します。

4 ［**OK**］ボタンを選択します。

［ダイアログ属性］ダイアログボックスが閉じます。

これで、属性 `STRING` は値 "Degressive Depreciation" に、属性 `BACKGROUND-COLOUR-NAME` は希望する色の値（例：`GRAY`）に設定されました。

ダイアログ内のダイアログエレメントの作成

手順 65.4. ダイアログ内のダイアログエレメントを作成するには

1 ［ツール］メニューの［オプション］を選択します。

［オプション］ダイアログボックスが表示されます。

2 ［ダイアログエディタ］ページを選択します。

3 ［グリッド表示］チェックボックスがオンであることを確認し、［線］オプションボタンを選択します。

これにより、グリッドの表示方法が決まります。

4 ［**OK**］ボタンを選択して、変更を確定します。

これで、グリッドをダイアログエレメントの配置と整列に使用できます。

Note: グリッドが表示されない場合は、グリッドの色を変える必要があります（［ダイアログエディタ］ページの［オプション］ダイアログボックス）。これは、グリッドの色と背景色がどちらもグレーで定義されている場合などに発生します。

5 ［挿入］メニューの［テキスト定数］を選択します。

Or:

テキスト定数コントロールを表すツールバーのボタンを選択します。

6 カーソルをダイアログウィンドウの左上隅に移動します。

エディタウィンドウのステータスバーに 50 未満の x 値および y 値が表示されることを確認します。 このとき、テキスト定数コントロールの幅および高さは未定義であることに注意してください。

7 テキスト定数コントロールの位置を固定するためにクリックします。

ダイアログエレメントを表す灰色の矩形が、小さな黒い四角形に囲まれて表示されます。同時に、`#TC-1` を選択していることがステータスバーに表示されます。

8 小さな黒い四角形の 1 つをポイントします。

カーソルの形状が変わり、テキスト定数コントロールをサイズ変更できる方向を示します。

9 `#TC-1` を約 200 の幅にサイズ変更します。

10 テキスト定数コントロールが選択されていることを確認してください。

11 ［編集］メニューの［コピー］を選択します。

12 ［編集］メニューの［貼り付け］を選択します。

新規テキスト定数コントロール #TC-2 が、#TC-1 の上に作成されます。

13 新規テキスト定数コントロールをマウスでクリックしてドラッグするか、またはShiftキーを押したままキーボードの矢印キーを押して、#TC-1 の下に配置します。

14 （同様に）その下にもう 1 つテキスト定数コントロールを作成します。

15 ［挿入］メニューの［入力フィールド］を選択します。

Or:

入力フィールドコントロールを表すツールバーのボタンを選択します。

16 （上記のテキスト定数コントロールで説明したのと同じ方法で）入力フィールドコントロールをダイアログウィンドウの右上隅の、最初のテキスト定数コントロールの隣に配置します。

17 さらに2つの入力フィールドコントロールを作成します（最初のコントロールを複製。上記参照）。これらの入力フィールドコントロールは高さを36にします。入力フィールドコントロールを相互に横方向に整列させ、3つのテキスト定数コントロールに対して縦方向に整列させます（下記参照）。

18 ［挿入］メニューの［プッシュボタン］を選択します。

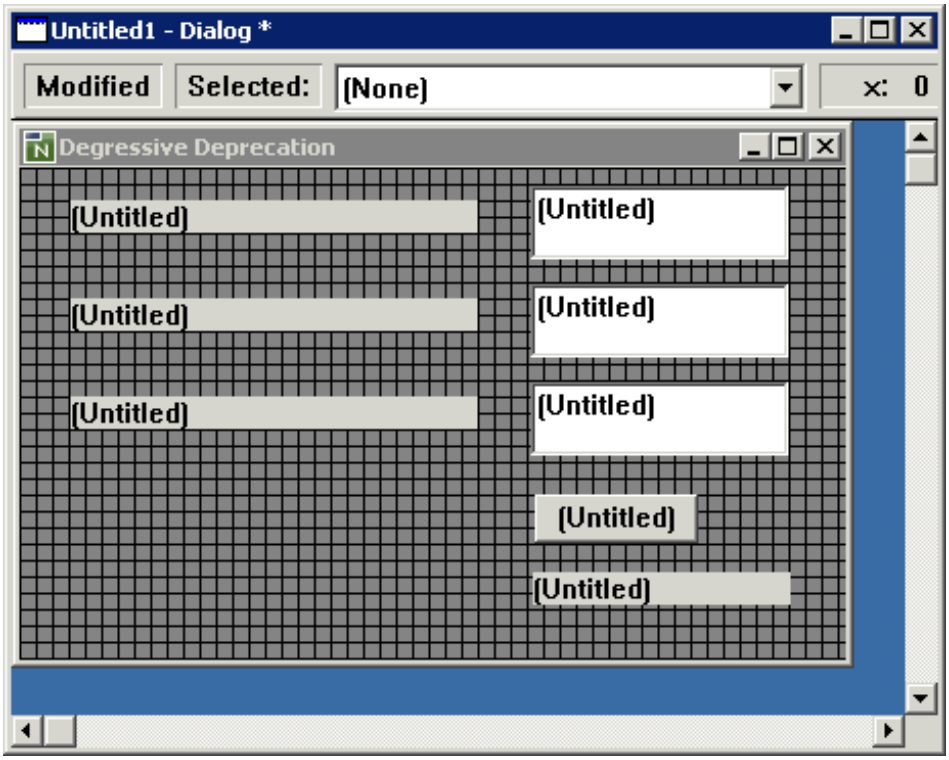
Or:

ブッシュボタンコントロールを表すツールバーのボタンを選択します。

19 3 つの入力フィールドコントロールの下にプッシュボタンコントロールを配置します。

20 プッシュボタンコントロールの下にテキスト定数コントロールを作成します。

次のようなダイアログが表示されます。



ダイアログエレメントへの属性の割り当て

手順 65.5. ダイアログエレメントに属性を割り当てるには

1 最初のテキスト定数コントロール #TC-1 を選択して、［コントロール］メニューの［属性］を選択します。

Or:

最初のテキスト定数コントロール #TC-1 をダブルクリックします。

対応する属性のダイアログボックスが表示されます。

2 ［文字列］テキストボックスに、表示するテキスト文字列を「Initial Value」と入力します。

3 ［**OK**］ボタンを選択します。

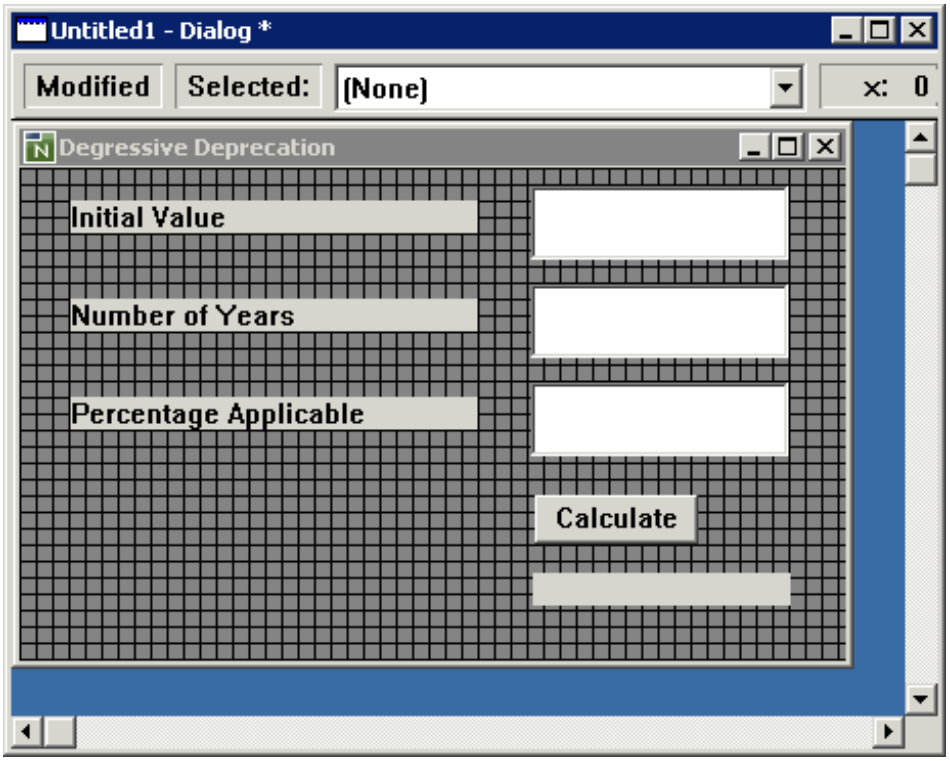
属性ダイアログボックスが閉じます。

4 その下の 2 つのテキスト定数コントロールにテキスト文字列を設定します（#TC-2 には "Number of Years"、#TC-3 には "Percentage Applicable"）。

5 3つすべての入力フィールドコントロールと 4 番目のテキスト定数コントロールから、テキスト文字列（つまり、文字列 "無題"）を削除します。

6 プッシュボタンコントロールにテキスト文字列 "Calculate" を設定します。

次のようなダイアログが表示されます。



アプリケーションのローカルデータエリアの作成

このアプリケーションのローカルデータエリアでは、アプリケーションのリンク変数を定義します。 これらのリンク変数で、エンドユーザーが入力フィールドコントロールに入力した数値を受け取ります。 変数およびその値は、プッシュボタンコントロールのクリックイベントハンドラコードの計算に使用されます。

手順65.6.ローカルデータエリアの作成を準備するには、入力フィールドコントロールにリンク変数を使用する必要があります。

1 最初の入力フィールド `#IF-1` を選択して、［コントロール］メニューの［属性］を選択します。

Or:

最初の入力フィールドコントロール `#IF-1` をダブルクリックします。

対応する属性のダイアログボックスが表示されます。

2 ［文字列］テキストボックスの右にあるブラウズボタン（3つの点が表示されているるボタン）を選択します。

［**#IF-1.STRING** のソース］ダイアログボックスが表示されます。

3 ［リンク変数］オプションボタンを選択（および有効化）します。

4 ［変数名］テキストボックスに「#INITIAL-VALUE」と入力します。

5 ［**OK**］ボタンを選択して［**#IF-1.STRING** のソース］ダイアログボックスを閉じ、再度［**OK**］ボタンを選択して属性ダイアログボックスを閉じます。

6 残りの2つの入力フィールドコントロールにリンク変数名を設定します（`#IF-2` には "#YEAR-NUM"、`#IF-3` には "#PERC-APPLIC"）。

手順 65.7. アプリケーションのローカルデータエリアを作成するには

1 ［ダイアログ］メニューの［ローカルデータエリア］を選択します。

［ダイアログローカルデータエリア］ダイアログボックスが表示されます。

2 ローカルデータを以下のように定義します。

```
1 #INITIAL-VALUE (N6.2)
1 #PERC-APPLIC (N2.1)
1 #YEAR-NUM (N2)
```

3 ［**OK**］ボタンを選択します。

これで、Natural で入力データを処理できます。

ダイアログエレメントへのイベントハンドラコードの関連付け

手順 65.8. イベントハンドラコードを関連付けるには

1 ［**Calculate**］というラベルのプッシュボタンコントロールを選択します。

2 ［コントロール］メニューの［イベントハンドラ］を選択します。

対応するイベントハンドラ定義セクションのダイアログボックスが表示されます。

`CLICK` イベントがあらかじめ選択されています。エンドユーザーがこのプッシュボタンコントロールをクリックすると、特定の Natural コードが起動されます。

3 イベントハンドラ編集エリアに、以下の Natural コードをフリーフォームで入力します。

```
#RESULT:= #INITIAL-VALUE * ( ( ( 100 - #PERC-APPLIC )
/ 100 ) ** #YEAR-NUM )
MOVE EDITED #RESULT (EM=Z(5)9.99) TO #TC-4.STRING
```

4 ［**OK**］ボタンを選択して、ダイアログボックスを閉じます。

アプリケーションのチェック、STOW、実行

手順 65.9. アプリケーションの構文エラーをチェックするには

1 ［オブジェクト］メニューの［**Check**］を選択します。

宣言が必要な変数がある、という内容のNaturalエラーが出力されたダイアログボックスが表示されます。

2 ダイアログボックスの［編集］ボタンを選択します。

ダイアログのコードが表示され、カーソルがエラー（#RESULT）をポイントします。

3 ［キャンセル］ボタンを選択します。

4 ［ダイアログ］メニューの［ローカルデータエリア］を選択します。

5 以下の定義を追加します。

```
1 #RESULT (N6.2)
```

6 ［**OK**］ボタンを選択します。

7 再度アプリケーションをチェックします。

情報メッセージボックスで、チェックが正常に行われたことを確認します。

手順 65.10. アプリケーションを STOW するには

1 ［オブジェクト］メニューの［**Stow**］を選択します。

［ダイアログに名前を付けて **STOW**］ダイアログボックスが表示されます。

2 名前を「Degrdep」と入力します。

3 ［ライブラリ］リストボックスから、ダイアログを STOW するライブラリを選択します。

4 ［**OK**］ボタンを選択します。

情報メッセージボックスで、STOW が正常に行われたことを確認します。

手順 65.11. アプリケーションを実行するには

■ ［オブジェクト］メニューの［**Run**］を選択します。

66 基本的な用語

イベントドリブン Natural では、以下の基本的な用語を使用します。

属性

特定の値が想定されているダイアログまたはダイアログエレメントのプロパティ。

例：ダイアログの HAS-STATUS-BAR 属性を TRUE に設定すると、ダイアログにステータスバーが組み込まれます。

属性には、以下の操作を行うことができます。

操作	結果
クエリ	イベントハンドラコードでは、ランタイム時に属性の値をクエリできます。 例： `#L:= #DLG$WINDOW.HAS-STATUS-BAR`

設定	イベントハンドラコードでは、ダイアログエレメントをダイナミックに作成する前に、属性をグローバル属性リストの値に設定できます。 例：
	
	`#PUSH.STYLE:= 'O'`
	`PROCESS GUI ACTION ADD WITH #W`
	`PUSHBUTTON #PUSH`

変更	イベントハンドラコードでは、ランタイム時に既存のダイアログエレメントの属性値を変更できます。 例： `#PUSH.STYLE:= 'C'`

基本ダイアログ

基本ダイアログは、アプリケーションのメインダイアログです。 基本ダイアログは、コマンド行から、またはオブジェクトリスト経由で開始します。このダイアログを閉じると、アプリケーションの他のダイアログもすべて閉じられます。

コントロール

ダイアログエレメントのタイプ。例：編集エリアコントロール、プッシュボタンコントロール、リストボックスコントロール。

ダイアログ

イベントドリブンアプリケーションでウィンドウとして表され、ウィンドウに直接関連付けられたすべてのイベントハンドラと属性を含む、マップやプログラムに類似したNaturalオブジェクト。 ダイアログには、ウィンドウ、モーダルウィンドウ、ダイアログボックス、MDI 子ウィンドウ、および MDI フレームウィンドウがあります。 これらのウィンドウはハンドルで識別され、ダイアログ全体はシステム変数 `*DIALOG-ID` の値によって表されます。

ダイアログボックス

アプリケーションで排他的に処理される、特別な種類のダイアログ。このダイアログがアクティブな間は、アプリケーションの他のダイアログはすべて無効化され、どのようなユーザー入力も受け入れられません。 ダイアログが `OPEN DIALOG` ステートメントでダイアログボックスを呼び出すと、ダイアログボックスが閉じられるまで、制御は `OPEN DIALOG` ステートメントからダイアログに戻りません。 これにより、アプリケーションはダイアログボックスからダイアログにパラメータを返すことができます。

ダイアログエディタ

ダイアログを作成およびメンテナンスする Natural エディタ。

ダイアログエレメント

ほとんどの場合において、ダイアログエレメントは、エンドユーザーとイベントドリブンアプリケーションとの対話を可能にするウィンドウ内のグラフィック要素です。 ダイアログを作成して属性を設定した後に、プログラマはウィンドウ内にダイアログエレメントを配置します。通常、これはメニューコントロール、必要に応じてツールバー、およびその他の要素（プッシュボタンコントロールや入力フィールドコントロールなど）から構成されます。 要素には、コントロールと項目の 2 つのタイプがあります。

イベント

ユーザーがダイアログエレメントと対話するときに発生します。 また、コード（ユーザー定義イベント）内からイベントを送ることもできます。 例えば、CLICK イベントハンドラコードが定義されているプッシュボタンコントロールをユーザーが マウスでクリックすると、CLICK イベントが発生します。 システム変数 `*EVENT` にはイベント名が設定されます。

イベントハンドラ

ダイアログエレメントに関連付けられているプログラミングコードで、特定のタイプのイベントが発生すると起動されます。

ハンドル

ダイアログエレメントをコード内で識別するためのもので、ハンドル変数に格納されます。例：`#PB-1`.

項目

コントロールの一部である、ダイアログエレメントのタイプ。 例：選択ボックス項目は、選択ボックスコントロールの一部です。

MDI - マルチドキュメントインターフェイス

アプリケーションによる、メインアプリケーションウィンドウ（MDI フレームウィンドウ）内での複数の異なるドキュメントまたは同じドキュメントの複数のビューの管理を可能にします。これらのビューまたはドキュメントは、別々の MDI 子ウィンドウに表示されます。

MDI 子ウィンドウ

MDI アプリケーションの MDI フレームウィンドウ内にドキュメントのビューを表示します。

MDI フレームウィンドウ

MDI アプリケーションにおける、他のすべての子ウィンドウ（ドキュメント）に対する親ウィンドウ。

モーダルウィンドウ

ダイアログで `OPEN DIALOG` ステートメントを使用してモーダルウィンドウを呼び出した場合はモーダルウィンドウが完全に開かれた時点で即座に制御が `OPEN DIALOG` ステートメントからダイアログに戻る点以外は、ダイアログボックスと同様です。

SDI - 単一ドキュメントインターフェイス

MDI アプリケーションとは対照的に、SDI アプリケーションは、ドキュメントウィンドウを格納する MDI フレームウィンドウを持っていません。単一のドキュメントの単一のビューのみを表示します。

ポップアップ

"ポップアップ" スタイルを持つダイアログはモードレスで、デスクトップの任意の位置に移動できます。

ウィンドウ

ウィンドウの基本タイプ。

67 イベントドリブンプログラミングの手法

このセクションでは、次のトピックについて説明します。

- 標準化されたプロシージャの実行
- **Natural** 変数へのダイアログエレメントのリンク
- ダイアログエレメントでの入力の検証
- ダイアログエレメントのクライアントデータの保存および取得
- キャンバスコントロールでのダイアログエレメントの作成
- ツリービューおよびリストビューコントロールでのラベル編集
- **ActiveX** コントロールの操作
- ダイアログエレメントの配列の操作
- コントロールボックスの操作
- 日付／時刻ピッカー（**DTP**）コントロールの操作
- ダイアログバーコントロールの操作
- エラーイベントの操作
- ラジオボタンコントロールグループの操作
- イメージリストコントロールの操作
- リストボックスコントロールおよび選択ボックスコントロールの操作
- リストビューコントロールの操作
- ネスト構造のコントロールの操作
- ダイナミックな情報行の操作
- スピンコントロールの操作
- ステータスバーの操作
- ステータスバーコントロールの操作
- タブコントロールの操作
- ツリービューコントロールの操作
- ダイナミックな情報行およびステータスバーの操作
- ［最大化］／［最小化］／［システム］ボタンの追加
- カラーの定義

- 特定のフォントのテキストの追加
- オンラインヘルプの追加
- ニーモニックキーおよびアクセスキーの定義
- ダイナミックデータ交換 **- DDE**
- オブジェクトのリンクおよび埋め込み **- OLE**

68 はじめに

このドキュメントは、経験豊富な GUI プログラマを対象としており、不可欠のプログラミング手法について説明します。 ダイアログエディタでプログラムする場合、以下の 2 つの方法があります。

- ダイアログエディタのメニューバーとツールバーを使用して新規ダイアログまたはダイアログエレメントを作成し、属性ウィンドウを使用してこれらのダイアログやダイアログエレメントに属性値を割り当てます。 ダイアログエディタは対応する Natural コードを内部的に生成します。
- イベントハンドラセクションまたは内部サブルーチンセクションを開いて、Natural コードを明示的に指定します。 このコードは、内部的に生成されたコードに追加されます。 また、パラメータデータエリア、グローバルデータエリア、およびローカルデータエリアを、対応する定義セクションに入力することもできます。

ダイアログエディタのメニューバーを［オブジェクト］、［リスト］の順に選択すると、現在のダイアログの生成コードおよび指定コードを表示できます。

ダイアログエディタを使用したプログラム方法の実践的なデモが必要な場合は、SYSEXEVT ライブラリを参照してください。 このライブラリには、基本機能を紹介するサンプルダイアログが含まれています。 ただし、サンプルダイアログにアクセスする前に、README ファイルをお読みください。 その後、MENU ダイアログを実行してください。

Note: ダイアログエディタでは、ストラクチャードモードでコードを記述する必要があります。

ダイアログを使用する Natural アプリケーションを実行する場合は、アプリケーションを開始するダイアログを使用する必要があります。

イベントドリブンプログラミングの詳細については、「*イベントドリブンプログラミングについて*」を参照してください。

69 ダイアログをオープンおよびクローズする方法

このchapterでは、次のトピックについて説明します。

ダイアログを開く

イベントドリブンアプリケーションは、基本となるダイアログを実行することによって開始されます。 通常、エンドユーザーが起動したイベントによって、他のダイアログが開始されます。 アプリケーションは、基本となるダイアログが閉じられると終了します。

手順 69.1. イベントドリブンアプリケーション内の任意の箇所からダイアログを開くには

- `OPEN DIALOG` ステートメントを使用します。

このステートメントによって、ダイアログがロードされ、ダイアログを開くときの処理が実行されます。

"ダイアログボックス" スタイルのダイアログ以外は、ダイアログが開くと制御が戻ります。 ダイアログボックススタイルのダイアログの場合は、ダイアログが終了したときにのみ制御が戻ります。

開くダイアログのパラメータデータエリアに `BY VALUE` でパラメータが宣言されているか、またはダイアログが "ダイアログボックス" スタイルである場合を除き、渡されるパラメータには、ダイアログを開く処理（BEFORE-OPEN イベントおよび AFTER-OPEN イベント）の間のみアクセスできます。

イベントドリブン型Naturalアプリケーション内の任意の箇所でダイアログを開くには、以下の構文を使用します。

```
OPEN DIALOG operand1    [USING]     [PARENT] operand2
                     [  [GIVING]    [DIALOG-ID] operand3  ]
                     [  WITH     { operand4...           } ]
                                 { PARAMETERS-clause     }
```

オペランド

Operand1 は、開かれるダイアログボックスの名前です。 *PARAMETERS-clause* を使用する場合、*operand1* は定数（カタログ化されたダイアログの名前）である必要があります。

Operand2 は、親のハンドル名です。

Operand3 は、ダイアログの作成時に返される一意のダイアログ ID です。 これは、フォーマット／長さ I4 で定義する必要があります。

ダイアログへのパラメータ渡し

ダイアログボックスを開いたときに、パラメータがダイアログボックスに渡されます。

ダイアログに渡すパラメータは *operand4* として指定します。

PARAMETERS-clause を使用すると、選択的にパラメータを渡すことができます。

```
PARAMETERS [parameter-name=operand 4] _ END-PARAMETERS
```

Note: *operand1* が英数字定数で、ダイアログがカタログ化されている場合にのみ、*PARAMETERS-clause* を使用できます。

Parameter-name は、ダイアログのパラメータデータエリアセクションに定義されたパラメータの名前です。

渡すオペランドとパラメータ間のフォーマット／長さの矛盾を避けるには、『ステートメント』ドキュメントに記載されている、DEFINE DATA ステートメントの BY VALUE オプションの説明を参照してください。

operand4 のみを使用してパラメータを渡すと、以下の例のようなダイアログが開かれます。

```
/* The following parameters are defined in the calling dialog's parameter
/* data area (not in the parameter data area of the dialog to be opened):
 1 #MYDIALOG-ID (I4)
 1 #MYPARM1 (A10)
/* Pass the operands #MYPARM1 and 'MYPARM2' to the parameters #DLG-PARM1 and
/* #DLG-PARM2 defined in the dialog to be opened:
  OPEN DIALOG 'MYDIALOG' USING
  #DLG$WINDOW GIVING
  #MYDIALOG-ID WITH
  #MYPARM1 'MYPARM2'
```

PARAMETERS-clause を使用して選択的にパラメータを渡すと、以下の例のようなダイアログが開かれます。

```
/* The following parameters are defined in the calling dialog's parameter
/* data area (not in the parameter data area of the dialog to be opened):
 1 #MYDIALOG-ID (I4)
 1 #MYPARM1 (A10)
/* Pass the operands #MYPARM1 and 'MYPARM2' to the parameters #DLG-PARM1 and
/* #DLG-PARM2 defined in the dialog to be opened:
  OPEN DIALOG 'MYDIALOG' USING
```

```
  #DLG$WINDOW GIVING
  #MYDIALOG-ID WITH PARAMETERS
  #DLG-PARM1=#MYPARM1
  #DLG-PARM2='MYPARM2'
END-PARAMETERS
```

データの生成、引き渡し、およびチェック時の持続性

Natural では 「持続性」 という用語は、基本ダイアログのローカルデータエリアに定義されたデータはダイアログの稼働中は存在を保証されていることを示すために使用します。 グローバルデータエリアはアプリケーション実行中に変更される可能性があるため、グローバルデータエリアに定義されたデータには持続性がありません。

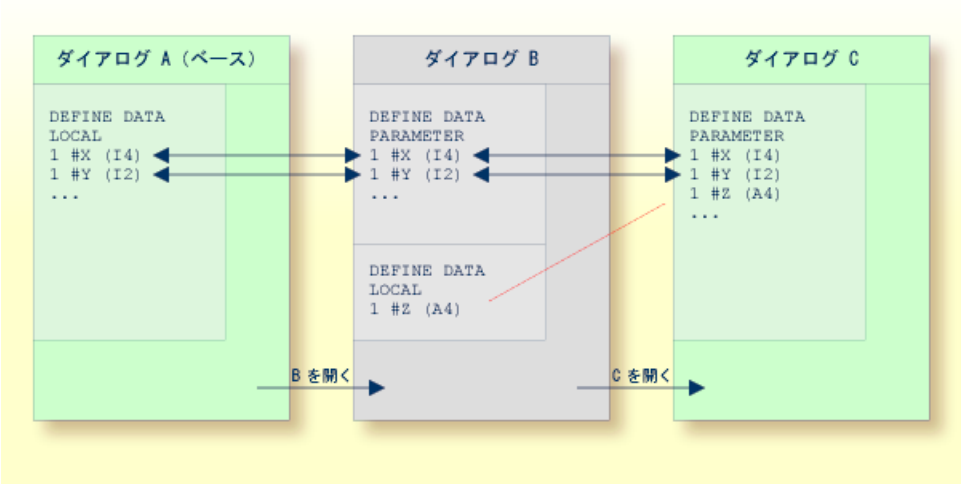
ダイアログを開くときにパラメータデータエリアを内部的に保存することによって、持続性のあるデータへの参照が維持されます。 以下の場合にこの参照が再使用されます。

- ダイアログエレメントがイベントを受け取ったとき
- 1つのダイアログから別のダイアログに渡されるすべてのパラメータが持続的であるとき、つまり基本ダイアログのローカルデータエリアを参照するとき

パラメータは以下の場合にアクセスできます。

- ダイアログを開く際の BEFORE-OPEN および AFTER-OPEN イベント処理の間
- すべてのパラメータが基本ダイアログのローカルデータエリアを参照する場合

以下の例は、2つのパラメータが持続的で、もう1つのパラメータが持続的でない場合を示しています。 まず、基本ダイアログ A が ダイアログ B を開いてパラメータ #X と #Y を渡します。 次に、ダイアログ B がダイアログ C にパラメータ #X と #Y を渡します。 現在のダイアログ C 内のパラメータ #X と #Y は、ダイアログ B が閉じられても持続性があります。 ただし、ダイアログ C を開くときにダイアログ B 独自のパラメータ #Z をダイアログ B から渡す場合、ダイアログ B が閉じられるとそのローカルデータエリアへの参照が有効でなくなるため、パラメータ #Z は持続的ではありません。 したがって、ダイアログ C のパラメータ #Z はアクセスできなくなります（#Z は基本ダイアログのローカルデータエリアを参照しません）。



ダイアログを開くときの処理手順

このセクションでは、ダイアログを開くときの処理内容について説明します。 ダイアログを開くには、コマンド行などから実行するか、または OPEN DIALOG ステートメントを使用して起動します。

- ダイアログオブジェクトがロードされ、実行が開始されます。
- BEFORE-ANY イベントハンドラセクションが実行されます。このときのシステム変数 *EVENT の値は OPEN です。
- BEFORE-OPEN イベントハンドラセクションが実行されます。
- ダイアログエディタの指定に従って、ダイアログウィンドウが作成されます。
- BEFORE-ANY イベントハンドラセクションが実行されます。 *EVENT = AFTER-OPEN。
- ダイアログエディタの指定に従って、すべてのダイアログエレメントが作成されます。
- VISIBLE = FALSE のものを除き、ダイアログウィンドウとすべてのダイアログエレメントが表示されます。
- AFTER-OPEN イベントハンドラセクションが実行されます。
- AFTER-ANY イベントハンドラセクションが実行されます。 *EVENT = AFTER-OPEN。
- AFTER-ANY イベントハンドラセクションが実行されます。 *EVENT=OPEN（ダイアログの STYLE 属性値が "ダイアログボックス" の場合）。

ダイアログを閉じる

ダイアログをダイナミックに閉じるには、次のように指定します。

```
CLOSE DIALOG [USING] [DIALOG-ID] { operand1    }
                                 { *DIALOG-ID  }
```

Operand1 は、OPEN DIALOG ステートメントで返されるダイアログ ID です。

例：

```
CLOSE DIALOG *DIALOG-ID /* Close the current Dialog
```

ダイアログが画面から消去され、メモリから削除されます。 ダイアログに関連付けられていたローカルデータもすべて削除されます。

Note: モーダルダイアログがダイアログ階層で子である場合、モーダルダイアログの親を閉じるとデッドロックが生じるので、親を閉じないでください。

operand1

Operand1 は、閉じるダイアログの名前です。

現在のダイアログを閉じるには、*DIALOG-ID を指定します。

属性値の初期化

ダイアログを開いたり閉じたりするための条件を指定できます。この条件指定は、BEFORE-OPEN、AFTER-OPEN、および CLOSE の各イベントに適用されます。 これらの条件を使用して、ダイアログ内の属性値を初期化できます。

以下に、AFTER-OPENイベントハンドラコードの例を示します。関連付けられている入力フィールドにデータを入力した後に押す必要のあるプッシュボタンの前景色に、赤を設定します。

```
DEFINE DATA LOCAL
...
  1  #OK-BUTTON HANDLE OF PUSHBUTTON
  1 #CALC-BUTTON HANDLE OF PUSHBUTTON
  1 #SAVE-BUTTON HANDLE OF PUSHBUTTON
  1 #CONVERT-BUTTON
HANDLE OF PUSHBUTTON
...
END-DEFINE
...
#OK-BUTTON.FOREGROUND-COLOUR-NAME := RED
#CALC-BUTTON.FOREGROUND-COLOUR-NAME := RED
#SAVE-BUTTON.FOREGROUND-COLOUR-NAME := RED
#CONVERT-BUTTON.FOREGROUND-COLOUR-NAME := RED
```

ダイアログを開いて画面に表示する前にダイアログエレメントおよびダイアログの属性値を変更する場合、ダイアログエレメントおよびダイアログウィンドウはまだ作成されていないため、"BEFORE OPEN" イベントハンドラコードで設定を変更しないでください。 代わりに、ダイアログエディタでダイアログを作成し、［ダイアログ属性］ウィンドウで `VISIBLE` 属性を `FALSE` に設定します。 そして（ハンドルが使えるようになったときに）、AFTER-OPEN イベントハンドラコードですべての属性値を変更します。 その後、`VISIBLE = TRUE` を指定して、ダイアログを表示可能にします。

例：

```
DEFINE DATA LOCAL
...
 1 #DIA-1 HANDLE OF DIALOG
 1 #OK-BUTTON HANDLE OF PUSHBUTTON
 1 #CALC-BUTTON HANDLE OF PUSHBUTTON
...
END-DEFINE
...
/* AFTER OPEN event-handler code section
...
#OK-BUTTON.FOREGROUND-COLOUR-NAME := RED
#CALC-BUTTON.FOREGROUND-COLOUR-NAME := RED
#DIA-1.VISIBLE := TRUE
```

70 ダイアログの拡張ソースコードを編集する方法

このchapterでは、次のトピックについて説明します。

拡張ソースコードフォーマットとは

拡張ソースコードフォーマットを使用すると、ダイアログエディタによって生成されたソースコードを編集できます。 拡張ソースコードは、プログラムエディタウィンドウで編集します。 ダイアログを編集すると、ダイアログエディタによってその結果が内部構造に保存されます。 ダイアログで他のシステムコマンドを保存、STOW、リスト、または実行すると、これらの内部構造からソースコードが生成されます。 また、プログラムエディタのソースコードウィンドウをリフレッシュしたときにも、コードが生成されます。

拡張ソースコードは、他の Natural ユーザーコードと同じように編集できます。 ただし、拡張ソースコード構文は、多くの形式的な規則に従う必要があります。 拡張ソースコード構文の詳細については、『エディタ』ドキュメントの*ダイアログエディタ*部分にある「*拡張されたソースコードフォーマット*」を参照してください。

プログラムエディタのソースコードウィンドウで編集したダイアログ上でシステムコマンドを実行すると、ダイアログエディタによって内部構造が更新され、ソースコードウィンドウがリフレッシュされます。

Note: ダイアログエディタでは、イベントハンドラなどのユーザーコードセクションにあるコードレイアウトのみが保存されます。

ダイアログエディタは、次のソース形式をサポートします。

- 213. これは、Natural バージョン 2.1.3 で生成されるフォーマットです（新しい次元）。 入力でのみサポートされています。 Natural バージョン 3.1 およびバージョン 3.2 では、2.1.3 フォーマットを生成できません。
- 22C。 これは、Natural バージョン 2.2.2 で生成されるフォーマットです。 Natural for Windows、Natural for UNIX、および Natural for OpenVMS では、このフォーマットのダイアログを生成できません。 このフォーマットも入力に対してのみサポートされています。
- 22D。 このフォーマットは、今後の標準となる 「拡張」 ソースコードフォーマットです。 Natural バージョン 2.2.3 以上で、ダイアログのコンパイル、保存、および編集のために生成されます。

拡張ソースコードフォーマットには以下の特徴があります。

- ダイアログソースは、変換なしで読み込みおよび印刷が可能です。
- ダイアログソースは、完全にドキュメント化された有効な Natural 構文だけで構成されます。
- ダイアログソースは、テキストの検索や置換などのプログラムエディタ機能を使用して、テキスト編集できます。
- ダイアログソースは、Natural デバッガに表示できます。

- ダイアログソースは、213 または 22C フォーマットのソースより大きくなります（1.25～3.5 倍）。
- ダイアログエディタで作成可能なコードはすべて、手動でコーディングすることもできます。例えば、プッシュボタンコントロールをユーザーインターフェイスに 「作成」 すると、対応するコードが暗黙的に生成されます。また、プログラムエディタの機能を提供するソースコードウィンドウを使用して、プッシュボタンコントロールを明示的に作成することもできます。
- ソースコードウィンドウまたはダイアログウィンドウを選択して、ダイアログエディタとプログラムエディタを切り替えることができます。 どちらのウィンドウで編集する場合も、更新を同期する必要があります。（図を使用して）ダイアログを修正すると、ソースコードウィンドウがロックされ、そこで変更を加えることができなくなります。 これに対応して、ソースコードを変更すると、ダイアログウィンドウがロックされ、そこで変更を加えることができなくなります。 エディタがロックされると、ステータスバーに "ロック" と表示されます。

古いフォーマットのダイアログは以下のように処理されます。

- ダイアログは、ダイアログエディタで処理されるまで変更されません。 ダイアログは、古いフォーマットでコンパイルおよび実行できます。
- ダイアログは、ダイアログエディタにロードするときに、新しいフォーマットで保存されます。 ダイアログを拡張フォーマットで保存する場合、ローカルデータエリア NGULKEY1 を含める必要があります。 ダイアログを保存するとストレージサイズが増えることに注意してください。
- "拡張リストモード"オプションを使用可能に設定してダイアログをリストまたは印刷すると、ダイアログは拡張ソースコードフォーマットで出力されます。

ダイアログエディタとプログラムエディタとの矛盾の回避

拡張ソースコードフォーマットを編集する場合、プログラムエディタで使用可能な構文要素の一部にダイアログエディタで使用できないものがあることに注意してください。 拡張ソースコード編集は、ダイアログエディタ以外を使用する新しいプログラミング手法を意図するものではありません。

- 構文的には、ダイアログエレメントの数値座標（RECTANGLE-X 属性値）を変数参照で置き換えることができます。 ただし、ダイアログエディタは変更を同期するときにこの構文を受け付けないため、ソースコードを要求するコマンドが発行されるとプロンプトを出力します。
- ダイアログエディタは、変数が宣言されていなくても変数の STRING 属性への参照を受け付けますが、コンパイラは受け付けません。

ユーザーコード以外のセクションでは、コンパイラとダイアログエディタの両方で受け付けられるコードだけを追加することによって、このような矛盾を避ける必要があります。

イベントハンドラセクションや外部／内部サブルーチンなどのユーザーコードセクションでは、ダイアログエディタはプログラミング手法の選択を制限しません。 ただし、これらのセクションでは、ビジュアル編集はサポートされません。

一般的に、ダイアログエディタによって生成されるコードを開始地点として使用する場合は特に、複数の手法を組み合わせて使用するのが最適です。

Note: ダイアログエディタでダイアログエレメントをクリップボードにコピーしてユーザーコードに貼り付けると、テキストとして表示されます。

拡張ソースコードフォーマットの使用方法

手順 70.1. ダイアログを拡張ソースコードフォーマットで編集するには

1 ダイアログをダイアログエディタにロードします。

2 ［ダイアログ］メニューの［ソースコード］を選択します。

Or:

［ソースコード］ツールバーボタンを選択します。

Or:

Ctrl + Alt + C キーを押します。

ダイアログのソースコードウィンドウが表示され、プログラムエディタがロードされます。このエディタを使用すると、テキスト文字列のスキャンや置き換えなどを実行できます。プログラムエディタの使用方法の詳細については、「*プログラムエディタ*」を参照してください。

拡張ソースコードフォーマットの構文形式については、『*エディタ*』ドキュメントの*ダイアログエディタ*部分にある「*拡張されたソースコードフォーマット*」を参照してください。

拡張ソースコードは通常のコードと同様にリストおよび印刷できます。また、［編集］メニューの［検索］コマンドを使用して文字列を検索することもできます。

Note: このオプションを使用して文字列を置き換えると、ダイアログエディタと矛盾するダイアログソースができる可能性があります。

71 ダイアログ、コントロール、および項目が階層的に関連付けられる仕組み

ダイアログおよびダイアログエレメントは階層的に構成されます。 通常、ダイアログウィンドウは多くのコントロールを含みます。 コントロールは、ウィンドウまたはコンテナとして機能できる他のコントロールの子です。 コントロールには、多くの項目を含めることができます。 例えば、リストボックスコントロールは、複数のリストボックス項目を含めることができ、 これら項目の親となります。

ダイアログ自体も階層的に構成されます。 OPEN DIALOG ステートメントを指定するたびに、新規作成するダイアログの親をパラメータとして指定する必要があります。 このパラメータは NULL-HANDLE または既存ダイアログのハンドルです。 NULL-HANDLE を指定すると、ダイアログは他のダイアログではなくデスクトップに属します。 つまり、ダイアログはアプリケーション内の他のダイアログに関係なく閉じたり、最小化したりできます。 既存ダイアログを親として持つダイアログは、親ダイアログが閉じたり最小化したりすると、親と同じように動作します。

アプリケーションの最初のダイアログは特別な役割を果たし、基本ダイアログと呼ばれることがあります。 基本ダイアログを閉じると、アプリケーション内の他のダイアログもすべて、基本ダイアログの子であるかどうかに関係なく閉じられます。

同一階層レベルの子はすべて生成順にソートされます。 したがって、各ダイアログエレメントは、自分の親、（同一階層レベルの）自分の前と後ろ、および最初と最後の子（存在する場合）を常に「認識」していることになります。 この情報は以下の属性を使用して知ることができます。

- PARENT
- PREDECESSOR
- SUCCESSOR
- FIRST-CHILD
- LAST-CHILD

これらの属性はダイアログエレメントのハンドル値を含みます。ハンドル値がNULLの場合、ダイアログエレメントは親、同レベルの前、後ろ、または子を持ちません。 以下の例は、ダイアログの全ダイアログエレメントの調べ方を示しています。

例 **1**

```
1 #CONTROL HANDLE OF GUI

#CONTROL := #DLG$WINDOW.FIRST-CHILD
REPEAT UNTIL #CONTROL = NULL-HANDLE
  ...
  #CONTROL := #CONTROL.SUCCESSOR
END-REPEAT
```

リストボックスコントロールおよびリストボックス項目には、さらに1つの属性があります。

SELECTED-SUCCESSORは、リストボックスコントロール自体またはリストボックス項目に対して設定できます。 これはリストボックスコントロール内の次に選択される項目を指します。 リストボックスコントロール自体の場合は、最初に選択される項目を示します。

例 **2**

```
1 #ITEM HANDLE OF LISTBOXITEM

#ITEM := #LISTBOX.SELECTED-SUCCESSOR
REPEAT UNTIL #ITEM = NULL-HANDLE
  ...
  #ITEM := #ITEM.SELECTED-SUCCESSOR
END-REPEAT
```

例2は、複数選択（MULTI-SELECTION属性）が可能なリストボックスコントロールで選択されたすべての項目を見つけるために必要なクエリです。

72 ダイアログエレメントを定義する方法

このchapterでは、次のトピックについて説明します。

はじめに

ダイアログエレメントはハンドルによって一意に識別されます。 ハンドルとは、ダイアログエレメントの作成時に返されるバイナリ値のことです。 ハンドルは、ダイアログの DEFINE DATA ステートメントに定義する必要があります。

以下の方法でハンドルを定義できます。

- ダイアログエディタでダイアログまたはダイアログエレメントを作成します。この場合、ハンドル定義が生成されます。
- ダイアログのグローバル、ローカル、またはパラメータの各データエリアに定義を明示的に入力します。
- サブプログラムまたはサブルーチンに定義を明示的に入力します。

Note: ActiveX コントロールのハンドルの定義方法は、以下に説明する標準のハンドル定義方法とは少し異なります。 これについては、「*ActiveX コントロールの操作*」に記載されています。

ハンドルは DEFINE DATA ステートメント内に以下のように定義されます。

```
level handle-name [(array-defintion)] HANDLE OF dialog-element-type
```

ハンドルは任意の *level* に定義できます。

Handle-name は、ハンドルに割り当てる名前です。ユーザー定義変数の命名規則が適用されます。

Dialog-element-type は、ダイアログエレメントのタイプです。 設定可能値は、TYPE 属性の値です。 これは再定義できず、グループの再定義に含めることができません。

例

```
1 #SAVEAS-MENUITEM HANDLE OF MENUITEM
1 #OK-BUTTON (1:10) HANDLE OF PUSHBUTTON
```

ハンドルを定義すると、オペランドを指定できるNaturalステートメントのハンドル属性オペランドに *handle-name*を使用できます。 ハンドル属性オペランドを使用すると、定義された *dialog-element-type* に対し、例えば属性値のクエリ、設定、または変更をダイナミックに実行できます。 これは、ダイアログエディタにおける最も重要なプログラミング手法です。 詳細については、「*ダイアログエレメントを操作する方法*」を参照してください。

2つの異なるダイアログに同じダイアログエレメントのハンドル名があっても、PARENT属性（異なる2つのPARENT値）によって、Naturalでは2つのハンドルを確実に区別できます。ハンドルはパラメータとして渡されるか、または1つのハンドル変数から別のハンドル変数に割り当てられます。

GUIのハンドル

1つのダイアログエレメントを指すハンドルタイプの他に、総称的なハンドルタイプHANDLE OF GUIがあります。イベントハンドラコードでHANDLE OF GUIを使用すると、任意のタイプのダイアログエレメントのハンドルを参照できます。

例えば、あるレベルですべてのダイアログエレメントの属性値に対してクエリを実行している場合、すなわち複数のダイアログエレメントに順番にクエリをする場合、このクエリ中に、どのタイプのダイアログエレメントに対して次にクエリが実行されるかわからないときにこれが役に立つことがあります。GUIハンドルを使うと、タイプに関係なく、次のダイアログエレメントをクエリできます。その結果、各ダイアログエレメントの属性値を個別にクエリする必要がなくなるため、コーディングを大幅に削減できます。

例：

```
...
1 #CONTROL HANDLE OF GUI
...
#CONTROL := #DLG$WINDOW.FIRST-CHILD
REPEAT UNTIL #CONTROL = NULL-HANDLE
  ...
  #CONTROL := #CONTROL.SUCCESSOR
END-REPEAT
```

NULL-HANDLE

HANDLE定数NULL-HANDLEは、HANDLEのNULL値をクエリ、設定、または変更するために使用できます。このNULL値は、ダイアログエレメントが（明示的に作成された場合でも）存在しないことを意味します。

例：

```
DEFINE DATA PARAMETER
  1 #PUSH HANDLE OF PUSHBUTTON
END-DEFINE
...
IF #PUSH = NULL-HANDLE
...
```

HANDLE 定数 NULL-HANDLE は、HANDLE 変数または HANDLE 形式の属性の NULL 値を表します。HANDLE 変数の場合は、この値は、式 *handle.attribute* がグローバル属性リストを指すことを示します。 属性の場合は、値が現在設定されていないことを示します。

73 ダイアログエレメントを操作する方法

このchapterでは、次のトピックについて説明します。

はじめに

ダイアログエレメントを操作するために、Naturalではハンドル属性オペランドが提供されています。Naturalステートメントでオペランドを指定する場合は常に、ハンドル属性オペランドを使用します。 これは、イベントハンドラコードにおける最も重要なプログラミング手法です。

Important: あらかじめハンドルを定義しておく必要があります。

Note: ActiveX コントロールの操作方法は、以下に説明する標準の方法とは少し異なります。 これについては、「*ActiveX コントロールの操作*」に記載されています。

ハンドル属性オペランドは以下のように指定します。

```
handle.name-attribute.name[(index-specification)]
```

handle-name は DEFINE DATA ステートメントの HANDLE 定義で定義した、*dialog-element-type* のハンドルです。

*attribute-name*は、ハンドルの *dialog-element-type*に対して有効でなければならない属性の名前です。

例

```
1 #PB-1 HANDLE OF PUSHBUTTON   /* #PB-1 is a handle-name of the
                               /* dialog-element-type PUSHBUTTON
RESET #PB-1.STRING...          /* #PB-1.STRING is the handle attribute operand
                               /* where STRING is a valid attribute-name of the
                               /* dialog-element-type PUSHBUTTON

1 #RB-1(1:5) HANDLE OF RADIOBUTTON /* #RB-1 is an array of five RADIOBUTTONs
```

```
IF #RB-1.CHECKED(3) = CHECKED      /* If the third radio-button control is
   THEN...                         /* checked ...
```

属性値のクエリ、設定、および変更

ほとんどのアプリケーションで、以下のことを行う必要があります。

- ダイアログエレメントを作成する前に属性値を設定する。
- ダイアログエレメントを作成した後に値を変更する。
- 属性値をクエリする。

場合によっては、例えば、ラジオボタンコントロールのチェック状態をクエリしたり、メニュー項目を使用不可に変更したりするために、処理中にいくつかの属性を変更およびクエリする必要があります。

これは、ASSIGN、MOVE、または CALLNAT ステートメントなどで実行できます。

例

```
1 #PB-1 HANDLE OF PUSHBUTTON      /* #PB-1 is a handle-name of the
...                               /* dialog-element-type PUSHBUTTON
#PB-1.STRING:= 'MY BUTTON'        /* Set or modify the value of the STRING
                                  /* attribute to 'MY BUTTON'
#TEXT:= #PB-1.STRING              /* Query the value of the STRING attribute
                                  /* and assign the value to #TEXT
CALLNAT 'SUBPGM1' #PB-1.STRING    /* Query the value of the STRING attribute
                                  /* and pass it on to the subprogram
```

上記の例の 3 つのステートメントの 1 番目のように *handle-name* 変数をステートメントの左辺だけに使用すると、属性値が設定または変更されます。つまり、指定した *operand* の値が属性値に割り当てられます。

2番目のステートメントのように *handle-name*変数をステートメントの右辺に使用すると、属性値がクエリされます。つまり、属性値が *operand* に割り当てられます。

ハンドルは、（指定した Natural コードで明示的に、またはダイアログエディタで暗黙的に）いったん定義すると、ほとんどの Natural ステートメントで使用できます。 ただし、特定のダイアログエレメントに対しては、特定の属性だけをクエリ、設定、または変更できます。 属性に定義できる値については、『*ダイアログコンポーネントリファレンス*』の「*属性*」を参照してください。

ほとんどの属性値には正確なデータタイプを指定しますが、ハンドル属性オペランドにはMOVEでの互換性のある値を指定すれば十分です。 この規則は Natural 変数の場合と同じです。

制限

以下のステートメントでは、ハンドル属性オペランドを使用しないでください。

AT BREAK、FIND、HISTOGRAM、INPUT、READ、READ WORK FILE

代わりに、ユーザー定義変数を使用できます。

数値／英数字割り当て

英数字属性に数値オペランドを割り当てると、これらの属性の値は非表示形式になります。Natural の算術代入規則が適用されます。

表示形式にする場合は、MOVE EDITED を使用できます。

例

```
#PB-1.STRING:= -12.34                          /* Non-displayable format
MOVE EDITED #I4 (EM = -Z(9)9) TO #PB-1.STRING /* Displayable format
```

以下の編集マスクを使用して、さまざまなフォーマット／長さの数値オペランドを定義できます。

フォーマット／長さ	編集マスク
I1	-ZZ9
I2	-Z(5)9
I4	-Z(9)9
N*n*.*m*/P*n*.*m*	-Z(*n*).9(*m*)

74 ダイアログエレメントをダイナミックに作成および削除する方法

このchapterでは、次のトピックについて説明します。

はじめに

通常、ダイアログエレメントはダイアログエディタを使用してダイアログに追加されます。 ただし、ダイアログエレメントは、ダイナミックに作成したり削除したりすることもできます。 例えば、ダイアログのレイアウトがコンテキストに強く依存している場合に、ダイアログエレメントをダイナミックに作成および削除します。

PROCESS GUI ステートメントの ADD アクションを使用すると、ダイアログエレメントがダイナミックに作成されます。 このアクションは、新規作成されたダイアログエレメントにハンドルを返します。 このハンドルは、ダイアログエレメントが作成されるとただちに、作成されたダイアログエレメントに指定された一連の属性を指し示します。

Note: ActiveX コントロールの作成方法は、以下に説明する標準の方法とは少し異なります。 これについては、「*ActiveX コントロールの操作*」に記載されています。

使用可能なアクション、および渡すことのできるパラメータの詳細については、「*標準化されたプロシージャの実行*」を参照してください。

グローバル属性リスト

"*handlename.attributename*"形式のハンドル属性オペランド（例：#PB-1.STRING）を変更することによって、特定のダイアログエレメントの属性値を変更します。 ダイアログエレメントが未作成でハンドル変数が初期値（NULL-HANDLE）の場合、ハンドル属性オペランド *handlename.attributename* はグローバル属性リストを参照します。

グローバル属性リストとは任意のダイアログエレメントに定義されたすべての属性の集合です。 Naturalにはこの集合が1つ含まれています。 ダイアログエレメントは、作成されるときに常にグローバル属性リストから属性を 「継承」 します。 ダイアログエレメントが WITH PARAMETERS オプション付きの PROCESS GUI ACTION ADD ステートメントで作成されるときは、属性は継承されません。

ダイアログエレメントのスタティックおよびダイナミック作成

それぞれの属性設定を持つダイアログエレメントを（ダイアログエディタで）スタティックに定義するには、まずグローバル属性リスト内の属性に適切な値を設定し、それからダイアログエレメントを作成する必要があります。 作成後も、グローバル属性リスト内の属性値は変わりません。 次に作成されるダイアログエレメントは、変更された属性以外は、グローバル属性リストから前と同じ属性値を取得します。

「AFTER OPEN」 イベントハンドラで検出される、グローバル属性リストのステータスは、スタティックに定義されるダイアログエレメントの影響を受けます。 したがって、「AFTER OPEN」イベントハンドラでダイアログエレメントのダイナミックな作成を始める前に、PROCESS GUI ステートメントの RESET-ATTRIBUTES アクションを使用して属性をリセットする必要があります。これにより、ダイアログエレメントがグローバル属性リストから予期しない値を継承するのを防ぐことができます。 この継承問題を避けるには、WITH PARAMETERS オプション付きの PROCESS GUI ステートメントの ADD アクションを使用します。

異なるタイプのダイアログエレメントで使用すると異なる意味を持つ属性値を指定すると、予期しない結果になることがあります。 例えば、STYLE 属性値 "s" は、ビットマップコントロールタイプのダイアログエレメントに対しては 「伸縮」 を意味しますが、線コントロールタイプに対しては 「固定」 を意味します。

ダイアログエレメントをダイナミックに定義するには、PROCESS GUI ステートメントの ADD アクションを使用します。 PROCESS GUI ステートメントの ADD アクションを使用すると、ステートメント内で属性値を指定できます。 PROCESS GUI ステートメントの ADD アクションは、グローバル属性リストからの属性の継承に影響されません。 ADD アクションが実行される前に、ステートメントで指定された属性がグローバル属性リストに転送されます。

Note: PARAMETERS 節 2 オプション付きの PROCESS GUI ステートメントの ADD アクションを使用すると、グローバル属性リストは使用されないか、または影響を受けません。 ダイアログエレメントの作成に必要なパラメータのうち、PROCESS GUI ステートメントの ADD アクションの WITH PARAMETERS セクションで指定されなかったものには、デフォルト値が使用されます。 パラメータリストで明示的に渡されるパラメータだけがダイアログエレメントの作成に使用されます。

リストボックス項目および選択ボックス項目をダイナミックに作成するには、PROCESS GUI ステートメントの ADD-ITEMS アクションを使用する方が便利です。 この方法を使用すると、複数の項目を同時に作成できます。

例：

```
/* #PB-A inherits the current settings of the global attribute list
#PB-A.STRING := 'TEST1'
PROCESS GUI ACTION ADD WITH #DLG$WINDOW PUSHBUTTON #PB-A
#PB-B.STRING := 'TEST2'
/* #PB-B has the same attributes as #PB-A except STRING. This leads to #PB-B
/* covering #PB-A.
PROCESS GUI ACTION ADD WITH #DLG$WINDOW PUSHBUTTON #PB-B
COMPUTE #PB-C.RECTANGLE-Y = #PB-B.RECTANGLE-Y + #PB-C.RECTANGLE-H + 20
/* #PB-B has the same attributes as #PB-A except RECTANGLE-Y
/* #PB-C will be located 20 pixels below #PB-B
PROCESS GUI ACTION ADD WITH #DLG$WINDOW PUSHBUTTON #PB-C
```

ダイアログエレメントをダイナミックに削除するには、PROCESS GUI ステートメントの DELETE アクションを使用します。 この方法は、設計時にダイアログエディタで作成したダイアログエレメントを削除する場合にも使用できます。 削除したダイアログエレメントのハンドルの使用は無効であるため、使用しないでください。

ダイアログエレメントをダイナミックに作成する必要のない場合もあります。場合によっては、コンテキストに応じてダイアログエレメントを VISIBLE = TRUE または VISIBLE = FALSE にするだけで十分です。 この手法の方が効果的で操作が簡単です。 また、ナビゲーションシーケンスの任意の箇所にダイアログエレメントを 「挿入」 することもできます。

例：

```
DEFINE DATA LOCAL
   ...
   1 #PB-1 HANDLE OF PUSHBUTTON
   ...
END-DEFINE
...
#PB-1.VISIBLE := FALSE
...
IF...                          /* Logical condition
```

```
   #PB-1.VISIBLE := TRUE
END-IF
```

ダイナミックに作成されたダイアログエレメントのイベントの処理方法

ダイナミックに作成されたダイアログエレメントにイベントを関連付ける場合は、ダイアログエディタは使用できません。 このようなダイアログエレメントのイベントはすべて、DEFAULT イベントで処理する必要があります。 このイベントでは、どのダイアログエレメントに対してどのイベントが発生したかを取り出す必要があります。このコードは、ダイアログエディタによって生成されるコードに似ています。 一般的な構造は下記の例のようになります。

例：

```
DECIDE ON FIRST *CONTROL
VALUE #PB-A
   DECIDE ON FIRST *EVENT
      VALUE 'CLICK'
         /* Click event-handler code
      NONE
         IGNORE
   END-DECIDE
VALUE #PB-B
   ...
VALUE #PB-C
   ...
END-DECIDE
```

ダイナミックに作成した ActiveX コントロールのイベントコードで*イベントパラメータが使用されている*場合、イベントの名前を含む OPTIONS 2 ステートメントをイベントコードの前に記述する必要があります。そうしない場合、コンパイラはパラメータ参照（例：#OCX-1.<<PARAMETER-...>>）を正常に処理できません。 ただし、スタティックに作成したコントロールのイベントに対する、ダイアログエディタによる OPTIONS ステートメントの暗黙的な生成とは対照的に、この場合は OPTIONS 3 ステートメントをコーディングしないでください。コーディングすると、ダイアログエディタは OPTIONS 3 ステートメントを DEFAULT イベントのエンドマーカーとして誤って解釈するため、ダイアログをロードしようとするとスキャンエラーが発生します。

例：

```
DECIDE ON FIRST *CONTROL
VALUE #OCX-1  /* MS Calendar control
  DECIDE ON FIRST *EVENT
      VALUE '-602' /* DispID for KeyDown event
        OPTIONS 2 KeyDown
            /* KeyDown event-handler code containing parameter
            /* access (e.g. #OCX-1.<<parameter-shift>>)
        NONE
         IGNORE
 END-DECIDE
...
END-DECIDE
```

75 ダイアログエレメントを有効および無効にする方法

エンドユーザーとの対話中に、特定のダイアログエレメントを使用できないようにする必要があることが明確になることがあります。 例えば、個人データを要求するダイアログに、既婚／未婚情報に関するラジオボタンコントロールのグループと結婚日のための入力フィールドコントロールが含まれている場合、既婚／未婚情報が "既婚" でない場合は入力フィールドコントロールを無効にする必要があります。

これを実現するには、以下の 2 つの方法があります。

- Natural コードを使用してダイアログエレメントをダイナミックに有効／無効にする。
- ダイアログエディタを使用して無効にする（初回のみ）。

よく使用される方法は 1 つ目の方法です。

Natural コードの例は、以下のとおりです。

```
/*First alternative
...
IF #RB-1.ENABLED = TRUE     /* Logical condition
   #IF-1.ENABLED := TRUE    /* Set ENABLED to TRUE
END-IF
...
/*Second alternative
#PB-1.ENABLED := #RB-1.ENABLED
```

ダイアログエディタを使用する場合は、ダイアログエレメントの属性ウィンドウで［有効化］エントリをマークすることによって ENABLED 属性を TRUE に設定します。

入力フィールドコントロール、選択ボックスコントロール、および編集エリアコントロールを編集できないようにする場合、これらのダイアログエレメントを完全に無効化する必要があるとは

限りません。 ダイアログエレメントを MODIFIABLE = FALSE にするだけで十分な場合もあります。

76 コンテキストメニューの定義および使用

このchapterでは、次のトピックについて説明します。

はじめに

Natural v4.1.1 から、Natural アプリケーション内で使用するコンテキストメニューを作成できるようになりました。 メニュー内容があらかじめ判明しており、ダイアログエディタで作成できる場合は、コンテキストメニューを完全にスタティックにすることができます。一方、メニュー内容や状態がランタイム時に決まるため設計時に判明していない場合は、コンテキストメニューの全体または一部をダイナミックにすることができます。

構築

コンテキストメニューはサブメニューの概念に非常に似ています。 そこで、コンテキストメニューの編集には、ダイアログのメニューバーの編集に使用するのと同じメニューエディタを使用します。 メニューバーのサブメニューとまったく同じ方法で、メニュー項目をコンテキストメニューに追加し、イベントと関連付けることができます。 ［**OLE**］コンボボックス（最上位メニューバーのサブメニューにのみ使用可能）が常に使用不可であること以外は、メニューバーエディタと機能的な相違はありません。 ただし、コンテキストメニュー項目が存在する限り、そのメニュー項目に定義されたアクセラレータはグローバルに使用できることに注意してください。 さらに、コンテキストメニューが表示されていない場合、または異なるコンテキストメニューを使用しているかまったく使用していないコントロールにフォーカスがある場合でも、アクセラレータは定義されたメニュー項目を起動します。

コンテキストメニューエディタは、［ダイアログ］メニューの新規メニュー項目である［コンテキストメニュー］、このメニュー項目に関連付けられたアクセラレータ（デフォルトでは Ctrl + Alt + X）、またはツールバーのアイコンから起動します。 ただし、コンテキストメニューエディタは、同時に1つのコンテキストメニューエディタしか編集できないため、直接は起動されません。 コンテキストメニューの全体的な操作を行い、選択したコンテキストメニューに対するメニューエディタを起動できる［ダイアログコンテキストメニュー］ウィンドウが表示されます。

サブメニューとコンテキストメニューを区別するために、コンテキストメニューは内部的に新規タイプ `CONTEXT MENU` を持ちます。 このタイプを使わないと、どちらの場合も生成コードが同じになります。以下のサンプルコードは、2つのメニュー項目を持つ簡単なコンテキストメニューを構築するために使用するステートメントを示しています。

```
/* CREATE CONTEXT MENU ITSELF:
PROCESS GUI ACTION ADD WITH PARAMETERS
  HANDLE-VARIABLE = #CONTEXT-MENU-1
  TYPE = CONTEXTMENU
  PARENT = #DLG$WINDOW
END-PARAMETERS GIVING *ERROR
/* ADD FIRST MENU ITEM:
PROCESS GUI ACTION ADD WITH PARAMETERS
  HANDLE-VARIABLE = #MITEM-1
  TYPE = MENUITEM
  DIL-TEXT = 'Invokes the first item'
  PARENT = #CONTEXT-MENU-1
  STRING = 'Item 1'
END-PARAMETERS GIVING *ERROR
/* ADD SECOND MENU ITEM:
PROCESS GUI ACTION ADD WITH PARAMETERS
  HANDLE-VARIABLE = #MITEM-2
  TYPE = MENUITEM
  DIL-TEXT = 'Invokes the second item'
  PARENT = #CONTEXT-MENU-1
  STRING = 'Item 2'
END-PARAMETERS GIVING *ERROR
```

コンテキストメニューまたはコンテキストメニュー項目をユーザー記述コード内でダイナミックに作成すると、コンテキストメニューまたはコンテキストメニュー項目がダイアログエディタで表示されないことに注意してください。 例えば、ダイナミックに作成したメニューは、コンテキストメニューリストボックスに表示されません。また、ダイナミックに作成されたメニュー項目はコンテキストメニューエディタに表示されません。

アソシエーション

コンテキストメニューの作成後に、コンテキストメニューをNaturalオブジェクトと関連付ける必要があります。 コンテキストメニューは、キーボードフォーカスを受け取ることのできるほとんどすべてのコントロールタイプと、ダイアログウィンドウ自体をサポートしています。 コントロールタイプには、ActiveXコントロール、ビットマップ、キャンバス、編集エリアフィールド、入力フィールド、リストボックス、プッシュボタン、ラジオボタン、スクロールバー、選択ボックス、テーブルコントロール、トグルボタン、標準ウィンドウ、MDI子ウィンドウ、MDIフレームウィンドウなどがあります。

コンテキストメニューをサポートするすべてのオブジェクトタイプについて、ダイアログエディタ内の対応する属性ダイアログには、ダイアログエディタによって作成されるすべてのコンテキストメニューと空のエントリをリストする読み取り専用のコンボボックスが含まれます。 空のエントリを選択すると（デフォルト）、このオブジェクトにコンテキストメニューを使用しないことを意味します。

内部的には、アソシエーションは新規属性 CONTEXT-MENU によって行われます。また、この属性にコンテキストメニューのハンドルを設定する必要があります。 この属性はオブジェクト作成時または作成後に割り当てることができます。指定しない場合、コンテキストメニューがないことを意味する NULL-HANDLE にデフォルト設定されます。 ダイアログエディタによって作成されるコンテキストメニューに対しては、以下の例のように、コントロールの作成時にコンテキストメニューが指定されます。

```
PROCESS GUI ACTION ADD WITH
PARAMETERS
  HANDLE-VARIABLE = #LB-1
  TYPE = LISTBOX
  RECTANGLE-X = 585
  RECTANGLE-Y = 293
  RECTANGLE-W = 142
  RECTANGLE-H = 209
  MULTI-SELECTION = TRUE
  SORTED = FALSE
  PARENT = #DLG$WINDOW
  CONTEXT-MENU = #CONTEXT-MENU-1
  SUPPRESS-FILL-EVENT = SUPPRESSED
END-PARAMETERS GIVING *ERROR
```

ユーザー記述イベントコード内で作成されるコントロールに対しても、同じ構文を使うことができます。 その他の場合、つまりダイアログエディタによってコントロールは作成されたがコンテキストメニューは作成されなかった場合、コントロールの作成後にダイアログの AFTER-OPEN イベントなどで、コンテキストメニュー属性をコントロールに割り当てる必要があります。

```
/* CONTEXT MENU SPECIFIED AFTER CREATION:

#LB-2.CONTEXT-MENU := #CONTEXT-MENU-2
```

コンテキストメニューを使用するオブジェクトが破棄されてもコンテキストメニューは破棄されないことに注意してください。 コンテキストメニューは、親オブジェクト（通常は、コンテキストメニューが定義されたダイアログ）が破棄されたときに破棄されます。 同様に、すでにメニューハンドルが割り当てられている CONTEXT MENU 属性に新しいメニューハンドルを割り当てても、前のコンテキストメニューは破棄されません。したがって、上記の例では、以下のステートメントのどちらを実行しても、CONTEXT-MENU-1 は破棄されません。

```
PROCESS GUI ACTION DELETE WITH #LB-1              /* #CONTEXT-MENU-1 LIVES ON

#LB-1.CONTEXT-MENU := #CONTEXT-MENU-2             /* SAME HERE
```

起動

Natural のコンテキストメニュー起動処理は、次の手順で実行されます。

1. マウスを使用してコンテキストメニューにアクセスした場合（つまり、マウスの第 2 ボタンをクリックした場合）、ターゲットコントロールはマウスカーソルのすぐ下にあるコントロールであるとみなされます。 そうでない場合、すなわち、キーボードによってコンテキストメニューにアクセスした場合（つまり、コンテキストメニューキーがあればそのキー、またはキーの組み合わせの Shift+F10 を押した場合）、ターゲットコントロールはその時点でキーボードのフォーカスが当たっているコントロールだとみなされます。
2. コントロールのクリック位置は、ターゲットコントロールのクライアントエリアに対する相対位置で設定されます。 キーボードからコンテキストメニューにアクセスした場合、クリック位置は (0, 0) に設定されます。
3. イベントが抑制されていない場合、SUPPRESS-CONTEXT-MENU-EVENT 属性に基づいて、ターゲットコントロールに対する CONTEXT-MENU イベントが発生します。
4. ターゲットコントロールの CONTEXT-MENU 属性がクエリされます。 ターゲットコントロールの値とタイプによって、以下のアクションが実行されます。
   - 属性が NULL-HANDLE に設定されていて、ターゲットコントロールがダイアログの場合、何もコンテキストメニューを表示せずにコンテキストメニュー起動処理は終了します。
   - 属性が NULL-HANDLE に設定されていて、ターゲットコントロールがダイアログエレメントの場合、そのターゲットコントロールはダイアログエレメントの PARENT であるとみなされ、コンテキストメニュー起動処理は上記の手順 2 から繰り返されます。
   - 属性がコンテキストメニューハンドルに設定されている場合、このコンテキストメニューは表示する必要のあるコンテキストメニュー（つまり、ターゲットコンテキストメニュー）だと判断され、下記の手順 5 から処理が続行されます。
5. イベントが抑制されていない場合、ターゲットコンテキストメニューに対する BEFORE-OPEN イベントが発生します。
6. ターゲットコンテキストメニューの ENABLED 属性がクエリされます。 ENABLED 属性が FALSE に設定されている場合、コンテキストメニューは表示されません。
7. そうでない場合、抑制されていなければ、ターゲットダイアログに対する COMMAND-STATUS イベントが発生します。 ターゲットダイアログとは、ターゲットコントロールがダイアログエレメントの場合はそのコントロールが組み込まれているダイアログを指し、ターゲットコントロールがダイアログの場合はターゲットコントロール自体を指します。
8. コンテキストメニューは、上記の手順 2 で設定されるクリック位置に表示されます。

コンテキストメニュー内の実際のナビゲーションおよびメニュー項目に関連付けられているイベントの起動は、Windows および Natural によって、アプリケーションの介入を受けずに実行されます。

上記のプロセスは、ダイアログまたはコンテキストメニューを持つダイアログエレメント（存在する場合）が見つかるまで、最初のターゲットコントロールからコントロールの階層に従って続行され、その後でコンテキストメニューが処理されることに注意してください。

CONTEXT-MENU イベントの目的は、コンテキストに応じて多数の候補の中から、（ターゲットコントロールの CONTEXT-MENU 属性を変更することによって）アプリケーションが適切なコンテキストメニューを選択できるようにすることです。 複数のコンテキストメニューの使用例については、「*リストビューコントロールの操作*」を参照してください。

同様に、コンテキストメニューの BEFORE-OPEN イベントは、その時点のプログラムの状態に応じて、アプリケーションによるコンテキストメニューの変更を可能にします。例えば、メニュー項目を追加または削除したり、特定のメニュー項目を無効化またはチェック状態にしたりできます。 以下に、BEFORE-OPEN イベントのサンプルコードを示します。

```
/* DELETE FIRST MENU ITEM:
PROCESS GUI ACTION DELETE WITH #MITEM-1
/* CHECK SECOND MENU ITEM:
#MITEM-2.CHECKED := CHECKED
/* DISABLE THIRD MENU ITEM:
#MITEM-3.ENABLED := FALSE
/* INSERT NEW MENU ITEM BEFORE #MITEM-3:
PROCESS GUI ACTION ADD WITH PARAMETERS
  HANDLE-VARIABLE = #MITEM-4
  TYPE = MENUITEM
  DIL-TEXT = 'Invokes the first item'
  PARENT = #CONTEXT-MENU-1
  STRING = 'Item 3'
  SUCCESSOR = #MITEM-3
END-PARAMETERS GIVING *ERROR
```

ダイアログエディタ以外で作成されたコンテキストメニューについては、ダイアログの DEFAULT イベントで BEFORE-OPEN イベントの処理を行う必要があります。 また、コントロールまたはダイアログを無効にすると、コンテキストメニューは表示されず、BEFORE-OPEN イベントも起動されないことにも注意してください。 コンテキストメニュー自体を無効にした場合も同様です。 次に例を示します。

```
#CONTEXT-MENU-1.ENABLED := FALSE        /* DISABLE CONTEXT MENU DISPLAY
```

この方法で BEFORE-OPEN イベント中にコンテキストメニューを無効にできます。つまり、コントロール内のマウスカーソルの位置に応じて、コンテキストメニューを選択的に無効にできることに注意してください。 例えば、選択されたリストボックス項目上にマウスカーソルがある場合にのみコンテキストメニューを表示する場合などが該当します。 このケースに該当するかどうかは、以下の 2 つの PROCESS GUI ACTION 呼び出しを使用して判断できます。

- INQ-CLICKPOSITION。このアクションは、コントロール内でマウスの右ボタンがクリックされたときの(X,Y)位置を返すために、ビットマップとキャンバス以外のコントロールにも拡張されています。これらのパラメータは現在はオプションパラメータです。また、新しいオプションパラメータが追加されています。このパラメータは、コンテキストメニューがマウスによってアクセスされると TRUE に、キーボードによってアクセスされると FALSE に設定されます。キーボードの場合、クリック位置は(0,0)に設定されます。 これらの情報はすべて、BEFORE-OPEN イベントを送る直前に更新されます。
- INQ-ITEM-BY-POSITION。 このアクションを使用すると、リストボックスに適用される INQ-CLICKPOSITION によって返される相対座標を、その座標が指す項目に変換できます。

この2つの新規アクションを使用する例として、マウスの右ボタンが押されたときに、選択されたリストボックス項目上にカーソルがあるかどうかを確認して、コンテキストメニューを表示するかどうかを判断する場合を考えてみます。これは、関連付けられているコンテキストメニューの BEFORE-OPEN イベントに以下のコードを記述することによって実現できます。

```
PROCESS GUI ACTION INQ-CLICKPOSITION WITH
   #LB-1 #X-OFFSET #Y-OFFSET
PROCESS GUI ACTION INQ-ITEM-BY-POSITION WITH
   #LB-1 #X-OFFSET #Y-OFFSET #LBITEM
#MENU = *CONTROL
IF #LBITEM = NULL-HANDLE                    /* NO ITEM UNDER (MOUSE) CURSOR */
  #MENU.ENABLED := FALSE
ELSE
  IF #LBITEM.SELECTED = FALSE               /* ITEM UNDER CURSOR DESELECTED */
     #MENU.ENABLED := FALSE
  ELSE                                      /* ITEM UNDER CURSOR IS SELECTED */
     #MENU.ENABLED := TRUE
  END-IF
END-IF
```

マウスカーソルの下にあるメニュー項目がまだ選択されていなくても、既存の選択をクリアして、この項目を自動的に選択したい場合があります。 リストボックスの場合、直接またはダイアログエディタの［リストボックス属性］ウィンドウ内の新しい［自動選択］チェックボックス経由で、新規属性AUTOSELECTを使用することによって、自動選択を実現できます。 この属性に TRUE を設定した場合、選択されていないリストボックス項目上でコンテキストメニューが起動されると、Natural は BEFORE-OPEN イベントを送る前に自動的に選択状態を更新します。

テーブルコントロールの場合、選択の変更は BEFORE-OPEN イベントの中でアプリケーション自体が行う必要があります。 これを実現するために、テーブルコントロールに対して、以下の新規 PROCESS GUI ACTION が導入されています。

- TABLE-INQUIRE-CELL。 これは、テーブル内の相対位置 (X, Y) に対するセルの行と列の番号（1 から始まる）を返します。 通常、この位置は前の PROCESS GUI ACTION INQ-CLICKPOSITION 呼び出しによって返される位置になります。

COMMAND-STATUS イベントは、アプリケーションでコマンド（つまり、メニュー項目、ツールバー項目、およびシグナル）の無効化やチェックを実行するためのもう 1 つの場所です。 すでにこのイベントを使用している場合、これらのアクションを BEFORE-OPEN イベントで実行する必要はありません。

手動による起動

前述のコンテキストメニューの自動起動処理の他に、SHOW-CONTEXT-MENU アクションを使用して、特定の位置で特定のコンテキストメニューを手動で起動することもできます。

これは基本的には、自動起動処理が常に有効とは限らない ActiveX コントロールを使用するための機能ですが、ActiveX に使用が限定されているわけではありません。 ActiveX コントロールの中には、その内部実装によって、Natural でコンテキストメニューの表示を起動するために使用されているメッセージが送出されないものがあります。 この場合、マウスの第 2 ボタンを押すと ActiveX コントロールでイベントが発生するのであれば、この操作によるイベントのイベントハンドラ内でコンテキストメニューを手動で表示できます。

例えば、コンテキストメニュー #CTXMENU-1 を Microsoft Rich Textbox ActiveX Control の #OCX-1 で表示する場合、コントロールの MouseDown イベントハンドラに以下のコードを記述します。

```
IF #OCX-1.<<PARAMETER-Button>> = 2
  #X := #OCX-1.<<PARAMETER-x>> + 2
  #Y := #OCX-1.<<PARAMETER-y>> + 2
  PROCESS GUI ACTION SHOW-CONTEXT-MENU WITH
```

```
    #CTXMENU-1 #OCX-1 #X #Y GIVING *ERROR
END-IF
```

ローカルデータの定義は以下のように想定されています。

```
01 #X (I4)
01 #Y (I4)
```

上記のコードでは、マウスの第2ボタンが押されたかどうかをまず確認してから、コントロールから渡された位置に基づいてコンテキストメニューを手動で起動していることに注意してください。 ただし、コントロールによって渡される位置は ActiveX コントロールとの相対位置（幅 2 ピクセルのくぼんだ境界線を含む）であるのに対し、コンテキストメニューの表示には ActiveX コントロールのコンテナウィンドウ（境界線なし）との相対位置が使用されるため、最初に位置を少し修正しています。

ActiveX コントロールによっては、twip（twentieths of a point）など、ピクセル以外の座標単位で位置を返すものがあることに注意してください。 以下の例は、座標 (#X, #Y) を twip からピクセルに変換する方法を示しています。

```
#CONTROL := *CONTROL
/* Convert x-coordinate
MULTIPLY #X BY #CONTROL.DPI
DIVIDE #X BY 1440
/* Convert y-coordinate
MULTIPLY #Y BY #CONTROL.DPI
DIVIDE #Y BY 1440
```

#CONTROL は HANDLE OF GUI として定義され、#X および #Y はフォーマット I4 を想定しています。

1440 という値は、論理インチごとの twip 数です。一方、ダイアログエレメントに適用されている DPI 属性は論理インチごとのピクセル数を返します。

コンテキストメニューの共有

1つのコンテキストメニューを複数のオブジェクト（コントロールまたはダイアログ）に関連付けることができます。 次に例を示します。

```
#LB-1.CONTEXT-MENU := #CTXMENU-1
#LB-2.CONTEXT-MENU := #CTXMENU-1
```

このような場合、コンテキストメニューがどのコントロールに対して起動されたかを識別する必要があります。 *CONTROL はコンテキストメニュー自体のハンドルを含むため、BEFORE-OPEN イベントでは使用できません。Naturalはコンテキストメニューが起動されているコントロールにフォーカスを自動的に位置付けるので、どのコントロールにフォーカスがあるのかを問い合わせる必要があります。 以下の例は、この手法の使用方法を説明した BEFORE-OPEN イベントコードです。

```
PROCESS GUI ACTION GET-FOCUS WITH #CONTROL
DECIDE ON FIRST VALUE OF #CONTROL
  VALUE #LB-1
     #MITEM-17.ENABLED := FALSE
  VALUE #LB-2
     #MITEM-17.CHECKED := CHECKED
  NONE
     IGNORE
END-DECIDE
```

ただし、あらゆる場合に使用できるより優れた方法は、コンテキストメニューの CONTROL 属性をクエリすることです。

```
#CONTROL := *CONTROL
DECIDE ON FIRST VALUE OF #CONTROL.CONTROL
  ...
END-DECIDE
```

77 クリップボードおよびドラッグ & ドロップの使用

このchapterでは、次のトピックについて説明します。

はじめに

クリップボードおよびドラッグ／ドロップのデータ転送には、同じメソッドのセットで両方の要件を扱うことができる、Natural 言語レベルの論理的なクリップボードを利用します。 論理クリップボードを扱うための PROCESS GUI アクションは、OPEN-CLIPBOARD、SET-CLIPBOARD-DATA、CLOSE-CLIPBOARD、GET-CLIPBOARD-DATA、および INQ-FORMAT-AVAILABLE です。 各 Natural プロセスには 1 つずつ論理クリップボードがあります。そのため、製品ドキュメントでは 「ローカル」 クリップボードと呼ばれています。

OPEN-CLIPBOARD は、論理クリップボードデータを構築するための最初の手順です。 OPEN-CLIPBOARD は、通常はデータを供給するコントロールのハンドルである、オプションパラメータ（オーナーウィンドウ）を使用します。 すでに何かが論理クリップボードにあれば、このアクションはそれを空にします。 Natural は、BEGIN-DRAG イベント（下記参照）を起動する前に暗黙的にこの処理を行うので、ドラッグ & ドロップのためにこのアクションを呼び出す必要がないことに注意してください。

SET-CLIPBOARD-DATA は、実際のデータを論理クリップボードに転送します。最初のパラメータは、文字列として指定するクリップボードフォーマットです。 このフォーマットには、標準的なテキスト転送に使われる 2 つの事前定義フォーマット（CF-TEXT および CF-FILELIST として NGULKEY1 に定義）とそれぞれに対応するファイルリスト（Windows エクスプローラやその他の多くのアプリケーションとのデータ交換に適切）があります。 また、任意の文字列（dgit で始まっていてはならない）を使用して、Natural アプリケーションだけが理解できるプライベートクリップボードフォーマットを指定できます（データを取得するためにこのフォーマット文字列を GET-CLIPBOARD-DATA に渡せるように、アプリケーションでは単にこの文字列を識別する必要があります）。 2 番目以降の引数は任意の数のデータオペランドです。 これらには、配列（添字範囲を含む）またはスカラ（ダイナミック変数およびラージ文字変数を含む）の任意の組み合わせを指定できます。 配列は個々の要素に内部的に展開され、その後、スカラが個別に処理されます。

例：

```
DEFINE DATA LOCAL
1 #ARR(A1/2,3) INIT (1,1)<'A'> (1,2)<'B'> (1,3)<'C'>
                             (2,1)<'X'> (2,2)<'Y'> (2,3)<'Z'>
```

```
END-DEFINE
PROCESS GUI ACTION SET-CLIPBOARD-DATA WITH CF-TEXT #ARR(*,*)
```

上記のコーディングと下記のコーディングは同等です。

```
PROCESS GUI ACTION SET-CLIPBOARD-DATA WITH CF-TEXT
  'A' 'B' 'C' 'X' 'Y' 'Z'
```

事前定義フォーマットの場合、オペランドは英数字（フォーマットA）である必要があります。プライベートフォーマットの場合、ほぼすべてのタイプのデータ引数を指定できます。 例外：ハンドル変数（HANDLE OF OBJECTを含む）はプロセス固有なので、サポートされません。 プライベートフォーマットのデータは 「現状のまま」（変換なし）で保存されます。

必要なフォーマットごとに SET-CLIPBOARD-DATA アクションを実行することにより、複数のデータフォーマットをクリップボードに転送できることに注意してください。 ただし、特定のフォーマットに対してSET-CLIPBOARD-DATAを呼び出すと、すでにそのフォーマットで存在するデータはすべて置き換えられます。

データはWindowsクリップボードには転送されないことにも注意してください。 これは論理クリップボードが閉じられるときに（下記参照）行われます。

CLOSE-CLIPBOARD は、論理クリップボードを閉じて Windows クリップボードにデータを転送し、他のアプリケーションで貼り付けできるようにします。 CLOSE-CLIPBOARDを呼び出した後は、SET-CLIPBOARD-DATAでデータを変更できません。 通常はドラッグしたデータを貼り付けできるようにする必要はないため、この呼び出しはドラッグ & ドロップでは必須ではないことに注意してください。 ドラッグ & ドロップは論理クリップボードで直接操作できます。

GET-CLIPBOARD-DATA は、貼り付けを実行したり、ドロップターゲットとして動作したりするアプリケーションによって、ドラッグ & ドロップ操作中の場合はドラッグドロップクリップボードから、そうでない場合は Windows クリップボードからデータを取得するために使用されます。 ドラッグドロップクリップボードは、ソース Natural プロセスに属している論理クリップボードの同義語です。 SET-CLIPBOARD-DATA では、クリップボードフォーマットを指定し、その後に（同じフォーマットタイプ制限の）任意のデータオペランドリストを続けます。プライベートフォーマットの場合、オペランドはGET-CLIPBOARD-DATAアクションで使用するのと同じフォーマットタイプである必要はありません。 例えば、クリップボードに整数を転送し、それをパック型数値（P）変数に読み込むことができます。 内部的に、MOVE 変換が行われます。 したがって、データの設定と取得に異なるフォーマットタイプを使用する場合、それらのフォーマットには MOVE での互換性が必要です。

事前定義フォーマットの場合、個々のデータ項目は CR/LF（CF-TEXT フォーマットの場合）または NULL 終端文字（CF-FILELIST フォーマットの場合）のいずれかで区切られ、通常は受け取りフィールドごとに 1 つの項目のみが読み込まれます。 例外：最後の受け取りフィールドがダイナミック文字変数の場合、デリミタを含めて、残りのすべてのデータ項目がこの変数に格納されます。 この例外により、アプリケーションでは、（例えば）単一のダイナミック文字変数を使用して、複数行のデータまたは複数のファイル／ディレクトリ名を設定および取得できます。 使用するフォーマットに関係なく、指定された受け取りオペランドが多すぎる場合、余っ

たフィールドはリセットされます（RESET ステートメントを参照）。 個々のデータフィールドは、*n*X 表記を使用するとスキップされることに注意してください。 例えば、5X は 5 個のデータ項目をスキップします（「データ項目」 が CF-TEXT フォーマットに対する単一行の場合）。

INQ-FORMAT-AVAILABLE は、指定したフォーマットでデータを使用できるかどうかをクエリするために使用します（構文の仕様を参照）。 通常、このアクションは、［貼り付け］コマンドを有効または無効にするか、あるいはドラッグ／ドロップ操作に対して 「ドロップ禁止」 カーソルを表示するかどうかを判断するために使用します。

クリップボードの指定

実際のクリップボードデータ転送については、上記の説明を参照してください。ただし、Natural では、（他の通常のコマンドと異なり）CLICK イベントが発生しない特殊タイプの［切り取り］、［コピー］、［貼り付け］、［削除］、および［元に戻す］のシグナル、メニュー項目、およびツールバー項目を定義できます。 入力フィールド、編集エリア、選択ボックス、およびテーブルコントロールに対して Natural が実行する必要がある処理は明確であるため、Natural は暗黙的にこれを実行します。 Natural では、これらのコマンドはリストボックスおよび ActiveX コントロールをサポートしています。 ただし、Natural がこれらのコマンドにどのように応答する必要があるかについてはあいまいであるため、この場合はメカニズムが異なります。 したがって、Natural では、アプリケーションからある程度の支援を受ける必要があります。 この支援は、6 個の新規イベント（CUT、COPY、PASTE、DELETE、UNDO、および CLIPBOARD-STATUS）の形を取ります。これらのイベントはすべて、新規属性 SUPPRESS-CUT-EVENT、SUPPRESS-COPY-EVENT、SUPPRESS-PASTE-EVENT、SUPPRESS-DELETE-EVENT、SUPPRESS-UNDO-EVENT、および SUPPRESS-CLIPBOARD-STATUS-EVENT によって抑制できます。 これらの 6 個のイベントはすべて、デフォルトで抑制されています。 CUT、COPY、PASTE、DELETE、および UNDO の各イベントは、対応するコマンドが起動されると発生します。 対応するイベント抑制フラグは、ユーザーインターフェイスの対応するコマンド（複数可）を有効と無効のどちらにするかを判断するために、Natural によって使用されます。

CLIPBOARD-STATUS イベントは、コンテキスト（例えば、アクティブな選択があるかどうか）に従って、アプリケーションでダイナミックにこれらのイベント抑制フラグを設定できるようにするために、アイドル処理中にフォーカスコントロールに送られます。 Natural では、クリップボードコマンドのステータスを更新するためにイベント抑制フラグをクエリする前に、このイベントが発生します。 これらの新規イベントは、（現時点では）リストボックスと ActiveX コントロールにのみ送られることに注意してください（これらのコントロールにフォーカスがある場合のみ）。 入力フィールドや選択ボックスなどは、現在でも暗黙的に処理されます。

CLIPBOARD-STATUS イベントは、更新する必要があるクリップボードコマンドがユーザーインターフェイスに少なくとも 1 つある場合にのみ、発生します。

以下の例は、リストボックスコントロールの典型的なCLIPBOARD-STATUSイベントを示しています。

```
DEFINE DATA LOCAL
1 #CONTROL HANDLE OF GUI
1 #FMT (A10) CONST<'MYDATAFMT'>
1 #AVAIL (L)
END-DEFINE
...
#CONTROL := *CONTROL
/*
/*
Cut, Copy & Delete are enabled if an item is selected,
/*or disabled otherwise
/*
IF #CONTROL.SELECTED-SUCCESSOR <> NULL-HANDLE
  #CONTROL.SUPPRESS-CUT-EVENT := NOT-SUPPRESSED
  #CONTROL.SUPPRESS-COPY-EVENT := NOT-SUPPRESSED
  #CONTROL.SUPPRESS-DELETE-EVENT := NOT-SUPPRESSEDELSE
  #CONTROL.SUPPRESS-CUT-EVENT := SUPPRESSED
  #CONTROL.SUPPRESS-COPY-EVENT := SUPPRESSED
  #CONTROL.SUPPRESS-DELETE-EVENT := SUPPRESSED
END-IF
/*
/* Paste command is enabled if data is available in a
/* recognized format, or disabled otherwise
/*
PROCESS GUI ACTION INQ-FORMAT-AVAILABLE
 WITH #FMT #AVAIL GIVING *ERROR
/*
IF #BOOL
  #CONTROL.SUPPRESS-PASTE-EVENT := NOT-SUPPRESSED
ELSE
  #CONTROL.SUPPRESS-PASTE-EVENT := SUPPRESSED
END-IF
```

ドラッグ & ドロップの指定

ドラッグ & ドロップ操作は、自動的に（リストボックスおよびビットマップコントロールの場合）または手動で、新規PERFORM-DRAG-DROPアクション経由で（ActiveXコントロールの場合は通常、コントロール固有のマウスクリックまたはドラッグ開始イベントに応答して）起動できます。自動的なドラッグ／ドロップでは、マウスカーソルはアクティブな選択（存在する場合）の上にある必要があります。 手動のドラッグ／ドロップでは、PERFORM-DRAG-DROPのパラメータに、ドラッグ／ドロップイベントを受け取るコントロール（ドラッグソース）のハンドル、およびドラッグ & ドロップをすぐに始めるのか、またはユーザーがシステム定義の最小ピクセル数だけマウスを移動した後にのみ始めるのかを示すオプションフラグを指定します。 自動およ

び手動のドラッグ／ドロップでは内部的に同じコードが使用されているので、いずれの場合も同じイベントを受け取ります。

ドラッグ／ドロップは、DRAG-MODE（ドラッグソース用）と DROP-MODE（ドロップターゲット用）の2つの新規I4属性によって制御されます。これらの属性は、DM-NONE（ドラッグ／ドロップ不可）、DM-COPY（コピー可能）、DM-MOVE（移動可能）、DM-COPYMOVE（コピーと移動可能）、DM-LINK（リンク可能）、DM-COPYLINK（コピーとリンク可能）、DM-MOVELINK（移動とリンク可能）、DM-COPYMOVELINK（コピー、移動、リンク可能）の 8 個の値のいずれかに設定できます（NGULKEY1に定義）。リンク操作は、ドロップターゲットによってソースデータのコピーが作成されるのではなく、ソースデータへのリンクが作成されることを暗黙的に意味します。ファイル操作については、通常はデスクトップショートカットが使用されます（現時点ではNaturalによって明示的にはサポートされていません）。 ソースの DRAG-MODE 属性がデフォルト値の DM-NONE 以外の値に設定されている場合にのみ、ドラッグ操作は開始されます。 また、アプリケーションは BEGIN-DRAG イベント（下記参照）に応答する必要があります。

ドロップターゲットとして動作できるコントロールタイプには、ActiveXコントロール、ビットマップコントロール、リストボックス、コントロールボックス、編集エリア、およびダイアログがあります（タブコントロールとテーブルコントロールは将来的に対応が予定されていますが、現在はサポートされていません）。 ただし、DROP-MODE 属性がデフォルトの DM-NONE 以外の値に設定されている場合、これらのウィンドウはドロップターゲットとして OLE に登録されるだけです。 ドラッグ／ドロップ操作中、OLE は、ドロップターゲットとして登録されたウィンドウが見つかるまで、カーソルの真下にあるウィンドウから順に、ウィンドウ階層を上へ自動的に検索します。このウィンドウが、OLEドロップ通知を受け取るウィンドウになります。したがって、Natural ドラッグ／ドロップイベント（下記参照）を受け取るウィンドウでもあります。

新規ドラッグ／ドロップ関連イベントには、BEGIN-DRAG、END-DRAG、DRAG-ENTER、DRAG-OVER、および DRAG-LEAVE があります。 また、既存の DRAG-DROP イベント（ビットマップコントロールに対する簡単な非 OLE ドラッグ／ドロップサポート）も使用されます。 すべてのイベントは、適切なイベント抑制属性（SUPPRESS-BEGIN-DRAG-EVENT など）によって抑制できます。これらの属性はデフォルトではすべて SUPPRESSED に設定されています。

BEGIN-DRAG イベントは（抑制されていない場合）、ドラッグ操作の開始時にドラッグソースに送られます。アプリケーションでは、このイベントを退出する前に、SET-CLIPBOARD-DATA アクションを使用してドラッグ／ドロップクリップボードにデータを転送する*必要があります*。転送しないと、ドラッグ／ドロップ操作はマウスカーソルを変更せずに暗黙的にキャンセルされます。 OPEN-CLIPBOARD アクションまたは CLOSE-CLIPBOARD アクションのどちらも呼び出す必要がないことに注意してください。

END-DRAG イベントは（抑制されていない場合）、ドラッグ／ドロップ操作の終了後に（ドラッグ操作がキャンセルされた場合でも）ドラッグソースに送られます。 このイベントの主な使用目的は、移動操作が生じた場合にソースデータを削除することです。 アプリケーションでは、既存の INQ-DRAG-DROP アクションを呼び出すことによって、移動操作が発生しているかどうかを把握できます。INQ-DRAG-DROP アクションは機能拡張され、2 つの整数型の出力オプションパラメータが新しく追加されています。 これらの新規パラメータの 1 つ目は、現在押されているマウスボタンを示します（1 = 左ボタン、2 = 右ボタン、4 = 中央のボタン、またはその組み合わせ）。2つ目の新規パラメータが、ここで必要とされているもので、ドラッグ／ドロップ操

作による最終的なドロップ効果を示します。ドロップが実行されなかった、または操作がキャンセルされた場合は DM-NONE、コピー操作が実行された場合は DM-COPY、移動操作が実行された場合は DM-MOVE、およびリンク操作が実行された場合は DM-LINK が設定されます。

DRAG-ENTER イベントは（抑制されていない場合）、ドロップターゲットによって占有されているリージョンにドラッグカーソルが（再度）入るときに、ドロップターゲットに送られます。アプリケーションは通常、INQ-FORMAT-AVAILABLE アクションを呼び出して互換データフォーマットがクリップボードで有効かどうかを確認し、その後 SUPPRESS-DRAG-DROP-EVENT 属性を適宜設定して、このイベントに応答します。 SUPPRESS-DRAG-DROP-EVENT は、DRAG-DROP イベントを発生させるかどうかを決定するだけでなく、ドロップを許可するかどうかを Natural に通知するため、重要です。 DRAG-ENTER と DRAG-OVER の各イベントを発生させた後、Natural は SUPPRESS-DRAG-DROP-EVENT 属性を確認して「ドロップ禁止」記号を表示します。 それ以外に、ドロップ効果は、ドラッグソースの DRAG-MODE 値、ドロップターゲットの DROP-MODE 値、および現在押されている修飾キー（Shift キーと Ctrl キー）の組み合わせによって決まります。

DRAG-OVER イベントは（抑制されていない場合）、ドラッグカーソルがドロップターゲットの上を移動するたびにドロップターゲットに送られます。 このイベントは、例えば、ユーザーがコントロール内の項目を横切るときにドロップの強調表示（行われている場合）を更新したり、ドロップ操作を実行できるかどうかがドロップターゲット内の位置に依存する場合に SUPPRESS-DRAG-DROP-EVENT 属性を更新したりするために使用できます。

DRAG-LEAVE イベントは（抑制されていない場合）、ドロップを発生させずに、ドロップターゲットによって占有されているリージョンをドラッグカーソルが離れるときに、ドロップターゲットに送られます。 このイベントは（使用されたとしても）主に DRAG-OVER イベントで適用されたドロップの強調表示（行われている場合）を削除するために使用されます。

DRAG-DROP イベントは（抑制されていない場合）、ユーザーがドロップを実行するときに、ドロップターゲットに送られます。 アプリケーションは、（必要であればコントロール内の現在の相対位置を使用して）貼り付け操作を効率よく実行して、これに応答する必要があります。相対位置と操作タイプは両方とも INQ-DRAG-DROP アクションを通して取得できます。 操作タイプは新規のオプションパラメータ「Drop Effect」に返されます（詳細は上記の END-DRAG イベントを参照）。

ドラッグ & ドロップの挿入マーク

リストボックスでは、新規の"挿入マーク（i）"スタイルを使用して、ドラッグカーソルがコントロール（ドロップターゲットだと想定）の上を移動するときに現在の挿入位置を示すために水平破線を使用するよう指定できます。 アプリケーションは挿入マーク位置を直接クエリできませんが、以下の例のように、INQ-DRAG-DROP アクションでコントロール内の相対位置をクエリし、その後これらの座標を INQ-ITEM-BY-POSITION アクションに渡すことにより、データをどこに挿入するかを確認できます。

```
DEFINE DATA LOCAL
1 #Y (I4)
1 #CONTROL HANDLE OF GUI
1 #ITEM    HANDLE OF GUIEND-DEFINE
...
/* DRAG-DROP event:
PROCESS GUI ACTION INQ-DRAG-DROP WITH 4X #Y GIVING *ERROR
*
IF #Y < 0
   #Y := 0
END-IF
#CONTROL := *CONTROL
PROCESS GUI ACTION INQ-ITEM-BY-POSITION WITH
 #CONTROL 0 #Y #ITEM GIVING *ERROR
```

上記のコードを実行すると、挿入位置にある項目のハンドルが変数 #ITEM に設定されます。 その後、SUCCESSOR 属性を #ITEM に設定して WITH PARAMETERS 節を指定した ADD アクションを呼び出すことによって、この位置に 1 つ以上のリストボックス項目をダイナミックに挿入できます。

ドロップ位置がリストボックス上部の境界線上にある場合に対応するために、上記の例のように負の値の y 座標を修正する必要があることに注意してください。 ここで修正しない場合、#ITEM は NULL-HANDLE に設定され、上記のように SUCCESSOR 属性として #ITEM を直接使用していると、新規リストボックス項目はリストの先頭ではなくリストの末尾に追加されます。

ドラッグドロップチェックリスト

便宜上、Naturalアプリケーションでのドラッグドロップを実装する手順の概要をここで説明します。

1. 各ドラッグソース対して DRAG-MODE を設定します。 ドラッグソースがビットマップコントロールの場合、DRAGGABLE 属性も TRUE に設定する必要があります。

2. 各ドラッグソースの BEGIN-DRAG イベントで SET-CLIPBOARD-DATA を使用して転送データを渡します。
3. 各ドロップターゲットに対して DROP-MODE を設定します。
4. DRAG-ENTER イベントでは INQ-FORMAT-AVAILABLE アクションを使用して、サポートされているクリップボードフォーマットが有効であれば SUPPRESS-DRAG-DROP-EVENT 属性を NOT-SUPPRESSED (0) に、有効でなければ SUPPRESSED (1) に設定します。 コントロールがドラッグソースとしても動作可能で、コントロール内でドラッグドロップ操作を禁止する必要がある場合、INQ-DRAG-DROP を呼び出して、ソースコントロールハンドルを取得し、それを現在のコントロール（*CONTROL）と比較します。どちらも同じであればドラッグドロップ操作を抑制します。
5. ドロップ効果がターゲットコントロール内で位置センシティブな場合、DRAG-OVER イベント内で INQ-DRAG-DROP アクションを使用して現在の位置を取得し、（INQ-ITEM-BY-POSITION アクションなどを通じて）ドラッグカーソルの下の項目を判断して SUPPRESS-DRAG-DROP-EVENT 属性を適切に設定します。 必要に応じて現在の項目を強調表示します。
6. 上記の手順 5 で現在の項目を強調表示した場合、DRAG-LEAVE イベントおよび（場合によっては）DRAG-DROP イベントで（必要に応じて）その強調表示を解除します。
7. DRAG-DROP イベントで GET-CLIPBOARD-DATA を使用して転送データを取得し、適切に処理します。
8. ドラッグソースに対する END-DRAG イベントでは、INQ-DRAG-DROP によって返されたドロップ効果が DM-MOVE に設定されていた場合、ソースデータを削除します。
9. ドラッグソースが ActiveX コントロールの場合、（例えば）現在の選択内の位置がクリックされると、"MouseDown" イベントに応答してドラッグドロップ操作を開始するために PERFORM-DRAG-DROP アクションを呼び出します。

例 - データ転送のための X-array の使用

ユーザーの操作に応答して Windows クリップボードまたはドラッグドロップクリップボードに置く必要のある、またはすでに置かれているデータの設定または取得の問題の1つが、ランタイム時に任意の大きさのデータに対処できるようにすることです。 例えば、ユーザーは、1つ、少数、さらには数百または数千のリストボックス項目を、クリップボード操作やドラッグドロップ操作を実行する前に選択できます。 固定サイズの配列では、通常はわずかな割合しか使用されないことが多い場合でも、最悪の事態に対処するために非常に大きな配列を定義する必要があります。

Natural には、この問題に対する有効な解決方法が 2 つあります。 1 つ目の方法は、単一のダイナミック文字変数を使用して、設定または取得するすべての項目を格納することです。つまり、アプリケーション側で、SET-CLIPBOARD-DATA を呼び出す前にダイナミック変数に項目（デリミタを含む）を構築し、GET-CLIPBOARD-DATA を呼び出した後にダイナミック変数から項目を抽出します。 プライベートフォーマットには区切りがないため、このアプローチはプライベートフォーマットには使用できません。

2 つ目のアプローチは、X-array を使用することです。 クリップボードデータを設定する場合、X-array は、特定の状況で必要な要素数を正確に格納するためにサイズを変更できる点を除き、固定サイズ配列と同様に動作します。 例えば、クリップボードに書き込む 17 個の項目がある場合、以下のように X-array を使用できます。

```
DEFINE DATA LOCAL
1 #X-ARR(A80/1:*)
1 #UPB (I4) INIT <17>
END-DEFINE
RESIZE ARRAY #X-ARR TO (1:#UPB)
PROCESS GUI ACTION SET-CLIPBOARD-DATA WITH CF-TEXT #X-ARR(*)
```

下記のように、ごく一部しか使用されない、巨大な固定サイズ配列を使用する必要はありません。

```
DEFINE DATA LOCAL
1 #ARR(A80/10000)
1 #UPB (I4) INIT <17>
END-DEFINE
PROCESS GUI ACTION SET-CLIPBOARD-DATA WITH CF-TEXT ARR(1:#UPB)
```

クリップボードデータを取得する場合、アプリケーションではクリップボード上の項目数をあらかじめ把握できないため、X-arrayはさらに有効です。すべての配列要素（上記の例では10,000）を渡すと、未使用の要素もすべてリセットする必要があるため、かなり処理が遅くなります。

ただし、代わりに X-Array を使用すると、Natural は自動的に配列を (1:N) にサイズ変更します。Nはクリップボードの（残りの）最小の項目数で、配列の最大上限（DEFINE-DATA で定義されているように、"*" は可能な最大値を示します）になります。 GET-CLIPBOARD-DATA とともに X-array を使用する際には次の 3 つの制限事項があることに注意してください。

- X-array は最後（または唯一）のパラメータにする必要があります。
- 1 次元の X-array だけがサポートされています。
- X-array の範囲定義には要素 1 を含んでいる必要があります。

以下は、ダイナミックX-arrayを使用してクリップボードデータを取得する方法を説明したプログラム例です。2 つ目の X-array はデータ長を保存および表示するために使用されています。

```
DEFINE DATA LOCAL
1 #FMT (A10) CONST<'MYDATAFMT'>
1 #X-ARR (A/1:*) DYNAMIC
1 #X-LEN (I4/1:*)
1 #UPB (I4)
1 #I (I4)
END-DEFINE
PROCESS GUI ACTION OPEN-CLIPBOARD GIVING *ERROR
PROCESS GUI ACTION SET-CLIPBOARD-DATA WITH #FMT
  'MIKE' 'FRED' 'JIM' 'LULU' 'FRANK' 'JANA' 'ELIZABETH'
'TONY'
  GIVING *ERROR
PROCESS GUI ACTION CLOSE-CLIPBOARD GIVING *ERROR
PROCESS GUI ACTION GET-CLIPBOARD-DATA WITH #FMT #X-ARR(*)
  GIVING *ERROR
#UPB := *UBOUND(#X-ARR)
RESIZE ARRAY #X-LEN TO (1:#UPB)
FOR #I 1 #UPB
  #X-LEN(#I) := *LENGTH(#X-ARR(#I))
END-FOR
DISPLAY #X-ARR(*) (AL=10) #X-LEN(*) / '*** END OF DATA ***'
END
```

78 システム変数

特定のダイアログエレメントでイベントが発生するように指定すると、ダイアログエディタによって、Naturalシステム変数*CONTROL、*DIALOG-ID、および*EVENTを含むコードが生成されます。

処理中は、*CONTROLにはダイアログエレメントのハンドルが、*EVENTにはイベント名が、*DIALOG-IDにはダイアログのインスタンス識別子が設定されます。

ダイアログエディタでNaturalコードを入力すると、いつでもこれらのシステム変数を参照できます。 例えば、エンドユーザーがプッシュボタンコントロールをクリックして、イベントハンドラが共有サブルーチンを呼び出す場合、サブルーチンを呼び出す論理条件としてこれらのシステム変数を使用できます。

これらのシステム変数の詳細については、『システム変数』ドキュメントを参照してください。

79 生成された変数

このchapterでは、次のトピックについて説明します。

#DLG$PARENT

例えば MDI 子ウィンドウを操作するために、この "ユーザー" タイプの変数を使用します。 ダイアログの作成時に、Naturalによって、親ダイアログのハンドルを保持するためにこの変数が生成されます。 例えば、下記の例のように、イベントハンドラコードでこの変数を使用して、MDI 子ダイアログから別の MDI 子ダイアログを開くことができます。

Note: 生成された変数との競合を避けるために、#DLG$ で始まるユーザー定義変数の名前は使用しないでください。

例：

```
OPEN DIALOG 'MDICHILD' #DLG$PARENT #CHILD-ID
```

#DLG$WINDOW

ダイアログ内の属性をダイナミックに設定するためにこの変数を使用します。 ダイアログの作成時に、Naturalによって、ダイアログウィンドウのハンドルを保持するためにこの変数が生成されます。 この変数のデフォルト名は #DLG$WINDOW ですが、ダイアログの属性ウィンドウ内の左上にある［名前］エントリを書き換えることによって変更できます。 例えば、下記の例のように、イベントハンドラコードでこの変数を使用して、一定の論理条件が満たされたときにダイアログウィンドウを最小化できます。

#DLG$WINDOW はダイアログのユーザーインターフェイスからの観点を表し、システム変数 *DIALOG-ID はランタイムからの観点を表します。 OPEN DIALOG、CLOSE DIALOG、および SEND EVENT の各ステートメントでは、*DIALOG-ID を使用する必要があります。

Note: 生成された変数との競合を避けるために、#DLG$ で始まるユーザー定義変数の名前は使用しないでください。

例：

```
...
IF ...
   #DLG$WINDOW.MINIMIZED := TRUE
END-IF
...
```

80 属性値のソースとしてのメッセージファイルおよび変数

ほとんどのダイアログエレメントはSTRING属性を持っています。属性ウィンドウの［文字列］エントリにテキスト値を入力して属性値を指定する代わりに、ランタイム時にテキストを取得する変数またはメッセージファイル番号を指定できます。 この場合、属性値は、ダイアログエレメントの作成時に、その時点の変数の値または指定されたメッセージファイルによって決まります。 BITMAP-FILE-NAME、DIL-TEXT、およびACCELERATORの各属性に対しても属性ソースを指定できます。

手順 80.1. メッセージファイル番号または変数を指定するには

1 ダイアログエレメントの属性ウィンドウを起動します。

2 ［文字列］エントリの右の［ソース］ボタンを選択します。

［属性ソース］ダイアログボックスが開きます。 デフォルトの属性ソースは "定数" です。メッセージファイルの番号または変数名を入力することもできます。

Note: 属性値のソースとして整数の変数を使用すると、ランタイム時にメッセージファイルからその番号のメッセージが表示されることに注意してください。 例えば、整数の変数の内容をフォーマットNの変数にMOVE（移動）すると、この問題を避けることができます。

81 ユーザー定義イベントのトリガ

このchapterでは、次のトピックについて説明します。

はじめに

BEFORE-OPEN などの標準イベントの他に、ダイアログに対してユーザー定義イベントを定義できます。 あるダイアログが別のダイアログでアクションを起こすことが必要なときに、ユーザー定義イベントは便利です。

ダイアログ A の SEND EVENT ステートメントでターゲットダイアログ B のユーザー定義イベント名を指定すると、そのユーザー定義イベントが発生します。 ユーザー定義イベントを発生させるターゲットダイアログ B は、すでにアクティブである必要があります。 ダイアログ B は、OPEN DIALOG ステートメントを使用してアクティブ化できます。 先に OPEN DIALOG ステートメントを発行していないと、SEND EVENT ステートメントはランタイムエラーになります。

ダイアログに対するユーザー独自のイベントは、［イベント］ダイアログイベントハンドラメニューの［新規作成］ボタンを押すか、またはダイアログのコンテキストメニューから定義できます。新しく定義するイベントの名前を入力して、対応するイベントセクションを指定します。ユーザー定義イベントを事前定義されているイベントと区別するために、"#" で始まるユーザー定義イベント名を使うことをお勧めします。

イベントハンドラの実行中に、SEND EVENT ステートメントは別のダイアログ内のユーザー定義イベントハンドラを起動します。 このユーザー定義イベントハンドラの実行が終わると、元のダイアログに制御が戻り、SEND EVENT ステートメントの次のステートメントから実行が再開されます。 これは、サブプログラムを実行する CALLNAT ステートメントにたとえることができます。

OPEN DIALOG ステートメントと同様に、ダイアログにパラメータを渡すことができます。 パラメータを選択的に渡すには（*PARAMETERS-clause*）、ダイアログの識別子（*operand2*）の他にダイアログ名を指定する必要があります。

SEND EVENT ステートメントでは、イベントを処理しようとしているダイアログでイベントを起動しないでください。 例えば、ダイアログ A がダイアログ B にイベントを送り、ダイアログ B のイベントハンドラが、まだイベント処理を終了していないダイアログ A にイベントを送るような場合が該当します。 ダイアログ A がダイアログ B を開き、BEFORE-OPEN イベントまたは AFTER-OPEN イベントにダイアログ A への SEND EVENT が含まれている場合も同様です。

ユーザー定義イベントを起動するには、以下の構文を指定します。

```
SEND EVENT operand1 TO [DIALOG-ID] operand2
                    [      {      WITH operand3... USING [DIALOG] 'dialog-name' }  ]
                           {      WITH PARAMETRS-clause                          }
```

オペランド

Operand1 は、送信されるイベントの名前です。

Operand2 はユーザー定義イベントを受け取るダイアログの識別子で、I4 のフォーマット／長さで定義する必要があります。 この識別子は、例えば #DLG$PARENT.CLIENT-DATA の値をクエリすることによって、取得できます。

ダイアログへのパラメータ渡し

ユーザーイベントを受け取るダイアログにパラメータを渡すことができます。

operand3 として、ダイアログボックスに渡されるパラメータを指定します。

PARAMETERS-clause を使用して、パラメータを選択的に渡すことができます。

PARAMETERS-clause

```
PARAMETERS [parameter-name = operand3]_ END-PARAMETERS
```

Note: ターゲットダイアログがカタログ化されている場合にのみ、*PARAMETERS-clause* を使用できます。

Dialog-name は、ユーザー定義イベントを受け取るダイアログの名前です。

operand3 だけを使用してパラメータを渡すと、ユーザー定義イベントは下記の例のようになります。

```
/* The following parameters are defined in the dialog's
/* parameter data area:
1 #DLG-PARM1 (A10)
1 #DLG-PARM2 (A10)
1 #DLG-PARM3 (A10)
1 #DLG-PARM4 (A10)
/* When sending the user-defined event, pass the operands #MYPARM1 'MYPARM2' to
```

```
the parameters #DLG-PARM1 and #DLG-PARM2:
SEND EVENT 'MYEVENT' TO #DLG$DIA-ID WITH #MYPARM1 'MYPARM2'
```

PARAMETERS-clause を使用する場合、ユーザー定義イベントは下記の例のようになります。

```

/* The following parameters are defined in the dialog's
/* parameter data area:
1 #DLG-PARM1 (A10)
1 #DLG-PARM2 (A10)
1 #DLG-PARM3 (A10)
1 #DLG-PARM4 (A10)
/* When sending the user-defined event, the operand #MYPARM2 is passed to the
/* parameter #DLG-PARM2 and the operand 'MYPARM3' is passed to the parameter
/* #DLG-PARM3:
SEND EVENT 'MYEVENT' TO #DLG$DIA-ID
  USING DIALOG 'MYDIALOG'
  WITH PARAMETERS
    #DLG-PARM3='MYPARM3'
    #DLG-PARM2=#MYPARM2
  END-PARAMETERS
```

渡すオペランドとパラメータ定義間のフォーマット／長さの矛盾を避けるには、『ステートメント』ドキュメントに記載されている、DEFINE DATA ステートメントの BY VALUE オプションの説明を参照してください。

82 イベントの抑制

イベントが発生すると、通常はイベントハンドラが起動されます。 ただし、イベントが発生したときにイベントハンドラコードの実行をダイナミックに抑制する必要がある場合もあります。例えば、CHANGE イベントハンドラ内で入力フィールドコントロールの文字列を変更する場合、変更すること自体がCHANGEイベントを発生させるので、無限ループを避けるために文字列を変更する前に CHANGE イベントを抑制する必要があります。

このイベントハンドラコードは下記の例のようになります。

```
...
IF...                                          /* Logical condition criteria
   #IF-1.SUPPRESS-CHANGE-EVENT := SUPPRESSED   /* Suppress the event
END-IF
...
```

デフォルトでは、ダイアログエディタは、イベントハンドラコードが入力されていないすべてのイベントを抑制するコードを生成します。ダイアログエディタで［イベント...］ダイアログボックスの［抑制］オプションを使用してイベントを抑制することもできます。

イベントを抑制すると、そのイベントに対する BEFORE-ANY イベントおよび AFTER-ANY イベントも抑制されます。

83 メニュー構造、ツールバー、および MDI

このchapterでは、次のトピックについて説明します。

メニュー構造の作成

メニュー構造は、以下の 3 種類のダイアログエレメントで構成されます。

- メニューバーコントロール
- メニュー項目
- サブメニューコントロール

メニュー構造は、メニューバーコントロールを1つ持ち、このコントロールはいくつかのメニュー項目から構成されています。 項目を含んだメニューバーは、ウィンドウのタイトルバーのすぐ下に表示されます。 各メニュー項目は単純な項目である場合も、サブメニューコントロールを表す場合もあります。サブメニューコントロールの場合は、グループ化された複数のメニュー項目を縦に表示できます。 したがって、サブメニューコントロールは、1 つ下のレベルのサブメニューコントロールを表す項目を含むことができます。サブメニューコントロールは、メニューバーコントロールまたは親のサブメニューコントロール内の該当する項目がクリックされると表示されます。

メニュー構造を作成するには、以下の 2 つの方法があります。

- ダイアログエディタを使用するか、
- Natural コードを使用する。

ダイアログエディタを使用する場合

1. ダイアログの属性ウィンドウで［メニューバー］エントリをチェックして、［**OK**］を選択します。 ダイアログに戻ると、ダミーのメニューバーコントロールが表示されます。
2. ダミーのメニューバーコントロールをダブルクリックするか、Natural メニューを［ダイアログ］、［メニューバー］の順に選択するか、または Ctrl キーを押したまま M キーを押します。［ダイアログメニューバー］ダイアログボックスが開きます。 このダイアログボックスは、メニューバー、選択されたサブメニュー、および選択されたメニュー項目の 3 つのグループフレームに分割されています。
3. 選択されたメニュー項目のグループフレームで、［新規作成］を使用してメニュー項目を選択した位置の後ろまたは先頭に追加します。次に、選択されたメニュー項目のグループフレームを使用して、属性値を変更したり、新規メニュー項目にイベントハンドラを追加したりします。

一般的なメニュー項目には、エンドユーザーがメニュー項目をクリックしたときにコードが実行される CLICK イベントがあります。

Note: メニュー項目の `MENU-ITEM-TYPE` を "セパレータ" にすることもできます。この場合、メニュー項目はテキスト項目ではありません。

Natural コードを使用する場合

1. PARENT 属性に "NULL-HANDLE" または "*windowhandle*" を設定したメニューバーを作成します。
2. 単純なメニュー項目を作成するには、PARENT 属性の値を "*menubarhandlename*" にする必要があります。
3. サブメニューコントロールを作成するには、サブメニューコントロールの PARENT 属性の値を "NULL-HANDLE" または "*windowhandlename*" にする必要があります。 そして、PARENT = "*menubarhandlename*" および MENU-HANDLE = "*submenuhandlename*" でメニュー項目を作成します。
4. 次に、ウィンドウの MENU-HANDLE 属性を手順 1 で設定したメニューバーのハンドルに変更して、メニューバーをダイアログウィンドウに関連付けます。
5. *「ダイアログエレメントをダイナミックに作成および削除する方法」*で説明したように、ダイナミックに作成するメニュー項目のイベント処理は DEFAULT イベントハンドラで行う必要があります。

PARENT 属性によって、メニューバーまたはサブメニューコントロールが破棄されるタイミングが決まります。 PARENT = "*windowhandlename*" の場合、メニューバー／サブメニューコントロールはウィンドウが破棄されるときに破棄されます。 これはデフォルトの設定で、ダイアログエディタでも使用されます。 PARENT = NULL-HANDLE の場合は、メニューバー／サブメニューコントロールはアプリケーションが終了するときにのみ破棄されます。

メニュー構造のハンドルをグローバルデータエリアに定義すると、定義を複数のダイアログで共有できます。

手順 83.1. 上記のメニュー構造を構築するには

1 適用可能なイベントのハンドラにメニューバーコントロール、メニュー項目、およびサブメニューコントロールのハンドルをユーザー定義変数として定義します。

2 属性（PARENT など）に値を割り当てて PROCESS GUI ステートメントの ADD アクションを実行することによって、コントロールおよび項目を作成します。

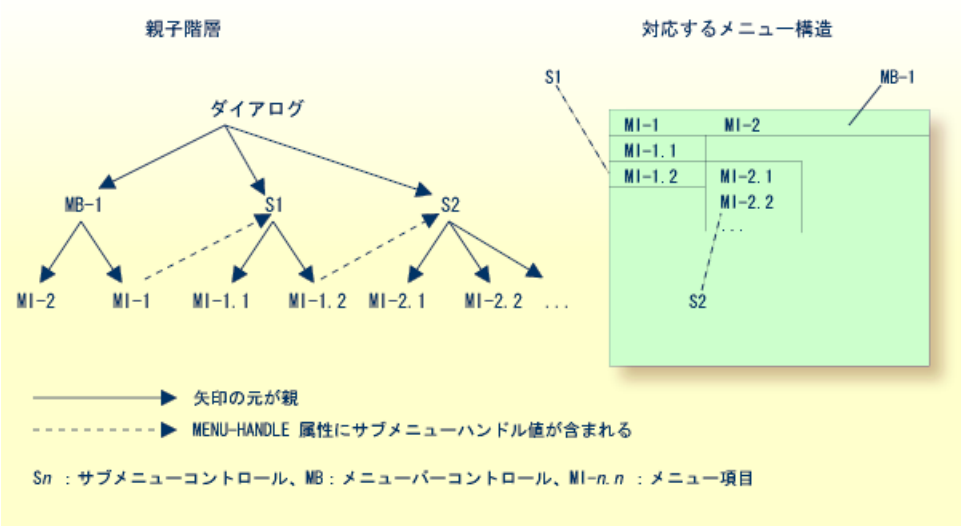
3 メニューバーコントロールおよびサブメニューコントロール内にコントロールとメニュー項目を作成します。

4 サブメニューコントロールをメニューバーコントロールに挿入し、メニューバーコントロールをダイアログウィンドウに挿入します。

エディタで構築したメニューを含むダイアログを拡張ダイアログリストモードでリストすることによって、コードでメニュー構造を構築する方法を学習できます。 メニュー項目を作成するためのコードモデルを取得するには、ダイアログエディタでメニューバーコントロールを作成し、メニューバーコントロール属性ウィンドウに進み、メニュー項目を切り取って任意のイベントハンドラセクションに貼り付けます。 メニュー項目の生成コードが表示されます。

メニュー構造の親子階層

メニュー構造内の各要素に進むにはコードを使用する必要があることがあります。 メニューの場合、親子階層は、メニュー構造のグラフィカル表示からはわからない方法で構成されます。



上図の場合、ダイアログの最初の子はメニューバーコントロールで、 その次にサブメニューコントロール S1 と S2 が続きます。 メニュー項目 MI-1 からサブメニュー S1 に進むには、MI-1の MENU-HANDLE 属性値をクエリします。 その結果得られる値が、S1 のハンドル値です。

ツールバーの作成

ツールバーおよびツールバー項目を作成するには、以下の 2 つの方法があります。

- ダイアログエディタを使用するか、
- Natural コードを使用してダイナミックに作成する。

手順 83.2. ダイアログエディタを使用するには

1 ツールバーをダブルクリックするか、または Natural メニューから［ダイアログ］、［ツールバー］の順に選択します。 ツールバー属性ウィンドウが開きます。

2 ［新規作成］ボタンをクリックしてツールバー項目を追加します。

3 新規ツールバー項目にビットマップファイル名およびその他の属性値を割り当てます。

ダイナミックに作成するためにNaturalコードを使用すると、コードでツールバーを構築する方法を学習できます。 拡張ダイアログリストモードを使用して、エディタで構築したツールバーを含むダイアログをリストしてください。

メニュー構造、ツールバー、および DIL（MDI アプリケーション）の共有

MDI（マルチドキュメントインターフェイス）アプリケーションは、すべての子ダイアログで共有されるメニュー構造、ツールバー、および DIL を提供するフレームダイアログで構成されます。 MDI フレームダイアログを使用すると、子ダイアログを並べたり重ねたりできます。

Note: ツールバーの PARENT が最上位のダイアログ（アプリケーションのメインダイアログ）である場合は、ツールバーだけを共有できます。

手順 83.3. MDI フレームダイアログを作成するには

1 ダイアログエディタを使用して、ダイアログオブジェクトの属性ウィンドウに移動します。

2 ［タイプ］エントリで［**MDI** フレームウィンドウ］を選択します。

MDI フレームダイアログには、メニューバーコントロール、サブメニューコントロール、メニュー項目、ツールバー、およびツールバー項目以外のダイアログエレメントを持たせないでください。

手順 83.4. MDI 子ダイアログを作成するには

1 ダイアログエディタを使用して、ダイアログオブジェクトの属性ウィンドウに移動します。

2 ［タイプ］エントリで［MDI 子ウィンドウ］を選択します。

MDI 子ダイアログは

- MDI フレームダイアログのエリア内だけで移動またはサイズ変更できます。
- MDI フレームダイアログのエリアの最大サイズまで最大化できます。
- 最小化できます。最小化すると、MDIフレームダイアログの下部にアイコンが表示されます。
- 独自のメニュー構造、ツールバー、および DIL を持つことができます。 これらは子ダイアログ内に表示されませんが、子ダイアログがアクティブになったときに MDI フレームダイアログ内に表示されます。別の MDI 子ダイアログがアクティブになると、メニュー構造、ツールバー、および DIL が同時に変わります。
- メニュー項目の属性 MENU-ITEM-TYPE に値［MDI 重ねて表示］または［MDI 並べて表示］を設定することによって、重ねて表示したり並べて表示したりできます。

- タイトルを MDI-WINDOWMENU タイプのサブメニューコントロールの最後に追加できます。 メニュー項目の 1 つを選択することによって、対応する MDI 子ダイアログがアクティブになります。

MDI フレームダイアログ内から MDI 子ダイアログを開く場合、例えば MDI フレームダイアログのメニュー構造にメニュー項目を作成し、そのメニュー項目に対して CLICK イベントを定義します。 次に、MDI 子ダイアログを開くための `OPEN DIALOG` コードを CLICK イベントハンドラに記述します。 エンドユーザーがメニュー項目をクリックすることによって CLICK イベントハンドラが起動されると、MDI フレームダイアログから MDI 子ダイアログが開かれます。

例：

```
OPEN DIALOG 'MDICHILD' #DLG$WINDOW #CHILD-ID
```

第 1 オペランドは、［タイプ］選択ボックスで［MDI 子ウィンドウ］を選択するとダイアログエディタによって作成されるダイアログの名前です。 第 2 オペランドは新しい MDI 子ダイアログの親で、MDI フレームダイアログである必要があります。 第 3 オペランドは、ダイアログのデータエリアに I4 として定義されている Natural 変数です。 この変数は、システムから返されるダイアログ ID を受け取ります。

Note: `#DLG$WINDOW` は生成された変数です。

MDI 子ダイアログから別の MDI 子ダイアログ（元の MDI 子ダイアログの兄弟）を開くこともできます。 これには、上記と類似した CLICK イベントハンドラを記述します。

```
OPEN DIALOG 'MDICHILD' #DLG$PARENT #CHILD-ID
```

第 1 および第 3 オペランドは前述の例と同じです。 第 2 オペランドは、両方の MDI 子ダイアログの親である必要があります。

Note: `#DLG$PARENT` は生成された変数です。

84 標準化されたプロシージャの実行

このchapterでは、次のトピックについて説明します。

はじめに

イベントドリブンのアプリケーションで頻繁に必要とされるプロシージャに対し、以下の2つを利用できます。

- PROCESS GUI ステートメントの一連のアクション
- SYSTEM ライブラリ内の、接頭語 NGU で始まる一連のサブプログラムおよびダイアログ

頻繁に必要とされるプロシージャの例としては、メッセージボックスの開始、編集エリアコントロールに入力された行の読み取り、ダイアログエレメントのダイナミック作成などがあります。

便宜上、ローカルデータエリア NGULKEY1 および NGULFCT1 は、新しいダイアログによって使用されるローカルデータエリアの一覧に自動的に含まれます。

- NGULFCT1 は、接頭語 NGU で始まるサブプログラムおよびダイアログを使用するのに必要です。
- NGULKEY1 は、任意のイベントハンドラコードで使用される予約キーワードをリストします。 これにより、数値 ID ではなく意味を持ったキーワードで特定の属性値を参照できます。また、意味を持ったダイアログエレメント名をパラメータとして使用することもできます。

PROCESS GUI ステートメントのアクション、使用可能なサブプログラムとダイアログ、および引き渡し可能なパラメータの詳細については、『*ダイアログコンポーネントリファレンス*』を参照してください。

PROCESS GUI ステートメント

PROCESS GUI ステートメントは、アクションを実行するために使用します。 ここでのアクションとは、イベントドリブンアプリケーションで頻繁に必要とされるプロシージャを指します。

action-name として、呼び出すアクションの名前を指定します。

operand1 として、アクションに渡されるパラメータを指定します。 パラメータは、指定した順序で渡されます。

アクション ADD の場合、パラメータを（位置ではなく）名前で渡すこともできます。このためには、*PARAMETERS-clause* を使用します。

```
PARAMETERS [parameter-name = operand1]_ END-PARAMETERS
```

この節は、アクション ADD にのみ使用でき、他のアクションには使用できません。

operand2 として、呼び出されたアクションの実行後にアクションからのレスポンスコードを受け取るフィールドを指定できます。

85 Natural 変数へのダイアログエレメントのリンク

データベースフィールドまたはプログラム変数をユーザーインターフェイスにマップする場合、入力フィールドコントロールおよび選択ボックスコントロールをNatural変数にリンクします。こうすることによってデータベースフィールドやプログラム変数の変更およびクエリが容易になります。

エンドユーザーが入力フィールドコントロールまたは選択ボックスコントロールにデータを入力して他のダイアログエレメントにフォーカスを移すと、LEAVE イベントが発生して内容（STRING）が変数に格納されます。 このようにして変数は更新されます。 エンドユーザーがデータを入力して CHANGE イベントが発生する場合は、変数は更新*されません*。

手順85.1. リンクされた変数がコード内で変更された後に、ダイアログエレメントの内容をリフレッシュするには

- PROCESS GUI ステートメントの REFRESH-LINKS アクションを使用します。

通常、ASSIGN ステートメントによる入力フィールドコントロールの変更およびクエリは以下のようになります。

```
...
#IF-1.STRING := '12345'
#TEXT := #IF-1.STRING
...
```

ただし、Natural変数を入力フィールドコントロールまたは選択ボックスコントロールにリンクすることもできます。 また、添字付き変数をダイアログエレメントやダイアログエレメント配列にリンクすることもできます。

Naturalコードで変数をリンクするには、属性LINKEDをTRUEに設定し、属性VARIABLEをNatural変数名に設定します。

```
...
#IF-1.LINKED := TRUE
#IF-1.VARIABLE := MYVARIABLE
...
```

ダイアログエディタを使用して **Natural** 変数名を入力するには

1. 入力フィールドコントロールをダブルクリックします。対応する属性ウィンドウが開きます。
2. ［文字列］エントリの右の［ソース］ボタンを選択します。 *handlename* のソース ダイアログボックスが開きます。
3. ［リンク変数］を選択します。
4. 変数名（上記の例の MYVARIABLE など）を入力します。

MYVARIABLE (A20/1:5) のような添字付き変数をリンクするには、以下の 2 つの方法があります。

- 添字付き変数に 1 つのダイアログエレメントをリンクします。次に、*handlename* のソースダイアログボックスの変数名フィールドに MYVARIABLE(2) のように添字を指定します。
- 添字付き変数にダイアログエレメント配列をリンクします。変数名フィールドには添字を指定しません。この場合、配列のオカレンスと変数の添字が一致する必要があります。MYVARIABLE (A20/1:5) は、最大 5 のオカレンスを持つ 1 次元配列にリンクできます。

86 ダイアログエレメントでの入力の検証

入力フィールドコントロールまたは選択ボックスコントロールがNatural変数にリンクされている場合、同じダイアログ内の別のダイアログエレメントにカーソルが移動すると、元のダイアログエレメントが自動的にチェックされます。 これにより、エンドユーザーの入力を検証できます。 エンドユーザーがメニュー項目をクリックするか、または別のアプリケーションに切り替えた場合は、入力フィールドコントロールまたは選択ボックスコントロールはチェックされません。

これらの 2 つのダイアログエレメントのいずれかで編集マスクを指定すると、フィールド内容が、編集マスクおよびリンクされた変数のNaturalデータタイプに照らし合わせてチェックされます。

編集マスクを指定しない場合は、Natural データタイプとのチェックのみが行われます。

入力フィールドコントロールまたは選択ボックスコントロールに編集マスクを指定するには、以下の 2 つの方法があります。

- Natural コードを使用する。
- ダイアログエディタを使用する。

Natural コードの例は、以下のとおりです。

```
...
/* Create an input-field control
   1 #IF-1 HANDLE OF INPUTFIELD
...
```

```
/* Assign the Edit Mask
#IF-1.EDIT-MASK := '999'
```

手順 86.1. ダイアログエディタで編集マスクを指定するには

- 入力フィールドコントロールの属性ウィンドウを開いて、［編集マスク］エントリを使用します。

フィールドチェックがエラーになると、エンドユーザーが［再試行］か［キャンセル］を選択できるメッセージボックスが表示されます。［再試行］は、入力したテキスト文字列がそのまま残され、訂正できることを意味します。［キャンセル］は、リンクされた変数の現在の内容にフィールドがリセットされることを意味します。

87 ダイアログエレメントのクライアントデータの保存および取得

このchapterでは、次のトピックについて説明します。

はじめに

このセクションでは、任意のユーザー定義情報（「クライアントデータ」）とダイアログまたはダイアログエレメントとの関連付けについて説明します。 この関連付けを実現するためのさまざまな補完的方法があります。これらの方法の詳細については、以降のセクションで説明します。 Naturalのクライアントデータ操作に関連する属性やアクションは、以下のとおりです（このドキュメントでの説明順）。

- CLIENT-DATA 属性
- CLIENT-HANDLE 属性
- CLIENT-KEY 属性
- CLIENT-VALUE 属性
- SET-CLIENT-VALUE アクション
- GET-CLIENT-VALUE アクション
- ENUM-CLIENT-KEYS アクション

整数データ

多くのダイアログエレメントのタイプでは、CLIENT-DATA属性を使用して、単一の任意の4バイト整数値をダイアログエレメントに関連付けることができます。 これは、データを特定のダイアログエレメントにリンクするのに便利です。 例えば、リストボックス項目は、データベースレコードの ISN を受け渡しできます。 CLIENT-DATA 属性値は随時変更できます。

このための Natural コードは以下のようになります。

```
DEFINE DATA
LOCAL
  1 #LBITEM-1 HANDLE OF LISTBOXITEM

  1 #ISN (I4)
   ...
END-DEFINE
...
READ...
   #LBITEM-1.CLIENT-DATA:= #ISN
```

```
END-READ
...
```

Note: ダイアログの CLIENT-DATA 属性はダイアログ ID のために予約されています。ユーザー定義のクライアントデータには使用しないでください。

ハンドルデータ

整数データと同様に、多くのダイアログエレメントについて、CLIENT-HANDLE 属性は任意の GUI オブジェクトハンドルを取ることができます。例えば、「*ダイアログバーコントロールの操作*」では、ダイアログ内で使用されるツールバーコントロールおよびダイアログバーコントロールのそれぞれに対してエントリが組み込まれたコンテキストメニューを構築する、一般的なサンプルコードが記載されています。このコードを使用すると、ユーザーはこれらのダイアログエレメントを個別に表示したり非表示にしたりできます。この例では、各メニュー項目の CLIENT-HANDLE 属性を、対応するツールバーまたはダイアログバーのハンドルに設定しています。これにより、メニュー項目がクリックされると、その項目を直接かつ簡単に取得できます。

キー付き英数字クライアントデータ

Note: 「キー付き」という用語は、特定のダイアログエレメントに対し、複数の項目情報を設定できることを意味します。各項目情報は、一意の取得キーに基づいて格納されます。

また、CLIENT-KEY 属性および CLIENT-VALUE 属性の組み合わせを使用することにより、最大 253 文字の英数字文字列としてクライアントデータを設定したり取得したりすることもできます。

手順 87.1. ダイアログエレメントに特定の文字列を設定するには

1 ダイアログエレメントの CLIENT-KEY 属性に必要な値が設定されていない場合はまず、この属性に値を割り当てます。これにより、文字列をダイアログエレメントに格納する際のキーが決定します。

2 次に、英数字の文字列をダイアログエレメントの CLIENT-VALUE 属性に割り当てます。

これにより、1 つのダイアログエレメントに多くのキー／値のペアを格納できます。

例：

```
#LB-1.CLIENT-KEY:= 'ANYKEY'
#LB-1.CLIENT-VALUE:= 'ANYSTRING'        /* The string to be stored
```

Note: 上記の例および以降のすべての例において、ハンドル変数 #LB-1 が使用されています。この変数は、通常は（規則に従って）リストボックスを参照します。ただし、CLIENT-DATA 属性には例外があり、タイマやシグナルなどのユーザーインタフェースを持たないものを含め、あらゆるタイプの GUI オブジェクトにクライアントデータを関連付けることができます。

手順 87.2. ダイアログエレメントを特定の文字列でクエリするには

1 この属性に必要な値が含まれていない場合は、最初に CLIENT-KEY 値をダイアログエレメントに割り当ててください。

2 次に、ダイアログエレメントの CLIENT-VALUE 属性をクエリして、対応する値を取得します。

CLIENT-KEY に対応する CLIENT-VALUE をクエリした結果、ダイアログエレメントにそのようなキー／値のペアがなかった場合、空の文字列（""）が返されます。

例：

```
#LB-1.CLIENT-KEY:= 'ANYKEY'
IF #LB-1.CLIENT-VALUE EQ 'ANYSTRING' THEN
...
END-IF
```

英数字以外のデータを格納および取得する場合は少し複雑となり、以下の例のように、データを元のフォーマットに戻します。

例：

```
DEFINE DATA LOCAL
01 #DATE (D)
...
END-DEFINE

#LB-1.CLIENT-KEY := 'ANYKEY'
/* Store the current date
#LB-1.CLIENT-VALUE := *DATX

/* Retrieve it as a date (D) field
```

```
STACK TOP DATA #LB-1.CLIENT-VALUE
INPUT #DATE
```

STACK ステートメントは、英数字フォーマットでクライアント値を取得し、Natural スタックに格納します。INPUT ステートメントはスタックからその値を取り出し、暗黙的に英数字フォーマットから日付フォーマットに変換して、指定された変数 #DATE に格納します。 また、クライアント値を取得して英数字変数に格納することもできます。この場合、格納後に MOVE EDITED ステートメントを使用して、明示的に日付フィールドに変換します。 ただし、日付フォーマット（DTFORM）に依存せず、中間的な英数字変数も必要としないため、前述のアプローチの方が適しています。

日付や時刻など、データタイプによっては、CLIENT-VALUE 属性で指定されているデフォルトの英数字表記が、元のデータタイプの情報すべてに対応していない場合があります。 例えば、時間（T）の値を表すデフォルトの英数字表記には時間、分、秒のみが含まれ、日付や 1／10 秒などは含まれません。同様に、日付（D）の値を表すデフォルトの英数字表記には、世紀の情報は含まれません。 したがって、上記の例で正しい世紀を使用するには、プログラムを実行する前に「スライディングウィンドウ」（YSLW）パラメータを正しく設定する必要があります。

ダイナミック文字変数を使用して CLIENT-VALUE 属性値を直接受け取る場合、受け取った値は 253 文字の長さになり、必要に応じて空白が埋め込まれます。 これは、A253 フォーマットの属性バッファが内部的に使用されているためです。これについては、後で説明します。 明示的に A253 で定義されているフィールドをダイナミック変数に割り当てるときも同様です。 どちらの場合も、ダイナミック変数の末尾に空白が格納されることを回避するには、以下の例のように、単純な MOVE や割り当ての代わりに COMPRESS ステートメントを使用する必要があります。

```
DEFINE DATA LOCAL 01 #DYN (A) DYNAMIC ... END-DEFINE
#DYN := 'ANYSTRING' /* Set the client data #LB-1.CLIENT-KEY := 'ANYKEY'
#LB-1.CLIENT-VALUE
:= #DYN /* Retrieve value as 253-character string: #DYN := #LB-1.CLIENT-VALUE
/* Retrieve value without trailing blanks: COMPRESS #LB-1.CLIENT-VALUE INTO #DYN
```

このようにすると、どちらのアプローチを使用したかに関係なく、CLIENT-VALUE 属性の値を取得および格納しても、ダイナミック英数字変数の末尾に空白は埋め込まれません。

手順 87.3. ダイアログエレメントから特定の文字列を削除するには

1 この属性に必要な値が含まれていない場合は、最初に CLIENT-KEY 値をダイアログエレメントに割り当ててください。

2 ダイアログエレメントの CLIENT-VALUE 属性を RESET するか、または明示的に空白値を割り当てて、対応する値を削除します。

例：

```
#LB-1.CLIENT-KEY:= 'ANYKEY' RESET #LB-1.CLIENT-VALUE
```

ネイティブフォーマットのキー付きクライアントデータ

CLIENT-KEY 属性および CLIENT-VALUE 属性の組み合わせを使用してクライアントデータを英数字文字列に設定する代わりに、SET-CLIENT-VALUE アクションおよび GET-CLIENT-VALUE アクションを使用すると、クライアントデータをそのままのフォーマットで変換せずに、直接格納したり取得したりできます。 ただし、この値は圧縮形式で格納される場合があります。 特に非ダイナミック英数字データの末尾の空白は、スペースを節約するため、格納されません。例えば、"FRED" という値とこれに続く 249 文字の空白文字が格納されている A253 フィールドを指定すると、A4 値の "FRED" のみがクライアントデータとして内部的に格納されます。 この最適化は、CLIENT-VALUE 属性経由で格納されたクライアントデータにも適用されます。

これらの 2 つの手法は、一緒に使用できます（1 つの手法をデータの設定に使用し、もう 1 つの手法をデータの取得に使用するなど）。 ただし、アクションの使用には、属性を使用する以上に多くの利点があります。この点については、以降のセクションで説明します。

手順87.4.アクションベースの手法を使用して、ダイアログエレメントのクライアントデータを更新するには

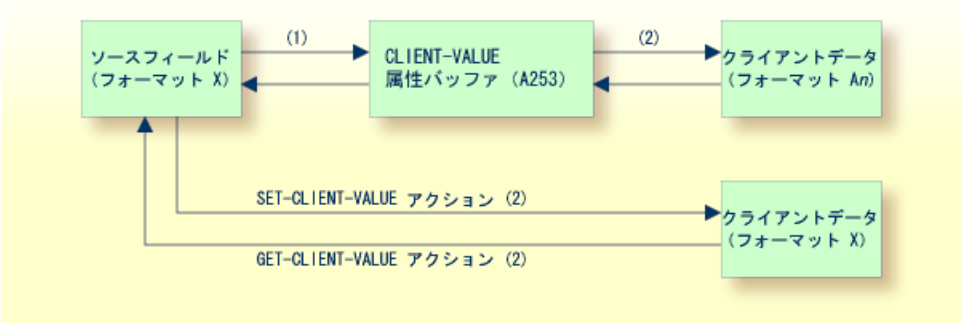
- SET-CLIENT-VALUE アクションを呼び出し、ダイアログエレメントのハンドル、格納する値の（クライアント）キー、および値そのものを渡します。 または、KEY パラメータを省略することもできます。この場合、ダイアログエレメントの CLIENT-KEY 属性の現在値が暗黙的にキーとして使用されます。

例：

```
#LB-1.CLIENT-KEY := 'ANYKEY' /* The following three statements are equivalent
ways of setting the same /* information: /* (1) attribute-based approach:
#LB-1.CLIENT-VALUE
:= 'ANYVALUE' /* (2) action-based approach, with explicitly-specified key PROCESS
GUI ACTION SET-CLIENT-VALUE WITH #LB-1 'ANYVALUE' 'ANYKEY' GIVING *ERROR /* (3)
```

```
action-based approach without key; CLIENT-KEY attribute implicitly used PROCESS
GUI ACTION SET-CLIENT-VALUE WITH #LB-1 'ANYVALUE' GIVING *ERROR
```

SET-CLIENT-VALUE アクションを使用してクライアントデータを格納する大きな利点は、CLIENT-VALUE 属性を使用する場合と異なり、英数字（A253）フォーマットに中間的に変換する必要がないという点です。 このことを以下の図で説明します。フォーマット X は任意のデータタイプを表し、フォーマット A*n* は末尾の空白が除去された英数字値を表します。



この図の矢印"（1）"および"（2）"で示されているように、CLIENT-VALUE 属性を使用したクライアントデータの格納と取得には 2 つの手順を実行する必要があります（Natural のすべての属性で同じことが当てはまります）。また、属性に定義されているフォーマットに対応する属性バッファ（この場合は A253）も使用します。 一方、SET-CLIENT-VALUE アクションおよび GET-CLIENT-VALUE アクションを使用すると、属性バッファとソースフィールドまたはターゲットフィールドとの変換を行わずに、単一の手順で、手順"（2）"と同じ処理を効率よく実行できます。 これにより、（処理がいくらか高速になる以外に）以下の利点がもたらされます。

- 属性バッファを経由する必要がないため、253 文字を超える英数字データも切り捨てを行わずに格納できます。
- ハンドル値を格納できます。ハンドルと英数字フィールドとの変換は認められていないため、ハンドル値に英数字属性バッファは使用できません。
- ソースデータがダイナミック英数字変数の場合、任意の末尾の空白を保持できます。属性バッファを使用すると、末尾の空白はバッファの充填文字と区別できなくなるため（また、バッファの充填文字とみなされるため）、格納するときに変数から除去されます。
- 英数字フォーマットとの変換を行わずにデータを格納するため、非英数字データも情報を失うことなく格納できます。 例えば、時間値を格納しても、日付情報および 1／10 秒は失われません。また、日付値を格納しても、世紀情報は失われません。

その他に、クライアントデータの格納にアクションベースのアプローチを使用することには、以下のような利点があります。

- 空白のみの英数字値を格納できます。 これは、CLIENT-VALUE 属性経由では、削除操作を暗黙的に意味するため実行できません。

- GIVINGフィールドを指定すると、標準のエラー処理を実行しなくても（必要であれば、GIVING *ERROR を使用して実行可能）このフィールドにエラーコード（無効なコントロールハンドルが渡された場合など）が返されます。

手順87.5.アクションベースの手法を使用して、ダイアログエレメントのクライアントデータをクエリするには

- GET-CLIENT-VALUE アクションを呼び出し、ダイアログエレメントのハンドル、取得する値の（クライアント）キー、および値そのものを受け取るフィールドを渡します。 または、KEY パラメータを省略することもできます。この場合、ダイアログエレメントの CLIENT-KEY 属性の現在値が暗黙的にキーとして使用されます。

例：

```
DEFINE DATA LOCAL 01 #VALUE (A253) ... END-DEFINE PROCESS GUI ACTION GET-CLIENT-VALUE
WITH #LB-1 #VALUE 'ANYKEY' GIVING *ERROR IF #VALUE <> ' ' /* Value found ... ELSE
/* Value not found ... END-IF
```

値を受け取るために指定するフィールドのフォーマットは、格納する値のフォーマットとのMOVEでの互換性が必要であることに注意してください。

指定したダイアログエレメントに対して指定したキーが見つからない場合、値フィールドは RESET されます。 例えば、英数字フォーマットの受け取りフィールドは空白で充填され、数値フォーマットの受け取りフィールドはゼロに設定されます。

ただし、このような値はプログラムで明示的にキーに対して設定できますが、この値のみを使用して必要なクライアントデータが見つかったかどうかを判断することはできません。

手順87.6.リセットされた値が明示的に格納されている場合にクライアントデータをクエリするには

- GET-CLIENT-VALUEアクションを呼び出し、前述の標準のパラメータに追加して、検出ステータスを受け取るためのタイプ L のフィールドを渡します。

例：

```
DEFINE DATA LOCAL 01 #VALUE (A253) 01 #FOUND (L) ... END-DEFINE * PROCESS GUI
ACTION GET-CLIENT-VALUE WITH #LB-1 #VALUE 'ANYKEY' #FOUND GIVING *ERROR * IF #FOUND
... END-IF
```

GET-CLIENT-VALUE アクションを使用してクライアントデータを読み込む主な利点もやはり、CLIENT-VALUE 属性を使用する場合と異なり、属性バッファを介さない（前述の図を参照）、つまり、英数字（A253）フォーマットとの中間的な変換を行う必要がないという点です。 代わり

に、格納されているデータは値を受け取るフィールドのフォーマットに直接変換されます。 これにより、以下の利点がもたらされます。

- CLIENT-VALUE 属性バッファ（未使用）の長さで切り捨てずに、253 文字を超える英数字データを取得できます。
- CLIENT-VALUE 属性バッファの英数字フォーマットでは MOVE での互換性がないハンドル値を取得できます。
- データをダイナミック英数字変数に読み込む場合、格納する英数字データの任意の末尾の空白を保持できます。 CLIENT-VALUE 属性を使用すると、ダイナミック変数でバッファの充填文字も受け取るため、元のデータの末尾の空白と区別できません。

また、空白のみで構成される、格納する英数字値を認識できます。 CLIENT-VALUE 属性では、この値と暗黙的な 「データなし」 値とを区別できないため、この属性経由では実行できません。

手順87.7.アクションベースの手法を使用して、ダイアログエレメントのクライアントデータを削除するには

- SET-CLIENT-VALUE アクションを呼び出し、ダイアログエレメントのハンドル、削除する値の（クライアント）キー、および削除する値そのものを渡します。 または、KEY パラメータを省略することもできます。この場合、ダイアログエレメントの CLIENT-KEY 属性の現在値が暗黙的にキーとして使用されます。

例：

```
/* No value supplied => delete any existing value for specified key PROCESS GUI
ACTION SET-CLIENT-VALUE WITH #LB-1 1X 'ANYKEY' GIVING *ERROR /* Alternatively,
a mixed attribute/action approach can be used: #LB-1.CLIENT-KEY := 'ANYKEY' PROCESS
GUI ACTION SET-CLIENT-VALUE WITH #LB-1 GIVING *ERROR
```

キーの列挙

これまでのセクションでは、クライアントキーおよびクライアント値データの作成、更新、クエリ、および削除について説明してきました。 ほとんどの場合はこれで十分です。 ただし、場合によっては、ダイアログエレメントで使用されているキーを読み込むコード側でそのキーが認識できない場合や、デバッグやテストで想定されるキーの検証が必要な場合があります。 特定のダイアログエレメントで現在使用されているキーを取得する繰り返しのプロセスを、*キーの列挙*と呼びます。

手順 87.8. ダイアログエレメントのクライアントキーを列挙するには

1 ENUM-CLIENT-KEYS アクションを呼び出し、キーパラメータは省略して、クライアントキーを列挙するダイアログエレメントのハンドルを渡します。 これにより、ダイアログエレメントの列挙カーソル（つまり、位置）が内部キーリストの先頭にリセットされます。 列挙カーソルはダイアログエレメントが作成されるときに初期化されるため、ダイアログエレメントに対する最初のキーの列挙では、この手順は厳密には必要ありません。 ただし、状況に関係なく列挙できるようにするために、このようにカーソルを明示的にリセットするのはよい習慣です。

2 ENUM-CLIENT-KEYS アクションを再度呼び出し、ダイアログエレメントのハンドル、および返される最初のキー（存在する場合）を受け取るキーパラメータを渡します。

3 上記の呼び出しによって、キーフィールドが内部的に空白に RESET される場合、これはそれ以上キーが残っていないことを意味するため、プログラムは列挙プロセスを終了する必要があります。

4 そうでない場合、手順 2 に戻って次のキー（存在する場合）を取得します。

例：

```
 /* Enumerate and output all client keys in use by
control #LB-1: /* (1) Reset enumeration cursor: PROCESS GUI ACTION ENUM-CLIENT-KEYS
WITH #LB-1 GIVING *ERROR /* (2) Enumerate and delete the keys one-by-one: REPEAT
PROCESS GUI ACTION ENUM-CLIENT-KEYS WITH #LB-1 #LB-1.CLIENT-KEY GIVING *ERROR
IF #LB-1.CLIENT-KEY <> ' ' RESET #LB-1.CLIENT-VALUE /* delete the key END-IF WHILE
#LB-1.CLIENT-KEY <> ' ' END-REPEAT
```

この例は、列挙プロセス中にキーが削除されても ENUM-CLIENT-KEYS アクションは対応可能であることを示しています。 この例のように、最後に列挙された、つまり「現在の」 キーが削除されると、Naturalによって自動的に、内部の列挙カーソルがそのキーの前の列挙シーケンスに移されるか、または先行するキーがない場合はリセットされます。 どちらの場合も、ENUM-CLIENT-KEYS によって返される次のキーは、前のキーが削除されなかった場合に返されるキーと同じキーになります。

Note: キーが列挙されるシーケンスは実装に依存しているため、将来のNaturalのバージョンで同一である保証はありません。 したがって、特定の列挙シーケンスに依存するようなコードをプログラムに記述しないようにしてください。

88 キャンバスコントロールでのダイアログエレメントの作成

キャンバスコントロールを背景として使用して、キャンバスコントロール上に矩形コントロール、線コントロール、およびグラフィックテキストコントロールの各ダイアログエレメントを描画できます。 これらのダイアログエレメントは情報を「視覚化」します。 例えば、ランタイム時に数個の矩形コントロールを作成し、色を付けてサイズを変更できます。 この方法で、ユーザー独自のバーチャートを構築できます。

ダイアログにキャンバスコントロールを作成すると、キャンバスコントロール上に矩形、線、およびグラフィックテキストの各コントロールを作成できるようになります。

Note: グラフィックテキストコントロールは、このコントロールが置かれる矩形の背景を再描画しません。 矩形の背景は、グラフィックテキストコントロールの作成時に指定します。 再描画されるのは、テキスト属性で指定されたテキストだけです。

手順 88.1. キャンバスコントロール上にダイアログエレメントを作成するには

- PROCESS GUI ステートメントの ADD アクションを使用します。

ダイアログ内に一度キャンバスコントロールを作成すると、矩形、線およびグラフィックテキストの各コントロールをその上に作成できます。これらのコントロールがキャンバスの境界線を越える場合は切り取られます。

キャンバスコントロールおよびキャンバスコントロール上に置かれるダイアログエレメントの動作を制御するには、以下の属性が有効です。

- OFFSET-X および OFFSET-Y を使用して、矩形、線、およびグラフィックテキストの各コントロールがキャンバスコントロールの境界線を越えたエリアの上部の境界線に対する、キャンバスコントロールの上部の境界線の X／Y 軸のオフセットを指定します。
- RECTANGLE-X、RECTANGLE-Y、RECTANGLE-W、および RECTANGLE-H を使用して、矩形コントロールのサイズと、背景となるキャンバスコントロールに対する相対的な位置を指定します。

- P1-X、P1-Y、P2-X、およびP2-Yを使用して、背景となるキャンバスコントロールに対する線コントロールの相対的な開始位置（P1*xx*）と終了位置（P2*xx*）を指定します。

以下の例は、キャンバスコントロールの作成方法を示しています。

```
/* In the dialog's local data area, the following must be defined:
01 #CNV1 HANDLE OF CANVAS
01 #XAX HANDLE OF LINE
01 #YAX HANDLE OF LINE
01 #H1 HANDLE OF RECTANGLE
01 #H2 HANDLE OF RECTANGLE
01 #H3 HANDLE OF RECTANGLE
01 #H4 HANDLE OF RECTANGLE
01 #RESPONSE (I4)
/* In the dialog's AFTER-OPEN event handler, the following must be defined:
PROCESS GUI ACTION ADD WITH
PARAMETERS
  PARENT = #DLG$WINDOW
  TYPE = CANVAS
  HANDLE-VARIABLE = #CNV1
  RECTANGLE-X = 20
  RECTANGLE-Y = 20
  RECTANGLE-W = 200
  RECTANGLE-H = 200
  STYLE = 'F'
END-PARAMETERS
GIVING RESPONSE
PROCESS GUI ACTION ADD WITH
PARAMETERS
  PARENT = #CNV1
  TYPE = LINE
  HANDLE-VARIABLE = #YAX
  STYLE = 'S'
  P1-X = 20
  P1-Y = 20
  P2-X = 20
  P2-Y = 180
END-PARAMETERS
GIVING RESPONSE
PROCESS GUI ACTION ADD WITH
PARAMETERS
  PARENT = #CNV1
  TYPE = LINE
  HANDLE-VARIABLE = #XAX
  P1-X = 180
  P1-Y = 180
  P2-X = 20
  P2-Y = 180
END-PARAMETERS
GIVING RESPONSE
```

```
PROCESS GUI ACTION ADD WITH
PARAMETERS
  PARENT = #CNV1
  TYPE = RECTANGLE
  HANDLE-VARIABLE = #H1
  RECTANGLE-X = 20
  RECTANGLE-Y = 180
  RECTANGLE-H = 20
  RECTANGLE-W = -60
  FOREGROUND-COLOUR-NAME = BLACK
  BACKGROUND-COLOUR-NAME = RED
END-PARAMETERS
GIVING RESPONSE
PROCESS GUI ACTION ADD WITH
PARAMETERS
  PARENT = #CNV1
  TYPE = RECTANGLE
  HANDLE-VARIABLE = #H2
  RECTANGLE-X = 40
  RECTANGLE-Y = 180
  RECTANGLE-H = 20
  RECTANGLE-W = -40
  FOREGROUND-COLOUR-NAME = BLACK
  BACKGROUND-COLOUR-NAME = BLUE
END-PARAMETERS
GIVING RESPONSE
PROCESS GUI ACTION ADD WITH PARAMETERS
  PARENT = #CNV1
  TYPE = RECTANGLE
  HANDLE-VARIABLE = #H3
  RECTANGLE-X = 60
  RECTANGLE-Y = 180
  RECTANGLE-H = 20
  RECTANGLE-W = -55
  FOREGROUND-COLOUR-NAME = BLACK
  BACKGROUND-COLOUR-NAME = GREEN
END-PARAMETERS
GIVING RESPONSE
PROCESS GUI ACTION ADD WITH
PARAMETERS
  PARENT = #CNV1
  TYPE = RECTANGLE
  HANDLE-VARIABLE = #H4
  RECTANGLE-X = 80
  RECTANGLE-Y = 180
  RECTANGLE-H = 20
  RECTANGLE-W = -80
  FOREGROUND-COLOUR-NAME = BLACK
  BACKGROUND-COLOUR-NAME = MAGENTA
END-PARAMETERS
GIVING RESPONSE
```

89 ツリービューおよびリストビューコントロールでのラベル編集

このchapterでは、次のトピックについて説明します。

はじめに

このセクションでは、ツリービューコントロールおよびリストビューコントロールの項目ラベルを編集するプロセスについて説明します。ここでは「項目」という言葉を、「ツリービュー項目」および「リストビュー項目」の代わりに使用します。

ラベル編集

項目のラベル編集は、禁止されていない限り（下記参照）、以下の3つの方法のいずれかで開始できます。

1. ユーザーが選択した項目のラベルをクリックする。
2. ユーザーがF2キーを押す（フォーカスのある矩形の項目があれば編集可能）。
3. プログラムで EDIT-LABEL アクションを呼び出す。

開始方法に関係なく、Naturalによって実行される一連のアクションは同じです。

1. コントロールの MODIFIABLE 属性が確認されます。この属性が FALSE の場合（コントロールの属性ウィンドウで［変更可］オプションが選択されていない場合など）、これ以上アクションは実行されず、ラベル編集モードは開始されません。
2. コントロールのITEM属性が、ラベル編集が要求されている項目（「ターゲット」項目）のハンドルに設定されます。
3. 抑制されていない場合、そのコントロールの BEFORE-EDIT イベントが発生します。
4. ターゲット項目の MODIFIABLE 属性が確認されます。この属性が FALSE の場合、これ以上アクションは実行されず、ラベル編集モードは開始されません。
5. ラベル編集モードに入ります。Escキーを押すと、入力した変更をすべてキャンセルできます。この場合、元のラベルがリストアされ、編集モードは終了します。これ以上アクションは実行されません。一方、Enterキーを押すか、または別のウィンドウやコントロールにフォーカスを移すと、変更を確定できます。
6. ターゲット項目の STRING 属性が、新しいラベルテキストで更新されます。
7. 抑制されていない場合、そのコントロールの AFTER-EDIT イベントが発生します。
8. 項目のラベルとその項目の STRING 属性が同一でなくなった場合（つまり、AFTER-EDIT イベント中にアプリケーションによって属性が変更された場合）、項目のラベルも変更に合わせて更新されます。

BEFORE-EDIT イベントの目的には、2つの部分があります。1つは、特定のコンテキストに従って、アプリケーションでダイナミックに項目のMODIFIABLE属性を設定できるようにすることで

す（これにより、ラベル編集を有効化／無効化できます）。もう 1 つは、後で実行される AFTER-EDIT イベントでリストアする場合に備えて、元のラベルをアプリケーションで保存できるようにすることです。

AFTER-EDIT イベントには、以下の 4 つのオプションがあります。

1. 何もしない。この場合、新しい項目ラベルが受け入れられます。
2. 新しいラベルを拒否する。項目の STRING 属性の元の値（BEFORE-EDIT イベントで保存）がリストアされます。
3. 新しいラベルを拒否する。項目の STRING 属性を、新しい値とも元の値とも異なる、別の値に設定します（例えば、ユーザーが入力したラベルを黙って「修正」する場合など）。
4. 項目の編集モードに戻る。（おそらく、新しく入力されたラベルが無効であることを伝えるメッセージボックスを表示してから）強制的にユーザーにもう一度ラベルを変更させます。

上記のトピックの実践例として、以下の例を考えてみます。まず、後で必要となるローカルデータ変数をいくつか定義します。

```
01 #CONTROL HANDLE OF GUI 01 #ITEM HANDLE OF GUI
01 #LABEL (A) DYNAMIC 01 #POS (I4)
```

定義が終了したら、編集する項目の既存のラベルをダイナミック変数 #LABEL に保存するだけの簡単な BEFORE-EDIT イベントを記述します。

```
#CONTROL := *CONTROL #ITEM := #CONTROL.ITEM #LABEL
:= #ITEM.STRING
```

上記の手法のいくつかを説明するために、以下のような AFTER-EDIT ハンドラを使用します。

```
#CONTROL := *CONTROL #ITEM := #CONTROL.ITEM IF
#ITEM.STRING = ' ' #ITEM.STRING := #LABEL ELSE EXAMINE #ITEM.STRING TRANSLATE
INTO LOWER EXAMINE #ITEM.STRING FOR ' ' GIVING POSITION #POS IF #POS > 0 PROCESS
GUI ACTION CALL-DIALOG WITH #DLG$WINDOW #ITEM 'EDIT-LABEL' FALSE END-IF END-IF
```

上記のコードにより、以下のアクションが実行されます。

1. 新しい項目ラベルが空白のみの場合、BEFORE-EDIT イベントで保存した元の項目ラベルがリストアされます。
2. そうでない場合、新しい項目ラベルは小文字に変換されます（EXAMINE TRANSLATE）。
3. 新しい項目ラベルに空白が含まれている場合、そのデータを無効とみなしてこの項目に対する非同期のユーザー定義イベントを発生させ、ユーザーにデータの訂正を（後で）要求します。

非同期のイベントにより、無効な項目ラベルは一時的に受け入れられます。 ただし、ユーザー定義の EDIT-LABEL イベントが発生するとすぐに、データが無効で訂正が必要であることを伝えるメッセージボックスをユーザーに表示して、ラベル編集モードに再び入ります。 これは、ダイアログの DEFAULT イベントハンドラに記述された、以下のようなコードによって実行されます。

```
IF *EVENT = 'EDIT-LABEL' #ITEM := *CONTROL OPEN
DIALOG NGU-MESSAGEBOX USING #ITEM.PARENT WITH #BUTTON 'Invalid data - please
re-enter'
'Label Edit' '!O' PROCESS GUI ACTION EDIT-LABEL WITH #ITEM GIVING *ERROR END-IF
```

文字列に編集マスクやラベルの最大長を設定したり、さらに大文字のみを認めるよう指定したりする場合、項目の EDIT-MASK 属性、LENGTH 属性、または "大文字（U）" STYLE フラグをそれぞれ設定することにより、コードをまったく記述することなくこの処理を実現できます。 編集マスクを指定した場合、編集マスクに合致しないラベルが入力されると、Naturalによって元のラベルが自動的にリストアされ、ビープ音が鳴らされます（有効な場合）。

プログラムによる項目ラベルの変更

項目のラベルは、STRING 属性を使用して直接設定できます。 次に例を示します。

```
#ITEM.STRING := New label'
```

#ITEM は、対応するツリービュー項目またはリストビュー項目のハンドルです。

この場合、項目の編集マスク（存在する場合）のみが使用されます。 前述のラベル編集に関する他のすべてのポイントは、ここでは適用されません。 特に次の点に注意してください。

1. ラベルはコントロールおよび項目の MODIFIABLE 属性の値に関係なく変更されます。
2. BEFORE-EDIT イベントおよび AFTER-EDIT イベントは発生しません。
3. コントロールの ITEM 属性は設定されません。
4. 項目の "大文字（U）" STYLE フラグが設定されていても、テキストは自動的には大文字に変換されません。
5. 指定されたラベルは、項目のLENGTH属性による長さの制限（存在する場合）を超えることができます。

90 ActiveX コントロールの操作

ActiveX コントロールとは、Natural ダイアログに統合できるサードパーティ製のカスタムコントロールです。

このchapterでは、次のトピックについて説明します。

用語

ActiveX コントロールと Natural とでは、以下のように異なる用語を使用します。

ActiveX コントロール	Natural
プロパティ	属性
メソッド	PROCESS GUI ステートメントアクション

ActiveX コントロールの定義方法

ActiveXコントロールのハンドルは、組み込みダイアログエレメントと同様に定義しますが、二重山カッコ（<< と >>）内にコーディングする点が異なります。

例：

```
01    #OCX-1 HANDLE OF <<OCX-Table.TableCtrl.1 [Table Control]>>
```

上記の例では、"Table.TableCtrl.1" が、ActiveX コントロールがシステムレジストリに登録される際のプログラム ID（ProgID）です。 接頭語 "OCX-" は、ActiveX コントロールであることを示します。 "[Table Control]" は定義オプションで、識別名を示します。

ActiveX コントロールの作成方法

ActiveX コントロールのインスタンスを作成するには、PROCESS GUI ステートメントの ADD アクションを使用します。そのためには、TYPE属性の値は、二重山カッコで囲まれた、接頭語"OCX-" で始まる ActiveX コントロールの ProgID である必要があります。 ProgID とは、ActiveX コントロールがシステムレジストリに登録される際の名前です。 また、識別名を角カッコ（[と]）内に指定できます。また、ActiveXコントロールのプロパティだけでなく、RECTANGLE-X などの Natural 属性も設定できます。 プロパティ名の前に文字列 "PROPERTY-" を付けて、この組み合せを二重山カッコで囲む必要があります。 ActiveX コントロールのイベントを抑制することもできます。 抑制するには、イベント名の前に文字列"SUPPRESS-EVENT" を付けて、この組み合

せを二重山カッコで囲む必要があります。 SUPPRESS-EVENT プロパティの値は、Natural キーワードの "SUPPRESSED" か "NOT-SUPPRESSED" のどちらかです。

例：

```
PROCESS GUI ACTION ADD
    WITH PARAMETERS
       HANDLE-VARIABLE = #OCX-1
       TYPE = <<OCX-Table.TableCtrl.1 [Table Control]>>
       PARENT = #DLG$WINDOW
       RECTANGLE-X = 44
       RECTANGLE-Y = 31
       RECTANGLE-W = 103
       RECTANGLE-H = 46
       <<PROPERTY-HeaderColor>> = H'FF0080'
       <<PROPERTY-Rows>> = 16
       <<PROPERTY-Columns>> = 4
       <<SUPPRESS-EVENT-RowMoved>> = SUPPRESSED
       <<SUPPRESS-EVENT-ColMoved>> = SUPPRESSED
    END-PARAMETERS
```

単純なプロパティへのアクセス

単純なプロパティとは、パラメータを持たないプロパティのことです。 ActiveX コントロールの単純なプロパティは、組み込みコントロールの属性と同様に指定します。 属性名を指定するには、プロパティ名の前に接頭語 "PROPERTY-" を付けて、山カッコで囲みます。コンポーネントブラウザに ActiveX コントロールのプロパティが表示されます。 以下の例は、単純なプロパティ CurrentRow および CurrentCol を持つ ActiveX コントロール #OCX-1 を示しています。

例：

```
* Get the value of a property.
#MYROW := #OCX-1.<<PROPERTY-CurrentRow>>
```

```
* Put the value of a property.
#OCX-1.<<PROPERTY-CurrentCol>> := 17
```

ActiveX コントロールプロパティのデータタイプは、OLE オートメーションによって定義されるデータタイプです。 Natural では、これらのデータタイプは、それぞれ対応する Natural データタイプにマップされます。以下の表は、OLE オートメーションのデータタイプと Natural データタイプとのマップ関係を示しています。

OLE オートメーションデータタイプ	NATURAL データタイプ
VT_BOOL	L
VT_BSTR	A ダイナミック
VT_CY	P15.4
VT_DATE	T
VT_DECIMAL	Pn.m
VT_DISPATCH	HANDLE OF OBJECT
VT_ERROR	I4
VT_I1	I2
VT_I2	I2
VT_I4	I4
VT_INT	I4
VT_R4	F4
VT_R8	F8
VT_U1	B1
VT_U2	B2
VT_U4	B4
VT_UINT	B4
VT_UNKNOWN	HANDLE OF OBJECT
VT_VARIANT	（任意の Natural データタイプ）
OLE_COLOR (VT_UI4)	B3
VT_FONT (VT_DISPATCH IFontDisp*)	HANDLE OF FONT HANDLE OF OBJECT (IFontDisp*) A ダイナミック
VT_PICTURE (VT_DISPATCH IPictureDisp*)	HANDLE OF OBJECT (IPictureDisp*) A ダイナミック

この表の見方：ActiveX コントロール `#OCX-1` が "Size" というプロパティを持っており、このプロパティのデータタイプが VT_R8 であるとします。 この場合、Natural では式 `#OCX-1.<<PROPERTY-SIZE>>` のデータタイプは F8 になります。

Note: コンポーネントブラウザでは、対応する Natural データタイプが直接表示されます。

いくつかの特殊なデータタイプは、それぞれ以下のようにみなされます。

カラー

カラータイプのプロパティは、Natural では B3 値として表されます。 B3 値は RGB カラー値として解釈されます。 3 バイトのうちの各バイトが、それぞれ赤、緑、および青の要素を示します。 例えば、H'FF0000' は赤に、H'00FF00' は緑に、そして H'0000FF' は青に相当します。

例：

```
...
 01 #COLOR-RED (B3)
...
 #COLOR-RED := H'FF0000'
 #OCX-1.<<PROPERTY-BackColor>> := #COLOR-RED
...
```

画像

画像タイプのプロパティは、Natural では `HANDLE OF OBJECT` として表されます。 代わりに、画像タイプのプロパティに文字値を割り当てることもできます。 この場合、文字値はビットマップファイル（拡張子が *.bmp*）のファイル名を含む必要があります。

例：画像タイプのプロパティの使用方法

```
...
 01 #MYPICTURE HANDLE OF OBJECT
...
 * Assign a Bitmap file name to a Picture property.
 #OCX-1.<<PROPERTY-Picture>>:= '11100102.bmp'
 *
 * Get it back as an object handle.
 #MYPICTURE := #OCX-1.<<PROPERTY-Picture>>
 *
 * Assign the object handle to a Picture property of another control.
```

```
 #OCX-2.<<PROPERTY-Picture>>:= #MYPICTURE
...
```

フォント

フォントタイプのプロパティは、Natural では HANDLE OF OBJECT として表されます。 代わりに、フォントタイプのプロパティに HANDLE OF FONT を割り当てることもできます。 また、文字値を割り当てることもできます。 この場合、文字値は、HANDLE OF FONT の STRING 属性によって返される形式のフォント指定を含む必要があります。

例1：HANDLE OF OBJECT の使用方法

```
...
01 #MYFONT HANDLE OF OBJECT
...
* Create a Font object.
CREATE OBJECT #MYFONT OF CLASS 'StdFont'
#MYFONT.Name := 'Wingdings'
#MYFONT.Size := 20
#MYFONT.Bold := TRUE
*
* Assign the Font object as value to a Font property.
#OCX-1.<<PROPERTY-TitleFont>> := #MYFONT
...
```

例2：HANDLE OF FONT の使用方法

```
...
01 #FONT-TAHOMA-BOLD-2 HANDLE OF FONT
...
* Create a Font handle.
PROCESS GUI ACTION ADD WITH PARAMETERS
    HANDLE-VARIABLE = #FONT-TAHOMA-BOLD-2
    TYPE = FONT
    PARENT = #DLG$WINDOW
    STRING = '/Tahoma/Bold/0 x -27/ANSI VARIABLE SWISS DRAFT/W/2/3/'
END-PARAMETERS GIVING *ERROR
...
* Assign the Font handle as value to a Font property.
```

```
#OCX-1.<<PROPERTY-TitleFont>> := #FONT-TAHOMA-BOLD-2
...
```

例3：フォント指定文字列の使用方法

```
...
01 #FONT-TAHOMA-BOLD-2 HANDLE OF FONT
...
* Create a Font handle.
PROCESS GUI ACTION ADD WITH  PARAMETERS
    HANDLE-VARIABLE = #FONT-TAHOMA-BOLD-2
    TYPE = FONT
    PARENT = #DLG$WINDOW
    STRING = '/Tahoma/Bold/0 x -27/ANSI VARIABLE SWISS DRAFT/W/2/3/'
END-PARAMETERS GIVING *ERROR
...
* Assign the font specification as value to a Font property.
#OCX-1.<<PROPERTY-TitleFont>> := #FONT-TAHOMA-BOLD-2.STRING
...
```

バリアント

バリアントタイプのプロパティは任意の Natural データタイプと互換性があります。 このことは、"Value" がバリアントタイプのプロパティの場合、式 `#OCX-1.<<PROPERTY-Value>>` はコンパイラによってタイプチェックされないことを意味します。 そのため、変数 `#MYVAL` のタイプに関係なく、式 `#OCX-1.<<PROPERTY-Value >> := #MYVAL` と `#MYVAL := #OCX-1.<<PROPERTY-Value >>` を使用できます。 ただし、ランタイム時に特定のプロパティ値を受け入れるかどうか、または要求されたフォーマットで値を提供をするかどうかは、ActiveX コントロールによって判断されます。 認めない場合は通常、ActiveX コントロールによって例外が起こされます。 この例外は、Natural プログラムに Natural エラーコードとして返されます。 Natural エラーコードは、`ON ERROR` ブロックで通常の方法で処理できます。 バリアントタイプの特定のプロパティに実際に認められているデータフォーマットについては、ActiveX コントロールのドキュメントを確認してください。

`#OCX-1.<<PROPERTY-Value>>`（"Value" はバリアントタイプのプロパティ）のような式は、あらゆるステートメントでソースオペランドとして使用できます。 ただし、割り当てステートメントでだけはターゲットオペランドとして使用できます。

例：バリアントタイプのプロパティの使用方法

"Value" は、ActiveX コントロール #OCX-1 のバリアントタイプのプロパティだとします。

```
...
01 #STR1 (A100)
01 #STR2 (A100)
...
* These statements are allowed, because the Variant property is used
* as source operand (its value is read).
#STR1 := #OCX-1.<<PROPERTY-Value>>
COMPRESS #OCX-1.<<PROPERTY-Value>> 'XYZ' to #STR2
...
* This leads to an error at compiletime, because the Variant
* property is used as target operand (its value is modified) in
* a statement other than an assignment.
COMPRESS #STR1 "XYZ" to #OCX-1.<<PROPERTY-Value>>
...
* This statement is allowed, because the Variant property is used
* as target operand in an assignment.
COMPRESS #STR1 'XYZ' to #STR2
#OCX-1.<<PROPERTY-Value>> := #STR2
...
```

配列

最大 3 次元の SAFEARRAY タイプのプロパティは、Natural プログラムでは、同じ次元数、次元ごとのオカレンス数、および対応するフォーマットを持つ配列として表されます（3次元を超える SAFEARRAY タイプのプロパティは Natural プログラムでは使用できません）。配列タイプのプロパティの次元数およびオカレンス数は、コンパイル時ではなくランタイム時にのみ決められます。これは、この情報は変数で、コンパイル時には定義されていないためです。ただし、フォーマットはコンパイル時にチェックされます。

配列タイプのプロパティは、常に全体としてアクセスされます。 したがって、配列タイプのプロパティでは添字表記は不要なため、許可されていません。

例：配列タイプのプロパティの使用方法

"Values" は、ActiveX コントロール #OCX-1 のプロパティで、VT_I4 の SAFEARRAY タイプだとします。

```
...
01 #VAL-L (L/1:10)
01 #VAL-I (I4/1:10)
...
* This statement is allowed, because the format of the property
* is data transfer compatible with the format of the receiving array.
* However, if it turns out at runtime that the dimension count or
* occurrence count per dimension do not match, a runtime error will
* occur.
VAL-I(*) := #OCX-1.<<PROPERTY-Values>>
...
* This statement leads to an error at compiletime, because
* the format of the property is not data transfer compatible with
* the format of the receiving array.
VAL-L(*) := #OCX-1.<<PROPERTY-Values>>
...
```

PROCESS GUI ステートメントの使用

ActiveXコントロールのメソッドはPROCESS GUIステートメントのアクションとして呼び出されます。 ActiveX コントロールの複雑なプロパティ（つまり、パラメータを持つプロパティ）の場合も同様です。 コンポーネントブラウザに ActiveX コントロールのメソッドとプロパティが表示されます。

このセクションでは、次のトピックについて説明します。

- メソッドの実行
- プロパティ値の取得
- プロパティ値の設定
- オプションパラメータ
- エラー処理
- パラメータ付きイベントの使用

- ランタイム時のイベントの抑制

メソッドの実行

ActiveX コントロールのメソッドを実行するには、PROCESS GUI ステートメントを使用します。対応する PROCESS GUI のアクション名を、メソッド名の前に接頭語 "METHOD-" を付けて山カッコで囲んで指定します。 ActiveX コントロールハンドルおよびメソッドパラメータ（存在する場合）は、PROCESS GUI ステートメントの WITH 節を使用して渡します。 メソッドの戻り値（存在する場合）は、PROCESS GUI ステートメントの USING 節で指定した変数で受け取ります。

つまり、メソッドを実行するには、以下のようにステートメントを入力します。

```
PROCESS GUI ACTION <<METHOD-methodname>> WITHhandlename [parameter]...
[USING method-return-operand]..
```

例：

```
* Performing a method without parameters:
PROCESS GUI ACTION <<METHOD-AboutBox>> WITH #OCX-1
* Performing a method with parameters:
PROCESS GUI ACTION <<METHOD-CreateItem>> WITH #OCX-1 #ROW #COL #TEXT
* Performing a method with parameters and a return value:
PROCESS GUI ACTION <<METHOD-RemoveItem>> WITH #OCX-1 #ROW #COL USING #RETURN
```

コンポーネントブラウザに表示されるように、コンパイル時にメソッドのパラメータおよび戻り値のフォーマットと長さがメソッドの定義と比較チェックされます。

プロパティ値の取得

パラメータを持つプロパティの値を取得するには、対応する PROCESS GUI のアクション名を、プロパティ名の前に接頭語 "GET-PROPERTY-" を付けて山カッコで囲んで指定します。 ActiveX コントロールハンドルおよびプロパティパラメータ（存在する場合）が、PROCESS GUI ステートメントの WITH 節で渡されます。 プロパティの値は、PROCESS GUI ステートメントの USING 節で受け取ります。

つまり、パラメータを持つプロパティの値を取得するには、以下のようにステートメントを入力します。

```
PROCESS GUI ACTION <<GET-PROPERTY-propertyname>> WITHhandlename[parameter]
... USING get-property-operand
```

例：

```
PROCESS GUI ACTION <<GET-PROPERTY-ItemHeight>> WITH #OCX-1 #ROW #COL USING #ITEMHEIGHT
```

プロパティパラメータのフォーマットおよび長さとプロパティ値が、コンパイル時にメソッドの定義（コンポーネントブラウザに表示されます）と照合されます。

プロパティ値の設定

パラメータを持つプロパティの値を設定するには、対応する PROCESS GUI のアクション名を、プロパティ名の前に接頭語"PUT-PROPERTY-"を付けて山カッコで囲んで指定します。 ActiveX コントロールハンドルおよびプロパティパラメータ（存在する場合）が、PROCESS GUI ステートメントの WITH 節で渡されます。 プロパティの値は、PROCESS GUI ステートメントの USING 節で渡します。

つまり、パラメータを持つプロパティの値を設定するには、以下のようにステートメントを入力します。

```
PROCESS GUI ACTION <<PUT-PROPERTY-propertyname>> WITHhandlename[parameter]
... USING put-property-operand
```

例：

```
PROCESS GUI ACTION <<PUT-PROPERTY-ItemHeight>> WITH #OCX-1 #ROW #COL USING #ITEMHEIGHT
```

プロパティパラメータのフォーマットおよび長さとプロパティ値が、コンパイル時にメソッドの定義（コンポーネントブラウザに表示されます）と照合されます。

オプションパラメータ

ActiveX コントロールのメソッドは、オプションパラメータを持つことができます。 これは、パラメータを持つプロパティにも該当します。 メソッドが呼び出される場合は、オプションのパラメータを指定する必要はありません。1つのオプションパラメータを省略するには、PROCESS GUI ステートメントにプレースホルダ 1X を使用します。 *n* オプションパラメータを省略するには、プレースホルダ *n*X を使用します。

以下の例は、ActiveX コントロール #OCX-1 のメソッド SetAddress のパラメータ FirstName、MiddleInitial、LastName、Street、および City のうち、MiddleInitial、Street、および City がオプションである場合を示しています。 以下に、正しいステートメントを示します。

```
* Specifying all parameters.
PROCESS GUI ACTION <<METHOD-SetAddress>> WITH #OCX-1
FirstName MiddleInitial LastName Street City
* Omitting one optional parameter.
PROCESS GUI ACTION <<METHOD-SetAddress>> WITH #OCX-1
FirstName 1X LastName Street City
* Omitting the optional parameters at end explicitly.
PROCESS GUI ACTION <<METHOD-SetAddress>> WITH #OCX-1
FirstName MiddleInitial LastName 2X
* Omitting the optional parameters at end implicitly.
PROCESS GUI ACTION <<METHOD-SetAddress>> WITH #OCX-1
FirstName MiddleInitial LastName
```

オプションでない（必須）パラメータを省略すると構文エラーになります。

エラー処理

従来どおりに PROCESS GUI ステートメントの GIVING 節を使用してエラー条件を処理できます。ユーザー変数にエラーコードを獲得して処理するか、または通常の Natural エラー処理を起動して ON ERROR ブロックで処理できます。

例：

```
DEFINE DATA LOCAL
1 #RESULT-CODE (N7)
...
END-DEFINE
...
* Catching the error code in a user variable:
PROCESS GUI ACTION <<METHOD-RemoveItem>> WITH #OCX-1 #ROW #COL USING #RETURN GIVING
#RESULT-CODE
*
* Triggering the Natural error handling:
PROCESS GUI ACTION <<METHOD-RemoveItem>> WITH #OCX-1 #ROW #COL USING #RETURN GIVING
```

```
*ERROR-NR
...
```

ActiveX コントロールのメソッドの実行中に発生する特定のエラー条件は以下のとおりです。

- メソッドパラメータ、メソッドの戻り値またはプロパティ値を、ActiveXコントロールが期待するデータフォーマットに変換できなかった場合（通常、このフォーマットチェックはコンパイル時にすでに行われているので、ランタイム時には発生しません。ただし、バリアントとして定義されたメソッドパラメータ、メソッドの戻り値またはプロパティ値は、コンパイル時にチェックされません。配列および複数の Natural データタイプにマップ可能なデータタイプも同様です）。
- メソッドの検索および実行中に COM またはオートメーションエラーが発生した場合。
- ActiveX コントロールがメソッドの実行中に例外を発生させた場合。

このような場合、エラーメッセージには ActiveX コントロールによって提供される追加情報が設定されます。この情報をActiveXコントロールのドキュメントで調べることによって、エラーの原因を把握できます。

パラメータ付きイベントの使用

ActiveX コントロールによって送られるイベントは、パラメータを持つことができます。コントロールのイベントハンドラセクションで、これらのパラメータをクエリできます。参照によって渡されるパラメータも変更できます。コンポーネントブラウザに、ActiveX コントロールのイベント、パラメータの名前とデータタイプ、およびパラメータが値と参照のどちらで渡されるかの区分のすべてが表示されます。

ActiveX コントロールのイベントパラメータは組み込みコントロールの属性と同様に指定します。属性名を指定するには、パラメータ名の前に接頭語 "PARAMETER-" を付けて山カッコで囲みます。代わりに、位置で指定することもできます。この場合、属性名を指定するには、パラメータの番号の前に接頭語 "PARAMETER-" を付けて山カッコで囲みます。イベントの最初のパラメータの番号が 1、2 番目が 2、のように続きます。これらの属性名は、特定イベントのイベントハンドラ内だけで有効です。

以下の例は、ActiveXコントロール `#OCX-1` の特定のイベントがパラメータ `KeyCode` および `Cancel` を持つことを想定したものです。この場合、イベントハンドラは以下のステートメントを含むことができます。

```
* Querying a parameter by name:
#PRESSEDKEY := #OCX-1.<<PARAMETER-KeyCode>>
```

```
* Querying a parameter by position:
#PRESSEDKEY := #OCX-1.<<PARAMETER-1>>
```

参照によって渡されるパラメータはイベントハンドラで変更できます。 以下の例は、Cancel パラメータが参照によって渡され、変更可能であると想定したものです。 イベントハンドラは以下のステートメントを含むことができます。

```
* Modifying a parameter by name:
#OCX-1.<<PARAMETER-Cancel>>:= TRUE
* Modifying a parameter by position:
#OCX-1.<<PARAMETER-2>>:= TRUE
```

ランタイム時のイベントの抑制

ActiveXコントロールのイベントをランタイム時に抑制または抑制解除するには、コントロールの対応する抑制イベント属性を変更します。 抑制イベント属性名を指定するには、イベント名の前に接頭語 "SUPPRESS-EVENT-" を付けて山カッコで囲みます。 コンポーネントブラウザにActiveX コントロールのイベントが表示されます。

以下の例は、ActiveX コントロール #OCX-1 がイベント ColMoved を持っていると想定したものです。

```
* Suppress the event.
#OCX-1.<<SUPPRESS-EVENT-ColMoved>> := SUPPRESSED
* Unsuppress the event.
#OCX-1.<<SUPPRESS-EVENT-ColMoved>> := NOT-SUPPRESSED
```

91 ダイアログエレメントの配列の操作

ダイアログエレメントを1次元または2次元に配置すると便利なことがよくあります。例えば、複数のラジオボタンコントロールを１つの列に配置する場合、１つを作成して残りを１次元配列として指定できます。

手順 91.1. ダイアログエレメントの配列を操作するには

1 ラジオボタンコントロールの属性ウィンドウで［配列］ボタンを選択します。［配列仕様］ダイアログボックスが開きます。

2 以下を入力します。

- 次元数
- 第１次元および第２次元の境界（ある場合）
- X／Y 軸のピクセルでの間隔（配列を行または列に配置するかどうかに依存します）
- 配置（行または列）

配列はグラフィカルエンティティとして扱われます。 各ラジオボタンコントロールに共通の GROUP-ID 属性を割り当てる必要があることに注意してください。 こうすることによって配列を１つの論理エンティティとして扱うことができます。

配列内の各ダイアログエレメントに対して、それぞれ以下の属性を指定できます。

- STRING
- DIL-TEXT
- BITMAP-FILE-NAME

ダイアログエレメントの配列に対するイベントハンドラでは、システム変数 *CONTROL は配列要素の１つを指します。

属性値のソースとして変数を選択する場合、配列は少なくともダイアログエレメントの添字範囲を含む必要があります。

属性値のソースとしてメッセージファイル ID を指定する場合、ダイアログエレメントの配列シーケンスに対して連続したメッセージが取得されます。

以下の例のように、ダイアログエレメントの配列に (*) 表記または範囲を使用すると、配列内の全ダイアログエレメントに 1 つの値を割り当てることができます。

```
#PB-1.ENABLED(*) := TRUE    /*invalid
#PB-1.ENABLED(1:3) := TRUE  /*invalid
```

同じダイアログエレメントのシーケンスを作成する別の方法は、各ダイアログエレメントを複製またはコピーおよび貼り付けして、グリッドと十字型カーソルを使用して配置することです。

以下の例は、1 次元のプッシュボタン配列の 2 番目の要素に STRING 属性を設定する方法を示しています。

```
#PB-2.STRING(2) := 'HUGO'
```

92 コントロールボックスの操作

このchapterでは、次のトピックについて説明します。

はじめに

コントロールボックスは、ネスト構造コントロールサポートの効力を強化するのに使用されます。 ただし、コントロールボックスには、特筆すべき多くの独自機能があります。

コントロールボックスは、まったく自動力のないコントロールで、フォーカスを受け取ることができず､マウスまたはキーボード入力を受け取らないテキスト定数およびグループフレームと同じカテゴリに属します。 コントロールボックスは、コントロール階層を構成するために、他のコントロール（他のコントロールボックスも含む）に対する汎用目的のコンテナの役割を果たすことを意図したものです。 そのために、コントロールボックスは特筆すべき次の 3 つのスタイルをサポートします。

- 利便性のためにコントロールをグループ化しても、ユーザーにコンテナそのものは見せたくない場合がよくあります。このような場合、コントロールボックスに "透過" スタイルをマークできます。 このスタイルをマークすると、コントロールボックスは表示されずに、背景となる色およびコントロールが表示されます。
- コントロールボックスに "排他制御" スタイルをマークすることもできます。ダイアログエディタでの編集時またはランタイム時に排他的コントロールボックスが表示されると、同様に "排他制御" がマークされている他のすべての兄弟コントロールボックスが非表示になります。 これは、編集時とランタイム時とでは少し違った方法で適用されます。 ランタイム時には、排他的コントロールボックスの VISIBLE 属性を TRUE に設定すると、すべての排他的な兄弟が非表示になり、それらの VISIBLE 属性が FALSE に設定されます。 編集時には、排他的コントロールボックスまたはその子孫の 1 つを選択すると常に、排他的コントロールボックスが表示され、他の排他的な兄弟はすべて非表示になります。ただし、後者の場合、関係するコントロールの VISIBLE 属性は影響を受けません。 このことは、ダイアログが実行されるときに最初に表示される排他的コントロールボックスは、ダイアログの最後の保存時に表示されていた排他的コントロールボックスとは関係ないということを意味します。
- また、コントロールボックスは "親と同サイズ" スタイルをサポートします。 コンテナコントロールまたはダイアログ自体がサイズ変更されると、このスタイルが設定されたすべての子コントロールボックスは親のクライアントエリアを完全に埋めるようにサイズ変更されます。 このスタイルがダイアログエディタで最初に設定されるときも同様です。 ただし、このようなコントロールボックスをコンテナとは関係なくサイズ変更することもできます。

排他的コントロールボックスの目的

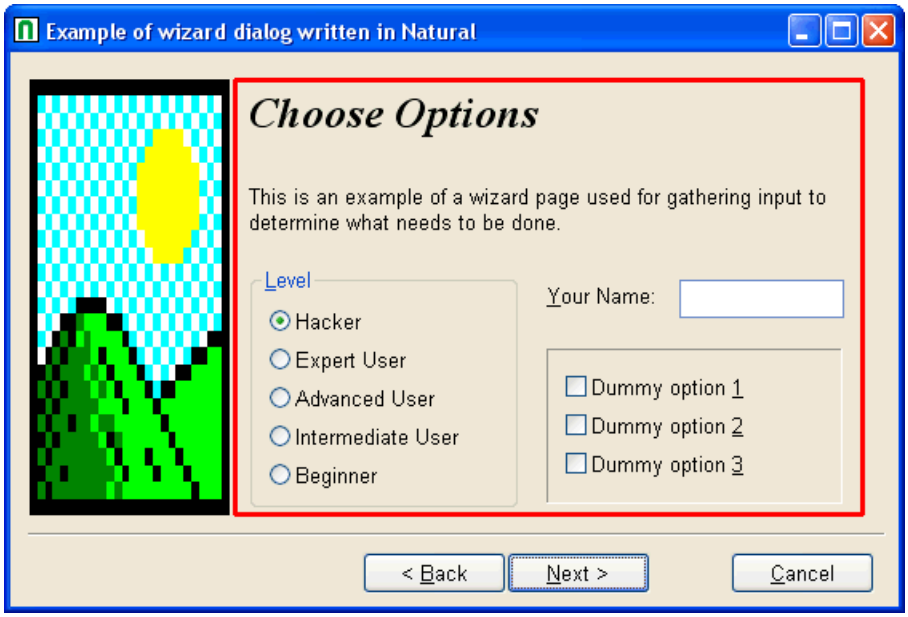
前述の排他的コントロールボックスは、ダイアログの同一リージョンを占める複数のコントロールが重なり合った「ページ」の管理が必要な状況に対応することを主目的としています。多くのコントロールが互いに部分的または完全に重なることがあるため、排他的コントロールボックスによる自動的に隠す機能を使用しないと、ユーザーがダイアログエディタでこの状況を処理するのは非常に困難です。編集中にコントロールシーケンスの最初にコントロールを移動することもできますが、これは非常に不便な方法であり、処理を続ける前にコントロールを元の位置に戻すのを覚えておく必要があります。

排他的コントロールボックスを使用すると、コントロールを選択するのと同じくらい簡単に、このような状況でコントロールを編集できます。現在表示されていないコントロールの場合、ダイアログエディタのステータスバーのコンボボックスを使用するか、またはターゲットコントロールに到達するまで Tab キーを押してコントロールを順番に移動することによって、選択できます。排他的コントロールボックスの子孫であるコントロールを選択すると、排他的コントロールボックスが表示されていなかった場合は表示され、それまで表示されていた排他的コントロールボックスが非表示になります。この変化は、ダイアログの生成ソースコードおよびランタイム状況に影響を与えません。

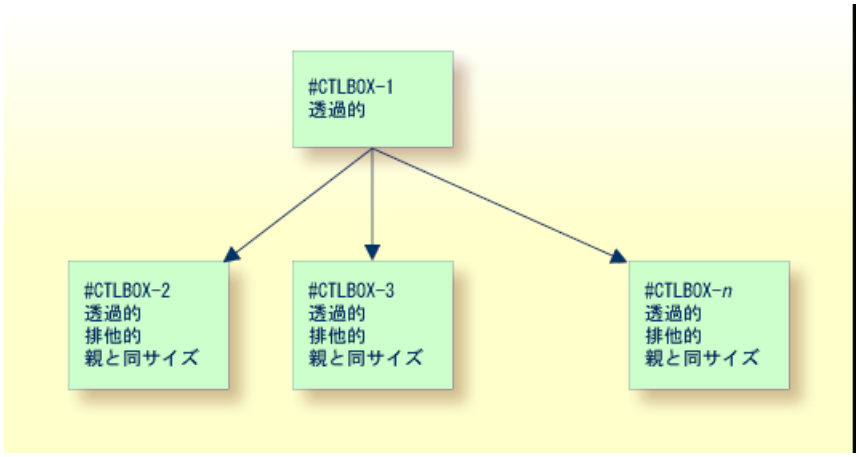
排他的コントロールボックスの使用例

コントロールボックスはできるだけ汎用性を持つことを意図して設計されましたが、コントロールページが重なることが望まれる（そのため、排他的コントロールボックスが極めて有効な）2つの状況について説明します。

- ウィザードダイアログ
- タブ化されたダイアログ（「プロパティシート」）



赤で強調された四角の中には、いわゆる "ウィザードページ" が表示されます。 必要な機能を実装するために、このエリア内で 2 階層のコントロールボックスを使用します。



#CTLBOX-1 は、「マスタ」 コントロールボックスとして使用され、後でページのサイズ変更を容易にするために必要です。 すべての子コントロールボックスにスタイル"親と同サイズ"がマークされているため、#CTLBOX-1 をサイズ変更するだけでウィザードページエリアをサイズ変更できます。

子コントロールボックスは実際のウィザードページを実装するのに使用されます。 #CTLBOX-2 はウィザードページ 1 用のコントロールを含み、#CTLBOX-3 はウィザード 2 用のコントロールを含むといった具合になります。

ウィザードページの作成

通常、ウィザードページを作成するには以下の手順を実行します。

1. 他のコントロールに対する最上位（「マスタ」）のコントロールボックスを作成します。
2. 属性ウィンドウで、"透過" スタイルを設定します。
3. 最初のコントロールボックス内で別のコントロールボックスを作成します。コントロールボックスは常にコンテナであるため、新規コントロールボックスは自動的に最初のコントロールボックスの子になります。
4. 子コントロールボックスの属性ウィンドウで、スタイル "透過"、"排他制御"、および "親と同サイズ" を設定します。 "親と同サイズ" を設定すると、子コントロールボックスがコンテナいっぱいに広がります。
5. これで、ウィザードページ 1 となる新規作成コントロールボックスへのコントロールの追加を始めることができます。
6. 新しいウィザードページを追加するための最も簡単な方法は、新しいページの直前としたい子コントロールボックスを選択してコマンドをコピーおよび貼り付けることです。 Natural は、コピーをする前に、子コントロールもコピーするかどうかを確認します。この質問に **No** と答えてください。
7. 新規追加した子コントロールボックスにも排他フラグが設定されているため、前に表示されていた子コントロールボックスが非表示になり、新しい空白のコントロールボックスが表示されます。最初のウィザードページに対して新しいコントロールセットを追加する準備ができました。

編集時におけるウィザードページの切り替え

編集時にページを切り替えるための最も簡単な方法は、ダイアログエディタのステータスバー上のコンボボックスから、該当するページに対する子コントロールボックスまたはページ上のコントロールの 1 つを選択することです。

境界線の作成

プッシュボタンとウィザードページとの境界線は、キャプションのない非常に細い（2 ピクセル）グループボックスとして実装できます。 薄く見えているグループボックスの境界線は、コントロールシーケンス内のグループフレームの後に透過的なコントロールボックスを使うことによって隠すことができます。 この手法を使用するにはダイアログの "コントロールクリッピング" スタイルを確実に設定してください。

Back プッシュボタンと Next プッシュボタンの実装

最初に、現在アクティブなページのハンドルを格納するために、以下の例のようにダイアログのローカル変数を定義します。

```
01 #ACTPAGE HANDLE OF CONTROLBOX ...
```

次に、ダイアログの AFTER-OPEN イベントで、この変数に最初のウィザードページのハンドルを設定します。

```
#ACTPAGE := #CTLBOX-1.FIRST-CHILD ..
```

#CTLBOX-1 は、最上位コントロールボックスのハンドルです。 これで、**Next** プッシュボタン（#PB-NEXT）に対する CLICK イベントコードを実装する準備ができました。 以下のように実装してください。

```
IF #ACTPAGE.SUCCESSOR = NULL-HANDLE
  CLOSE DIALOG *DIALOG-ID
ELSE
  REPEAT
    #ACTPAGE := #ACTPAGE.SUCCESSOR
    WHILE #ACTPAGE.ENABLED = FALSE
  END-REPEAT
  #ACTPAGE.VISIBLE := TRUE
  IF #ACTPAGE.SUCCESSOR = NULL-HANDLE
    #PB-NEXT.STRING := 'Finish'
    #PB-BACK.ENABLED := FALSE
    #PB-CANCEL.ENABLED := FALSE
  ELSE
    #PB-BACK.ENABLED := TRUE
  END-IF
END-IF
..
```

後でさらにウィザードページを追加する場合もこのロジックは変わらないことに注意してください。 また、対応するコントロールボックスが使用不可にされた中間ウィザードページは無視されることにも注意してください。 その結果、関連するコントロールボックスの ENABLED 属性を FALSE に設定するだけで、前の入力に基づいて特定のウィザードページを省略できます。 最後のページに達すると、**Next** プッシュボタンが "Finish" プッシュボタンに変わります。

Back プッシュボタン（#PB-BACK）の CLICK イベントコードも非常によく似ています。

```
REPEAT
  #ACTPAGE := #ACTPAGE.PREDECESSOR
  WHILE #ACTPAGE.ENABLED = FALSE
END-REPEAT
IF #ACTPAGE.PREDECESSOR = NULL-HANDLE
  #PB-BACK.ENABLED := FALSE
END-IF
#ACTPAGE.VISIBLE := TRUE
..
```

ダイアログエディタで最初は **Back** プッシュボタンを使用不可にすべきであることに注意してください。

ウィザードページ上の全コントロールの消去

ウィザードページ上の全コントロールの消去は、関連するページの最上位のコントロールを選択して、コントロールのすべての兄弟も選択するために［編集］メニューの［すべての選択］を選択することによって簡単に行えます。 これにより、選択したコントロールが通常どおり削除されます。

例2 - タブ化されたダイアログ

タブ化されたダイアログ（「プロパティシート」 とも呼ばれる）は、ウィザードダイアログの概念とよく似ています。 唯一の本質的な違いは、**Next** および **Back** プッシュボタンを使用してコントロール 「ページ」 間を移動するのではなく、ユーザーが該当するタブをクリックすることによって希望するページを直接アクセスする点です。 コントロールページの階層は、上記のウィザードダイアログの例と同じ方法で、ダイアログエディタで構成および処理できます。 実際のタブを提供するいくつかの ActiveX コントロールを使用できます。

ただし、ページの切り替え（つまり、対応するコントロールボックスの切り替え）は自動的に行われないことに注意する必要があります。 Natural プログラマが、タブ切り替えによって発生する ActiveX イベントに対してコードを挿入し、選択されたタブを調べ、該当する（排他的な）コントロールボックスの VISIBLE 属性に TRUE を設定する必要があります。 各 ActiveX コントロールは選択された任意の方法で機能を実装するため、Natural が暗黙のうちに実装することができないからです。 タブ切り替えによって発生する標準イベント、および現在アクティブなタブを決定するための標準パラメータ付きの標準メソッド（または標準プロパティ）はありません。

Microsoft の "Tab Strip" ActiveX コントロール（*V4-NEST.NS3*）を使用したタブ化されたダイアログの例が、Natural サンプルライブラリに提供されています。

93 日付／時刻ピッカー（DTP）コントロールの操作

このchapterでは、次のトピックについて説明します。

はじめに

日付／時刻ピッカー（DTP）コントロールは、ユーザーによる日付または時刻情報の入力を簡素化するために使用します。 DTP コントロールは、時刻入力に対しては、スピンコントロールに似た外観と動作をします。任意の日付入力に対しては、スピンコントロールまたは選択ボックスのいずれかに似た外観と動作をします。日付入力の場合、下矢印が表示されているボタンをユーザーがクリックすると、従来のリストボックスではなく、カレンダーが表示されます。

日付／時刻のフォーマット

デフォルトでは、日付／時刻の情報は、現在の地域の設定に定義されている日付／時刻のフォーマットに従って表示されます。 Windows には長いフォーマットと短いフォーマットの 2 つの代替日付フォーマットがあり（両方ともユーザーが変更可能）、また短い日付フォーマットには世紀の情報が含まれないことがあるため、3 つの STYLE フラグの 1 つにより、使用される標準の日付フォーマットが決定されます。 これらのフォーマットは以下のとおりです（相互排他的）。

- ［短い日付 (s)］。現在の地域の設定に短い日付フォーマットを使用することを意味します。
- ［世紀日付 (c)］。現在の地域の設定に短い日付フォーマットを使用することを意味しますが、世紀情報が含まれていない場合は、世紀情報を提供するように拡張されます。 多くの場合、短い日付フォーマットには世紀情報がすでに含まれています。この場合、このスタイルにより日付の外観が変わることはありません。
- ［長い日付（l）］。現在の地域の設定に長い日付フォーマットを使用することを意味します。

補足：［時刻（t）］スタイルフラグは、コントロールで（日付ではなく）時刻情報を表示することを示すために用意されています。

これらの標準のフォーマットが十分でない場合は、EDIT-MASK 属性を使用して、これらのフォーマットよりも優先させるカスタムフォーマット文字列を指定することができます。 ただし、フォーマット文字列の指定子は、Natural 内の他の場所にある編集マスクに使用された指定子には対応していません。 次のテーブルに、使用可能な指定子とその意味を示します。

指定子	説明
d	1 桁または 2 桁の日付。
dd	2 桁の日付。 1 桁の日付の場合、先頭に 0 が付けられます。
ddd	3 文字の週日の略記。
dddd	完全な週日名。
h	12 時間フォーマットの 1 桁または 2 桁の時間。

指定子	説明
hh	12 時間フォーマットの 2 桁の時間。 1 桁の値の場合、先頭に 0 が付けられます。
H	24 時間フォーマットの 1 桁または 2 桁の時間。
HH	24 時間フォーマットの 2 桁の時間。 1 桁の値の場合、先頭に 0 が付けられます。
m	1 桁または 2 桁の分。
mm	2 桁の分。 1 桁の値の場合、先頭に 0 が付けられます。
s	1 桁または 2 桁の秒。
ss	2 桁の秒。 1 桁の値の場合、先頭に 0 が付けられます。
M	1 桁または 2 桁の月番号。
MM	2 桁の月番号。 1 桁の値の場合、先頭に 0 が付けられます。
MMM	3 文字の月の略記。
MMMM	完全な月名。
t	AM/PM 略記の 1 文字（つまり、AM は "A" と表示される）。
tt	AM/PM 略記の 2 文字（つまり、AM は "AM" と表示される）。
yy	年の下 2 桁（つまり、2005 年は "05" と表示される）。
yyyy	完全な年（つまり、2005 年は "2005" と表示される）。

また、引用符で囲まれた文字は、指定したとおりに表示されます。 引用符で囲まれた引用符を指定する場合は、2つの一重引用符を連続して使用する必要があります。 スペースおよび句読点の記号（コンマなど）は、引用符で囲む必要はありません。

例えば、"John's birthday is Friday, December 31, 1969" という文字列を表示するには、DTP コントロールの `EDIT-MASK` 属性を "John''s birthday is' dddd, MMMM d, yyyy" に設定します。

日付および時刻の入力

DTP コントロールには、指定された情報を修正するための方法がいくつかあります。

- ユーザーが、数値情報（日付など）を直接入力します。
- ユーザーが、選択されたフィールド（日付、月名など）の値を、+ キーまたは - キーを使用して、それぞれ増やしたり減らしたりします。
- DTP コントロールのスタイルが［時刻 (t)］または［上下 (u)］のいずれかの場合、ユーザーが、必要なフィールドを選択し、上下（"スピン"）コントロールを使用して数値を増やしたり減らしたりします。
- DTP コントロールでカレンダーが使用されている場合、ユーザーが、下向きの矢印を押してカレンダーを開き、必要な日付に移動します。 上述した方法とは異なり、この方法ではすべての日付フィールドが同時に更新されます。
- プログラム的に、必要な日付または時刻で `TIME` 属性を更新します。

例えば、DTP コントロールで日付または時刻を現在の日付または時刻に設定するには、次の割り当てを使用します。

```
#DTP-1.TIME := *DATX
```

または

```
#DTP-1.TIME := *TIMX
```

それぞれ、#DTP-1 は DTP コントロールのハンドルであるとみなされます。

DTP コントロールでは、日付または時刻のコンポーネントの編集しか許可されませんが、コントロールのスタイルに応じて、日付および時刻の両方の情報を保存します。

DTP コントロールの日付または時刻がユーザーによって変更されると、そのコントロールの CHANGE イベントが発生します（抑制されていない場合）。 DTP コントロールがプログラムによって変更された場合、このイベントは発生しません。

空値

DTP コントロールに［値なし"" を許可 (n)］のスタイルが指定されている場合、コントロールにチェックボックスが表示されます。 このチェックボックスがオンになっていない場合、そのコントロールに関連付けられた日付または時刻はないと解釈されます。 アプリケーションは、コントロールの CHECKED 属性をクエリすることで、この状態をテストすることができます。 また、CHECKED 属性を UNCHECKED に戻すことで、"値なし"の状態にコントロールを戻すこともできます。 ただし、CHECKED 属性を明示的に CHECKED に設定することはできません。これは、日付または時刻がコントロールに適用されるたびに、暗黙的に行われるためです。 さらに、CHECKED 属性は、［値なし"" を許可 (n)］スタイルを使用しない場合、DTP コントロールにまったく設定できません。

カレンダーのカラーおよびフォント

DTPコントロールと関連してカレンダー（存在する場合）が使用するカラーおよびフォントは、それぞれ SET-AUX-COLOR アクションおよび SET-AUX-FONT アクションを使用して変更できます。

94 ダイアログバーコントロールの操作

このchapterでは、次のトピックについて説明します。

はじめに

ダイアログバーは、ダイアログのフレームの内部側辺の 1 つにドッキングまたは（オプションで）別個のフローティングウィンドウに表示可能なツールバーコントロールに類似しています。ただし、ツールバーコントロールと異なり、ダイアログバーコントロールは、汎用コンテナコントロールとして考えられており、主にツールバー項目を含むためだけのものとしては考えられていません。 さらに、ツールバーコントロールとダイアログバーコントロールには表示や動作に多数の違いがあります。その一部を説明します。

ダイアログバーコントロールのよい例がNaturalスタジオのライブラリワークスペースウィンドウです。

ダイアログバーコントロールの作成

ダイアログバーコントロールは、他の標準コントロール（リストボックスやプッシュボタンなど）と同じようにダイアログエディタで作成します。 つまり、ダイアログエディタの［挿入］メニューを使用して、あるいは［挿入］ツールバーからのドラッグ & ドロップによってスタティックに作成するか、または TYPE 属性を DIALOGBAR に設定した PROCESS GUI ACTION ADD ステートメントを使用してランタイム時にダイナミックに作成します。

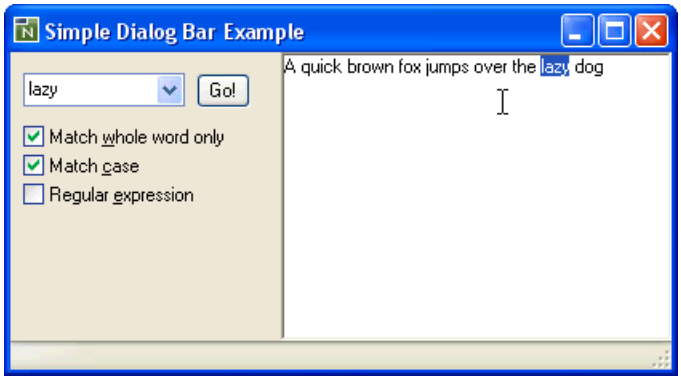
ダイアログバーコントロールのタイプ

ダイアログバーコントロールは以下の 3 つの基本形式（複雑さの順番）のいずれかで存在します。

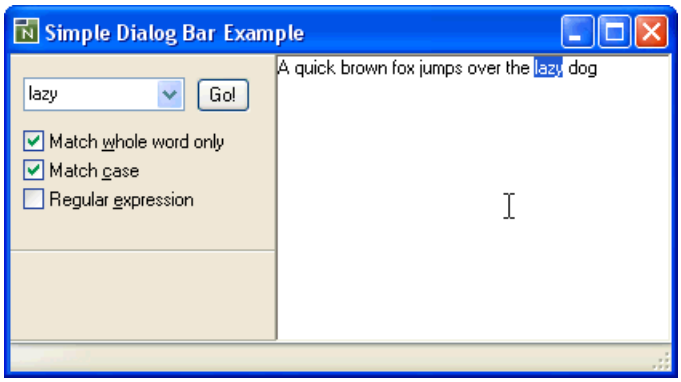
1. ドッキングもサイズ変更も不可能。
2. ドッキングは可能だが、サイズ変更は不可能。
3. ドッキングもサイズ変更も可能。

DRAGGABLE 属性が設定されている場合、ダイアログバーコントロールはドッキング可能です。［ダイナミック（Y）］ STYLE フラグが設定されている場合、サイズ変更可能です。

以下の例は、ドッキングもサイズ変更もできないダイアログバーの例を示しています。 右側の編集エリアはダイアログのクライアントエリア全体を満たしています。 ダイアログバーは、ドラッグ不可能で、配置されている側辺の長さいっぱいに拡張します。

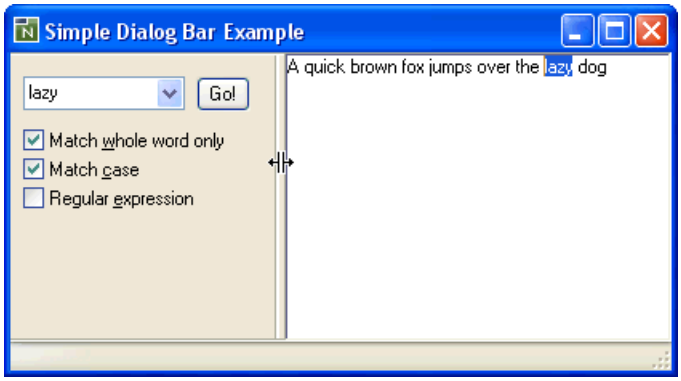


ダイアログエディタの［ダイアログバー属性］ウィンドウで「［ドッキング可能］」状態を設定すると、ウィンドウがドラッグ可能になります。 ダイアログバーは、ドッキングする側辺の全長を使用するために拡張しないことにも注意してください（別のダイアログバーコントロールまたはツールバーコントロールをその下にドッキングできます）。

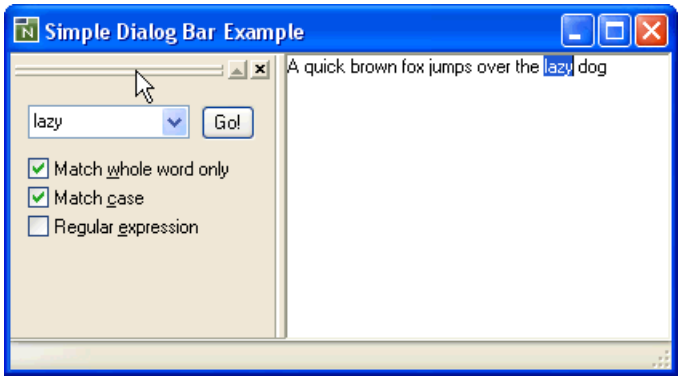


ユーザーは、ダイアログバーコントロールをドラッグし、それをオーナーダイアログの別の側辺に再度ドッキングしたり、以下のように別個のフローティングウィンドウにしたりできます。

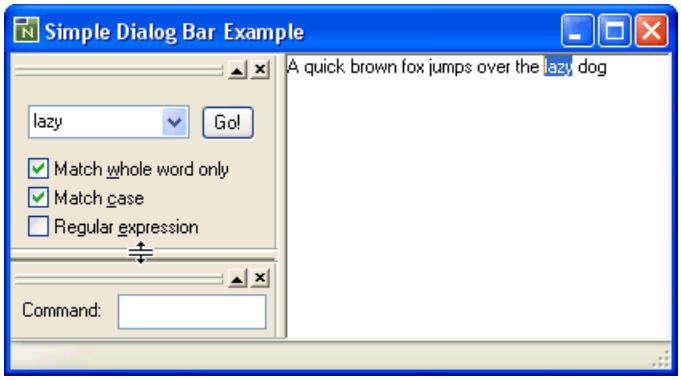
ダイアログバーコントロールを（属性ウィンドウの［ダイナミック（Y）］スタイルフラグをチェックして）サイズ変更可能にすると、垂直スプリットバーが表示され、ダイアログバーコントロールがサイズ変更可能になります。サイズ変更可能なダイアログバーコントロールは、ドッキングする側辺（サイズ変更不可能なバーが使用していない側辺）の全長を満たすために拡張することに注意してください。



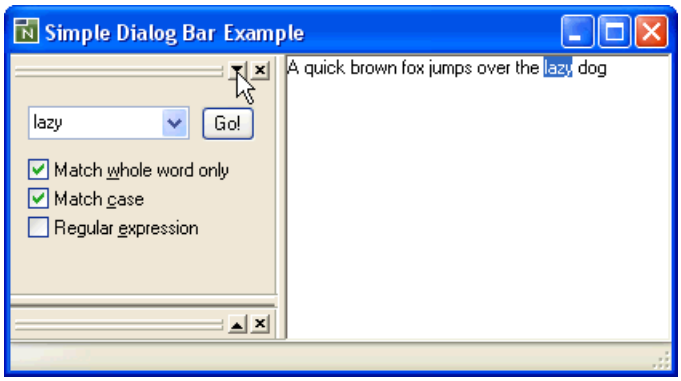
グリッパバー、ズームボタン、および閉じるボタンが（属性ウィンドウの［グリッパ（g）］、［ズームボタン（z）］、および［閉じるボタン（x）］の各スタイルフラグをチェックして）追加されると、ダイアログバーコントロールは、Naturalスタジオのライブラリワークスペースを表示するために使われているコントロールに似た外観になります。 同じ列にサイズ変更可能なダイアログバーが他にないので、ズームボタンが無効になっていることに注意してください。



2つ目のサイズ変更可能なダイアログバーコントロールを追加して、同じ列の最初のもののそばにドッキングすると、水平スプリットバーが表示され、2つのダイアログバーコントロールの相対的なサイズが変更可能になります。 ズームボタンが有効になったことに注意してください。



ズームボタンをクリックすると、サイズ変更可能なダイアログバーコントロールの最大化の状態と元の状態とが切り替わります。 ダイアログバーコントロールの最大化により、同じ列の他のサイズ変更可能なダイアログバーコントロールは最小化され、以下のように、最大化したバーが使用できるようにスペースが解放されます。



最大化したバー上のズームボタンの近くに表示される矢印の方向が、次にこのボタンが押されるとバーが（最大化ではなく）元のサイズに戻されることを示すために変わっていることに注意してください。 バーが復元されると、列上のすべてのサイズ変更可能なダイアログバーは標準のサイズに戻ります。 このサイズは通常、それまで列に表示されていたバーに変更（例：表示されているバーを非表示にする、非表示のバーを表示する、新規バーを列にドッキングするなど）がない限り、最大化される前のサイズになります。

列上のバーの長さが水平スプリットバーによって変更された場合、列上に表示されているすべてのバーは自動的に復元されます。

UI 透過

ドッキング可能なダイアログバーコントロールは、通常はそのグリッパバー（存在する場合）またはその背景をつかんでドラッグできます。 ただし、［UI 透過（T）］スタイルが設定されている場合、バーはグリッパバー（存在する場合）経由でのみドラッグできます。そのような（サイズ変更可能な）バーにグリッパバーがない場合、コントロールのサイズ変更はスプリットバー（複数可能）経由でのみ可能です。これは状況によっては適切な機能です。また、グリッパバー経由のドラッグのみを可能にすると、意図しないドラッグ操作の実行を回避できます。

Client-Size イベント

ダイアログのクライアントウィンドウは、ツールバー、ステータスバー、およびダイアログバーコントロールによって使われている領域を排除するためにサイズを縮小します。 ドッキング可能なウィンドウ（ツールバーまたはダイアログバーコントロール）がフローティングになる、オーナーウィンドウの別の側辺に再ドッキングされる、または表示されたり非表示にされたりすると、ウィンドウの外側の大きさが変わっていなくても、クライアントウィンドウのサイズが変更されます。 SIZE イベントは、ダイアログの外側のサイズの変更のために確保されているため、クライアントウィンドウのサイズを把握する必要のあるアプリケーションは、この目的のために SIZE イベントではなく CLIENT-SIZE イベントを使用します。 ダイアログクライアントウィンドウの実際のサイズは、このイベントで INQ-INNER-RECT アクションを使用することによって、確認できます。

閉じるボタン

閉じるボタン（存在する場合）はダイアログバーコントロールを閉じるのではなく、非表示にします。これは、フローティングツールバーコントロールの閉じるボタンでも同様です。 バーを再表示する方法は、アプリケーションによって提供されます。 次のセクションでは、これを実行するコードを（他の内容と一緒に）紹介します。

サンプルコード

以下のリストは、ユーザーによるツールバーとダイアログバーの表示の制御を可能にする、ほとんどのケースで「このまま」 使用できる外部サブルーチンの完全なリストです。 コードは、MDI アプリケーションで動作できるように設計されていますが、MDI 以外（つまり、SDI）のアプリケーションでも動作します。

サブルーチンは、ツールバーおよびダイアログバーのキャプション（STRING 属性）をダイアログのコンテキストメニューに付加します。 ダイアログにコンテキストメニューがない場合、コ

ンテキストメニューが 1 つ作成され、自動的にダイアログに割り当てられます。 MDI アプリケーションでは、ツールバーとダイアログバーの一部はグローバル（すべてのタイプの MDI 子ダイアログが該当）であり、一部はプライベート（1 つのタイプの MDI 子ダイアログだけが該当）であることが想定されています。例えば、Natural スタジオでは、オブジェクトツールバーはグローバルツールバーであり、ダイアログエディタオプションツールバーはダイアログエディタに対するプライベートなツールバーです。ユーザーが MDI 子ダイアログを切り替えると、グローバルツールバーと現在アクティブなダイアログに関連付けられているプライベートツールバーだけが表示されるように、コンテキストメニューが変更されます。 さらに、このダイアログが最後に表示されたときと同じプライベートバーが表示されます（［ダイアログ属性］ウィンドウの［レイアウト保存］チェックボックスがチェックされている場合、表示されたバーのサブセットもセッション間で保持されます）。

このサブルーチンは、以下のように、メインダイアログ（MDI アプリケーションの MDI フレームダイアログ）の AFTER-ANY イベントハンドラで呼び出す必要があります（メインダイアログのハンドル変数名は #DLG$WINDOW のデフォルト値に設定されているとします）。

```
PERFORM PROCESS-BAR-COMMANDS #DLG$WINDOW
```

その他に、以下の手順を任意で実行します。

1. バーは、コントロールシーケンスに表示される順番でコンテキストメニューにリストされます。 したがって、ツールバーおよびダイアログバーの順序付けの再実行が必要な場合があります（例：MDI アプリケーションでプライベートツールバーの前にグローバルツールバーが表示されるようにするため）。
2. コードは、コンテキストメニューで有効なバーのリストの前にセパレータを挿入しません。したがって、ダイアログに対してすでにコンテキストメニューを使用している場合、コンテキストメニューがセパレータで終了することを確認する必要がある場合があります。
3. MDI アプリケーションでは、プライベートバーごとに、ダイアログエディタでバーの属性ウィンドウの［コントロール **ID**］フィールドに、そのツールバーが 「属する」 ダイアログの名前（例：ダイアログのファイル名が *CHILD.NS3* であれば "CHILD"）を入力する必要があります。各グローバルバーに対しては、このフィールドは空にしておきます。MDI 子ダイアログがアクティブでないときにのみバーを表示する場合、ここに MDI フレームダイアログの名前を入力します。
4. MDI アプリケーションでは、デフォルトで表示しない各バーに対し、属性ウィンドウの［有効化］チェックボックスをオフにする必要があります。

```
DEFINE DATA
PARAMETER
  1 #DIALOG     HANDLE OF GUI
LOCAL
  1 #CONTROL       HANDLE OF GUI
  1 #ACTIVE-DLG    HANDLE OF GUI
  1 #CTXMENU       HANDLE OF CONTEXTMENU
  1 #MITEM-DYN     HANDLE OF MENUITEM
```

```
LOCAL USING NGULKEY1
END-DEFINE
*
DEFINE SUBROUTINE PROCESS-BAR-COMMANDS
  DECIDE ON FIRST *EVENT
    VALUE 'COMMAND-STATUS'
       PERFORM COMMAND-STATUS
    VALUE 'IDLE'
       PERFORM IDLE
    VALUE 'CLICK'
      PERFORM CLICK
    VALUE 'BEFORE-OPEN'
      PERFORM BEFORE-OPEN
    VALUE 'AFTER-OPEN'
      PERFORM AFTER-OPEN
   NONE
      IGNORE
   END-DECIDE
*
   DEFINE SUBROUTINE COMMAND-STATUS
     /* Must enable our commands, otherwise they're automatically disabled!
     #CTXMENU := #DIALOG.CONTEXT-MENU
     #MITEM-DYN := #CTXMENU.FIRST-CHILD
     REPEAT WHILE #MITEM-DYN <> NULL-HANDLE
        IF #MITEM-DYN.CLIENT-HANDLE <> NULL-HANDLE
          #MITEM-DYN.ENABLED := TRUE
        END-IF
        #MITEM-DYN := #MITEM-DYN.SUCCESSOR
     END-REPEAT
   END-SUBROUTINE
*
   DEFINE SUBROUTINE IDLE
     PERFORM SWITCH-BARS
   END-SUBROUTINE
*
   DEFINE SUBROUTINE CLICK
     #CONTROL := *CONTROL
     IF #CONTROL.TYPE = MENUITEM AND #CONTROL.PARENT = #DIALOG.CONTEXT-MENU
          #MITEM-DYN := #CONTROL
          #CONTROL := #MITEM-DYN.CLIENT-HANDLE
          IF #CONTROL <> NULL-HANDLE
             IF #MITEM-DYN.CHECKED = CHECKED
                 #CONTROL.ENABLED := FALSE
                 #CONTROL.VISIBLE := FALSE
         ELSE
                #CONTROL.ENABLED := TRUE
                #CONTROL.VISIBLE := TRUE
         END-IF
       END-IF
      END-IF
   END-SUBROUTINE
*
```

```
  DEFINE SUBROUTINE BEFORE-OPEN
     #CTXMENU := #DIALOG.CONTEXT-MENU
     #MITEM-DYN := #CTXMENU.FIRST-CHILD
     REPEAT WHILE #MITEM-DYN <> NULL-HANDLE
        IF #MITEM-DYN.CLIENT-HANDLE <> NULL-HANDLE
          #CONTROL := #MITEM-DYN.CLIENT-HANDLE
        IF #CONTROL.VISIBLE
          #MITEM-DYN.CHECKED := CHECKED
       ELSE
          #MITEM-DYN.CHECKED := UNCHECKED
       END-IF
     END-IF
     #MITEM-DYN := #MITEM-DYN.SUCCESSOR
    END-REPEAT
   END-SUBROUTINE
*
   DEFINE SUBROUTINE AFTER-OPEN
     /* for MDI frames, unsuppress IDLE event to track active child change
     IF #DIALOG.TYPE = MDIFRAME
       #DIALOG.SUPPRESS-IDLE-EVENT := NOT-SUPPRESSED
     END-IF
     /* if dialog has no context menu, create one
     #CTXMENU := #DIALOG.CONTEXT-MENU
     IF #CTXMENU = NULL-HANDLE
         PROCESS GUI ACTION ADD WITH PARAMETERS
            HANDLE-VARIABLE = #CTXMENU
            TYPE = CONTEXTMENU
            PARENT = #DIALOG
     END-PARAMETERS GIVING *ERROR
     #DIALOG.CONTEXT-MENU := #CTXMENU
   END-IF
   /* unsuppress context menu's BEFORE-OPEN event for item update
   #CTXMENU.SUPPRESS-BEFORE-OPEN-EVENT := NOT-SUPPRESSED
   /* display bars according to context
   PERFORM SWITCH-BARS
  END-SUBROUTINE
*
  DEFINE SUBROUTINE SWITCH-BARS
     IF #DIALOG.TYPE = MDIFRAME
       #ACTIVE-DLG := #DIALOG.ACTIVE-CHILD
     END-IF
     IF #ACTIVE-DLG = NULL-HANDLE
       #ACTIVE-DLG := #DIALOG
     END-IF
     IF #ACTIVE-DLG <> #DIALOG.CLIENT-HANDLE
       #CTXMENU := #DIALOG.CONTEXT-MENU
       IF #CTXMENU <> NULL-HANDLE
         /* Remove any dynamic menu items previously created
         #CONTROL := #CTXMENU.FIRST-CHILD
         REPEAT WHILE #CONTROL <> NULL-HANDLE
           #MITEM-DYN := #CONTROL.SUCCESSOR
           IF #CONTROL.CLIENT-HANDLE <> NULL-HANDLE
```

```
            PROCESS GUI ACTION DELETE WITH #CONTROL
          END-IF
          #CONTROL := #MITEM-DYN
        END-REPEAT
        /* Search for all tool bar and dialog bar controls
        #CONTROL := #DIALOG.FOLLOWS
        REPEAT WHILE #CONTROL <> #DIALOG
           IF #CONTROL.TYPE = TOOLBARCTRL OR
               #CONTROL.TYPE = DIALOGBAR
             #CONTROL.CLIENT-KEY := 'CONTROL-ID'
             IF #CONTROL.CLIENT-VALUE = ' ' OR
                 #CONTROL.CLIENT-VALUE = #ACTIVE-DLG.NAME
              #CONTROL.VISIBLE := #CONTROL.ENABLED
              /* Create menu entry for bar
              PROCESS GUI ACTION ADD WITH PARAMETERS
                HANDLE-VARIABLE = #MITEM-DYN
                TYPE = MENUITEM
                PARENT = #CTXMENU
                STRING = #CONTROL.STRING
                SUCCESSOR = #MITEM-DYN
                CLIENT-HANDLE = #CONTROL
              END-PARAMETERS GIVING *ERROR
             ELSE
               #CONTROL.VISIBLE := FALSE
             END-IF
           END-IF
           #CONTROL := #CONTROL.FOLLOWS
         END-REPEAT
       END-IF
       /* Save handle of currently active dialog
       #DIALOG.CLIENT-HANDLE := #ACTIVE-DLG
     END-IF
  END-SUBROUTINE
END-SUBROUTINE
END
```

95 エラーイベントの操作

ダイアログがアクティブな間にランタイムエラーが発生すると、ダイアログはエラーイベントを受け取ります。 このエラーが発生すると常に実行されるイベントハンドラコードを指定できます。エラーイベントハンドラコードを指定しないと、Naturalはエラーメッセージを出力して異常終了し、すべてのダイアログが閉じられます。

イベントハンドラコードの最後に ESCAPE ROUTINE ステートメントを指定すると、エラー処理後に通常のダイアログ処理を続行できます。

ダイアログエディタはイベントハンドラに対し ON ERROR ステートメントを生成します。 例えば、パラメータ ZD が ON に設定されていて、かつ、エンドユーザーが整数をゼロで割ろうとしたときにアプリケーション全体が終了するのを回避する場合、エラーイベントハンドラコードは以下のようになります。

```
COMPRESS 'Natural error' *ERROR 'occurred.' INTO #DLG$WINDOW.STATUS-TEXT
    ESCAPE ROUTINE
```

96 ラジオボタンコントロールグループの操作

ラジオボタンコントロールは、プッシュボタンコントロールまたはトグルボタンコントロールと同様に作成します。ただし、ラジオボタンコントロールは GROUP-ID 属性を使用してグループ化します。多くのラジオボタンを1つのグループとして定義すると、1つのボタンだけを選択できます。 この選択ロジックを提供するのが GROUP-ID 属性です。

複数のラジオボタンコントロールをグループ化するには、ラジオボタンコントロールに属性ウィンドウで同一の GROUP-ID 値（グループ番号）を割り当てます。エンドユーザーが1つのラジオボタンコントロールをクリックすると、ダイアログ内の同一 GROUP-ID を持つ他のすべてのラジオボタンコントロールが選択解除されます。 下記のようなコードで 1 つのラジオボタンコントロールが選択された場合も同様です。

```
...
   1 #RB-1 HANDLE OF RADIOBUTTON
...
#RB-1.CHECKED := CHECKED  /* Set the CHECKED attribute to value CHECKED
...
```

キーボードを使用してラジオボタンコントロールグループ内を移動できるようにする必要があることにも注意してください。例えば、Tab を押すと最初のラジオボタンコントロールが選択され、矢印キーによってラジオボタングループ内を移動できるようにします。Natural でこのような移動が自動的に行われるようにするには、ラジオボタンコントロールのナビゲーションシーケンスがお互いに連続するようにする必要があります。 PROCESS GUI ステートメントの ADD アクションでラジオボタンをダイナミックに追加する場合は、ボタンの FOLLOWS 属性に値を指定することによってこれを実現できます。

手順 96.1. ナビゲーションシーケンスを編集するには

■ ［ダイアログ］メニューの［操作順序設定］を選択します。

97 イメージリストコントロールの操作

このchapterでは、次のトピックについて説明します。

はじめに

イメージリストコントロールは、リストビューコントロールやツリービューコントロールなどの特定のコントロールタイプに関連付けることができる、順序付けされたイメージのコンテナです。 イメージリストコントロールを使用すると、ディスクからイメージを毎回再ロードすることなく、コントロールの項目で効率的にイメージを再利用できます。 また、イメージはすべて互換性があります（同じサイズおよびカラー構成など）。

イメージリストコントロールの作成

通常は、ADD アクションを使用してイメージリストコントロールを作成します。

```
PROCESS GUI ACTION ADD WITH
PARAMETERS
   HANDLE-VARIABLE = #IMGLST-1
   TYPE = IMAGELIST
   PARENT = #DLG$WINDOW
   STYLE = 'LS'
END-PARAMETERS GIVING *ERROR
```

イメージリストコントロールは、最大2つのイメージセットによって内部的に構成されます。その1つは大きいイメージ（通常は32x32ピクセル）で構成され、もう1つは小さいイメージ（通常は16x16ピクセル）で構成されます。 これらのどちらが内部的に作成されるかは、イメージリストコントロールの［大きいイメージ（L）］および［小さいイメージ（S）］の各 STYLE フラグによって決まります。どちらのフラグも指定されていない場合、イメージリストコントロールの ITEM-W および ITEM-H の各属性値に基づいたイメージサイズで、単一のイメージセットが作成されます。 これらの両方の値がゼロの場合、小さいイメージが使用されます。

イメージの追加

必要なイメージ（ビットマップまたはアイコン）ファイルに基づいて、イメージリストコントロールの子としてイメージコントロールを作成することにより、イメージリストにイメージを追加できます。

```
PROCESS GUI ACTION ADD WITH
PARAMETERS
   HANDLE-VARIABLE = #IMG-1
   TYPE = IMAGE
   PARENT = #IMGLST-1
   BITMAP-FILE-NAME = 'example.bmp'
END-PARAMETERS GIVING *ERROR
```

デフォルトでは、SUCCESSOR属性を使用して特定の位置に項目を挿入するよう指定されていない限り、イメージはリストの最後に追加されます。

合成イメージ

イメージコントロールは、単一イメージコントロールと複数イメージコントロールの2つに分類できます。

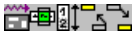
単一イメージコントロールは、親イメージリストコントロールで使用されている各イメージセットに対し、単一のイメージを提供します。 つまり、イメージリストで大きいイメージと小さいイメージの両方を使用する場合、どちらに対してもこのイメージコントロールが使用されます。 単一イメージコントロールは、ビットマップまたはアイコンになります。

複数イメージコントロールは、その名前が示すとおり、親イメージリストコントロールに対し、（サイズごとに）複数のイメージを提供します。 複数イメージコントロールでは、アイコンではなくビットマップファイルを使用する必要があり、［合成イメージ（C）］STYLE フラグを設定して単一イメージコントロールと区別します。

```
PROCESS GUI ACTION ADD WITH
PARAMETERS
   HANDLE-VARIABLE = #IMG-1
   TYPE = IMAGE
   PARENT = #IMGLST-1
   STYLE = 'CsT'
```

```
   BITMAP-FILE-NAME = 'composite.bmp'
END-PARAMETERS GIVING *ERROR
```

合成ビットマップ内のイメージ数は、親イメージリストコントロールで使用されている（最も小さい）イメージセット内のイメージの幅と高さ、およびビットマップのサイズから自動的に計算されます。 したがって、大きいイメージと小さいイメージの両方が使用されている場合、通常は、ビットマップの高さは16ピクセル、幅は16*Nピクセルになります。Nはイメージコントロールに格納されているイメージ数です。以下に、5つのイメージが格納されている合成ビットマップの例を示します。



伸縮と透過

前のセクションの例では、［合成イメージ（C）］スタイルの他に［伸縮（S）］および［透過（T）］という2つのスタイルが指定されていました。親イメージリストコントロール内でサイズの異なる複数のイメージセットを使用する場合、［伸縮（S）］スタイルを無条件に指定する必要があります。 例えば、大きいイメージも使用されている場合、以下のように、構成イメージに分割する前に、このスタイルフラグによって合成イメージが内部的に拡大／縮小されます。

［伸縮（S）］スタイルフラグを指定していない場合、以下のように、合成ビットマップは、分割前に拡大／縮小されずに背景色内に展開されることに注意してください。

この場合、5つの大きいイメージは以下のように分割されます。

言うまでもなく、通常はこの結果は好ましくありません。

［透過（T）］スタイルフラグは、イメージが透過的にレンダリングされることを示します。ビットマップの背景色部分のピクセルはすべて描画されません。背景色は、イメージコントロールの BACKGROUND-COLOUR-VALUE 属性および／または BACKGROUND-COLOUR-NAME 属性に必要な値を設定することにより、明示的に指定できます。 そうしない場合、つまり、上記の例のように背景色を指定しない場合は、ビットマップの最初（つまり、左上）のピクセルの色が背景色とみなされます。

当然ながら、［伸縮（S）］および［透過（T）］の各スタイルフラグは、非合成イメージに適用できます。

ビットマップとアイコン

複数イメージコントロールのソースとして使用できない点の他にも（上記参照）、アイコンは2つの大きな点でビットマップと異なっています。 1つ目は、単一のアイコン（*.ICO*）ファイルに、同じアイコンのサイズ違いのバージョンを複数格納できるという点です。 したがって、Naturalで大きいイメージが必要とされ、ソースがアイコンファイルの場合、ファイル内の他のアイコンを拡大／縮小して作り出すのではなく、アイコンファイルに大きいアイコンが定義されていればそのアイコンが使用されます。 同様に、Naturalで小さいイメージが必要とされ、ソースがアイコンファイルの場合、アイコンファイルに小さいアイコンが定義されていればそのアイコンが使用されます。 一方、ビットマップファイルには複数のイメージを格納できないため、イメージリストで大きいイメージと小さいイメージの両方が必要とされる場合、これらの2つのイメージのどちらかは（通常は大きいイメージ）、前のセクションで説明したように、もう1つのイメージから作り出す必要があります。

2つ目は、アイコンには通常、イメージ内のどのピクセルを透過（つまり、描画しない）にするかを指定する、イメージマスクと呼ばれる白黒のビットマップが組み込まれているという点です。 これにより、Naturalがイメージをアイコンファイルからロードするときに、イメージコントロールの `BACKGROUND-COLOUR-NAME` 属性が `DEFAULT` に設定され（または指定されていない）、かつ、［伸縮（S）］スタイルフラグの指定なしで［透過（T）］スタイルフラグが指定されている場合、ビットマップからイメージをロードする場合に最初のピクセルと同じ色のピクセルがすべて透過的にレンダリングされるのとは異なり（上記参照）、アイコンの透過マスクが使用されます。 （デフォルト以外の）背景色が明示的に指定されている場合、アイコンまたはビットマップが使用されているかどうかにかかわらず、その色のすべてのピクセルが透過とみなされます。 この場合、アイコンの透過マスクは無視されます。アイコンが拡大／縮小される場合も同様です。

したがって、大きいイメージと小さいイメージの両方が必要な場合、単一の合成ビットマップに基づいた複数イメージコントロールではなく、大小それぞれのイメージが格納されている複数のアイコンファイルに基づいた単一イメージコントロールを使用することをお勧めします。 アイコン（*.ICO*）ファイルを個別に使用する長所は、小さいイメージと大きいイメージ（ファイル内に両方とも格納されていると想定）を使って異なる詳細レベルで表示できる点です。 主な短所は、通常、複数のアイコンファイルからイメージをロードするには、単一の合成ビットマップファイルからロードするより時間がかかることです。

イメージリストの使用

イメージリストのイメージをコントロール（ツリービューコントロールやリストビューコントロールなど）で使用できるようにするには、まずイメージリストをコントロールに関連付ける必要があります。 この関連付けを実現するには、イメージリストコントロールのハンドルをホストコントロールの IMAGE-LIST 属性に割り当てます。 次に例を示します。

```
#LV-1.IMAGE-LIST := #IMGLST-1
```

イメージリストの設定後は、イメージリストコントロールのイメージをコントロールの項目で使用できます。

イメージリストのイメージの参照

特定の項目（リストビュー項目やツリービュー項目など）の親コントロールのイメージリストにある、特定のイメージを使用するには、以下の2つの方法のいずれかで使用するイメージを指定する必要があります。

1. 項目の IMAGE 属性をイメージコントロールのハンドルに設定し、（必要に応じて）項目の IMAGE-INDEX 属性をイメージコントロール内の必要なイメージの相対オフセット（ゼロから始まる）に設定する。 イメージコントロールに含まれているイメージが 1 つのみの場合、イメージインデックスを指定する必要はありません。 指定するイメージは、項目のコンテナに割り当てられているイメージリストコントロールに属している必要があります。
2. 項目の IMAGE-INDEX 属性をイメージリスト内のイメージの順序番号（1：最初のイメージリストコントロール、2：2 番目のイメージ、など）に設定する。 この場合、項目の IMAGE 属性は、指定しないか、またはデフォルト値の NULL-HANDLE に設定する必要があります。

最初の方法（相対インデックス）では、インデックスにラップアラウンドが使用されます。 したがって、イメージコントロールにN個のイメージが含まれているとすると、イメージインデックス 0 はイメージコントロールの最初のイメージを指し、イメージインデックス N - 1 は最後のイメージを指します。そして、イメージインデックス N はまた最初のイメージを指す、のように続きます。そのため、イメージコントロールにイメージが1つしか含まれていない場合、ラップアラウンドのために、イメージの相対インデックスは（指定しても）まったく効果がありません。常に最初（で唯一）のイメージが使用されます。

2つ目の方法（絶対インデックス）では、イメージインデックスにラップアラウンドは使用されません。この場合、インデックスには1からイメージリスト内のイメージ数（この値を含む）までの範囲の値を指定する必要があります。 指定した値がこの範囲にない場合、その項目にイメージは表示されません。

IMAGE-INDEX 属性はイメージコントロールにも適用できることに注意してください。 この場合、この属性は読み取り専用で、親イメージリストコントロール内のイメージコントロールの最初のイメージのオフセット（ゼロから始まる）を返します。

相対インデックスを使用する長所には、指定したイメージへの参照がNaturalによって追跡され（ダイアログエディタ内またはランタイム時）、イメージコントロールまたはイメージリストの位置に対する変更が自動的に反映される点があります。 ただし、実際には、イメージリストコントロールで単一の合成イメージ（つまり、複数イメージ）コントロールが使用され、かつ、ランタイム時にイメージが変更されないような場合は、絶対インデックスの使用がおそらく最も便利です。

オーバーレイイメージ

1 つのイメージのバリエーションを複数提供できることが望ましい場合があります。 例えば、フォルダを表す項目に対して表示するイメージでは、フォルダがアクティブであることを示すためにイメージの変更が必要な場合があります。 フォルダのイメージとアクティブなフォルダのイメージを用意するのではなく、最初のフォルダのイメージのみを用意し、アクティブな状態を示すときは 「アクティブな」 シンボルのみを含む 2 つ目のイメージを最初のイメージに重ね合わせる方が便利です。 このようなイメージは、*基本*イメージと区別するために*オーバーレイ*イメージと呼ばれます。

オーバーレイイメージは、ホストコントロールの IMAGE-LIST 属性で指定されている、基本イメージの表示に使用するイメージリストと同じイメージリスト内に格納します。 したがって、オーバーレイイメージのサイズは基本イメージと同じですが、下にあるイメージが見えるように、常に透過的にレンダリングします。

項目でオーバーレイイメージを使用するには、項目の OVERLAY 属性および／または OVERLAY-INDEX 属性に値を指定する必要があります。 これらの属性は、それぞれ基本イメージの IMAGE 属性および IMAGE-INDEX 属性と同様に使用されます（上記参照）。

技術的な理由により、オーバーレイイメージとして使用するイメージは 「事前登録」 する必要があります。 Natural では、イメージリストコントロールの "O"（オーバーレイ）STYLE を設定すると、事前登録されます。ただし、オーバーレイコントロールをスタティックに定義すると、このスタイルはダイアログエディタによって自動的に設定されます。このスタイルの有無によって、基本イメージとオーバーレイイメージは区別されます。 したがって、OVERLAY 属性を指定すると、このスタイルを持つイメージコントロールのみを参照できます。一方、IMAGE 属性を指定すると、このスタイルを持たないイメージコントロールのみを参照できます。 絶対インデックス（上記参照）が使用されている場合、IMAGE-INDEX はオーバーレイイメージを参照できます（基本イメージとしては「誤用」です）。 ただし、同様に OVERLAY-INDEX 属性を使用して基本イメージを参照しようとすると、エラーになります（オーバーレイイメージは表示されません）。

イメージリストに定義できるオーバーレイイメージの数は、Windows によって制限が設定されています。 この制限数は現時点では 15 です。 合成オーバーレイイメージコントロールを使用する場合、合成ビットマップ内のサブイメージはこれとは別にカウントされます。

例として、以下のように、個別のオーバーレイイメージを含む合成イメージに基づいたイメージコントロールを作成します。

```
PROCESS GUI ACTION ADD WITH
PARAMETERS
   HANDLE-VARIABLE = #IMG-2
   TYPE = IMAGE
   PARENT = #IMGLST-1
   STYLE = 'COs'
   BITMAP-FILE-NAME = 'overlays.bmp'
END-PARAMETERS GIVING *ERROR
```

その後、以下のコードを実行すると、（例えば）合成ビットマップの 2 番目のオーバーレイイメージを使用して、リストビュー項目を作成できます。

```
PROCESS GUI ACTION ADD WITH
PARAMETERS
   TYPE = LISTVIEWITEM
   PARENT = #LV-1
   STRING = 'Item with overlay'
   IMAGE = #IMG-1
   IMAGE-INDEX = 3
   OVERLAY = #IMG-2
   OVERLAY-INDEX = 1
END-PARAMETERS GIVING *ERROR
```

上記の例のリストビュー項目では、基本イメージとして *COMPOSITE.BMP* の 4 番目のイメージが、オーバーレイイメージとして *OVERLAYS.BMP* の 2 番目のイメージが使用されます（すでに説明したとおり、イメージの相対インデックスはゼロから始まります）。 リストビュー項目は匿名で（つまり、明示的に HANDLE-VARIABLE 属性値を指定しないで）作成されることに注意してください。

イメージの変更

使用中であっても、イメージコントロールを変更できます。 次に例を示します。

```
#IMG-1.BITMAP-FILE-NAME := 'new.bmp'
```

変更されたイメージコントロールのイメージを明示的に（つまり、相対インデックスを使用して）参照する各項目は、Naturalによって追跡され、自動的に更新および再描画されます。 ただし、絶対インデックスが使用されている場合、変更されたイメージコントロールのイメージを暗黙的に参照していても、対応する項目は更新されません。

イメージの削除

DELETE アクションを使用してイメージコントロール全体を削除することにより、イメージリストからイメージを削除できます。 次に例を示します。

```
PROCESS GUI ACTION DELETE WITH #IMG-1 GIVING *ERROR
```

削除されたイメージコントロールのイメージを*明示的に*（つまり、相対インデックスを使用して）参照している項目はすべて、イメージがないことを表すために自動的に更新および再描画されます。

ただし、絶対インデックスが使用されている場合、自動更新は行われません。 例えば、3つの単一イメージコントロールを含むイメージリストコントロールと、これらの3つのイメージを絶対インデックスを使用して参照する項目があるとします。2番目のイメージコントロールを削除すると、2 番目のイメージを参照していた項目が 3 番目のイメージを参照するようになり、3 番目のイメージを参照していた項目は参照範囲から「はみでて」、何も参照しなくなります。 なお、これらの項目を含むコントロールでは、変更を反映するための再描画は自動的には行われません。

また、DELETE-CHILDREN アクションを使用してイメージリストのイメージを一度にすべて削除することもできます。

```
PROCESS GUI ACTION DELETE-CHILDREN WITH #IMGLST-1 GIVING *ERROR
```

これは、イメージリストの各イメージを個別に削除するのと同じことです。

合成イメージ（つまり、複数イメージ）コントロール内のイメージは個別に削除できないことに注意してください。

イメージリストコントロールの削除

イメージリストコントロールは、使用中であっても、必要がなくなれば削除できます。 次に例を示します。

```
PROCESS GUI ACTION DELETE WITH #IMGLST-1 GIVING *ERROR
```

イメージリストコントロールを使用していたコントロールはすべて適切に更新されます。これらのコントロールの IMAGE-LIST 属性は自動的に NULL-HANDLE にリセットされます。

98 リストボックスコントロールおよび選択ボックスコントロールの操作

リストボックスコントロールおよび選択ボックスコントロールには、多くの項目が含まれます。コントロールと項目の両方はダイアログエレメントであり、コントロールは項目の親になります。

リストボックス項目および選択ボックス項目を作成するには、以下の 2 つの方法があります。

- Naturalコードを使用して、複数のリストボックス項目をそれぞれダイナミックに作成する。
- ダイアログエディタを使用して、1つまたは配列形式のリストボックス項目および選択ボックス項目を追加する。

Natural コードでは、以下のように記述します。

```
#AMOUNT := 5
ITEM (1) := 'BERLIN'
ITEM (2) := 'PARIS'
ITEM (3) := 'LONDON'
ITEM (4) := 'MILAN'
ITEM (5) := 'MADRID'
PROCESS GUI ACTION ADD-ITEMS WITH #LB-1 #AMOUNT #ITEM (1:5) GIVING #RESPONSE
```

最初に、作成する項目数を指定し、次に項目値を指定します。そして、PROCESS GUI ステートメントの ADD-ITEMS アクションを使用します。

リストボックスコントロールのどの項目が選択されたかを把握するためにすべての項目を調べる場合は、SELECTED-SUCCESSOR 属性を使用することをお勧めします。この方法は、リストボックスコントロールが非常に多くの項目（例：100）を含んでいる場合に、パフォーマンスの向上に有効です。 SELECTED-SUCCESSOR を使用すると、SELECTED 属性および SUCCESSOR 属性を使用すれば、個別に 100 回クエリする代わりに、1 回クエリするだけで確認できます。

例：

```
/* Displays the STRING attribute of every SELECTED list-box item
MOVE #LISTBOX.SELECTED-SUCCESSOR TO #LBITEM
REPEAT UNTIL #LBITEM = NULL-HANDLE
   .../* STRING display logic

    MOVE #LBITEM.SELECTED-SUCCESSOR TO #LBITEM
END-REPEAT
```

SELECTED-SUCCESSOR 属性を使用して同じダイアログエレメントのハンドルを 2 回参照すると、Natural は項目ハンドルのリストを 2 回調べるため、パフォーマンスが低下します。

```
/* Displays the STRING attribute of every SELECTED list-box item,
/* but may be slow
MOVE #LISTBOX.SELECTED-SUCCESSOR TO #LBITEM
REPEAT UNTIL #LBITEM = NULL-HANDLE
   IF #LBITEM.SELECTED-SUCCESSOR = NULL-HANDLE /* Searches in the list of items
     IGNORE
   END-IF
   .../* STRING display logic
   MOVE #LBITEM.SELECTED-SUCCESSOR TO #LBITEM /* Searches in the list of items
END-REPEAT                                    /* for the second time
```

この問題を回避するには、#LBITEM の他に 2 つ目の変数 #OLDITEM を使用します。

```
/* Displays the STRING attribute of every SELECTED list-box item
MOVE #LISTBOX.SELECTED-SUCCESSOR TO #LBITEM
REPEAT UNTIL #LBITEM = NULL-HANDLE
   #OLDITEM = #LBITEM
   #LBITEM = #LBITEM.SELECTED-SUCCESSOR/* Searches in the list of items (once)
   IF #LBITEM = NULL-HANDLE
     IGNORE
  END-IF
```

```
   .../* Display logic using #OLDITEM.STRING
END-REPEAT
```

選択した項目のハンドル値を受け取る場合、通常、選択した項目によって NULL-HANDLE 以外の値が返されます。 以下の例に示すように、選択していない項目の SELECTED-SUCCESSOR 値を取得する直前に SELECTED-SUCCESSOR に値を割り当てると、選択していない項目からもこのようなハンドル値が返されます。

```
...
PTR := #LB-1.SELECTED-SUCCESSOR
PTR := NOT_SELECTEDHANDLE.SELECTED-SUCCESSOR
IF NOT_SELECTEDHANDLE.SELECTED-SUCCESSOR = NULL-HANDLE THEN
   #DLG$WINDOW.STATUS-TEXT := 'NULL-HANDLE'
ELSE
   COMPRESS 'NEXT SELECTION: ' PTR.STRING TO #DLG$WINDOW.STATUS-TEXT
END-IF
...
```

リストボックスコントロール内の特定の項目が選択されているかどうかをクエリする場合、SELECTED 属性を使用することによって最高のパフォーマンスを得られます。

```
#DLG$WINDOW.STRING:= #LB-1-ITEMS.SELECTED(3)
```

選択ボックスコントロールおよび入力フィールドコントロールの保護

エンドユーザーによって選択ボックスコントロールまたは入力フィールドコントロールにデータが入力されないようにするには、以下のようないくつかの方法があります。

- ダイアログエレメントの MODIFIABLE 属性を FALSE に設定する。
- セッションパラメータ AD=P を設定する。
- 制御変数（CV）を使用する。

選択ボックスコントロールが保護されていても、項目を選択できます。ただし、項目リストの値が入力フィールドに表示されるだけです。 STRING 属性を項目リストにない値に（ダイナミックにまたは初期化で）設定すると、その値はエンドユーザーに表示されません。

99 リストビューコントロールの操作

このchapterでは、次のトピックについて説明します。

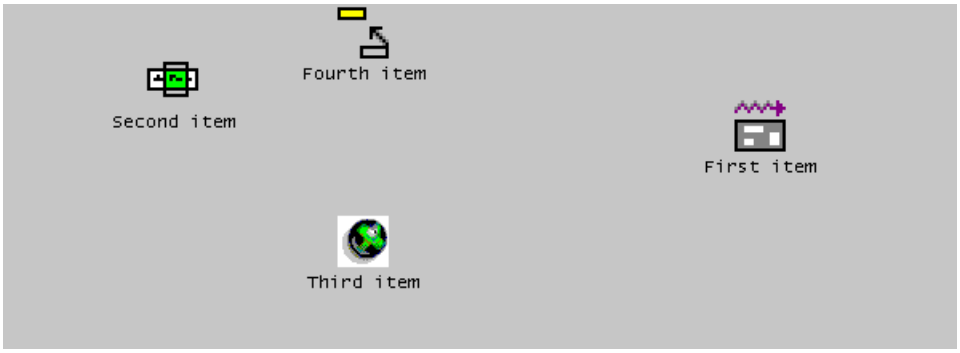
はじめに

リストビューコントロールは、アイコンまたは列を基準にした形式でデータを表示するために使用できます。 リストビューコントロールは、非常に強力なコントロールです。 ただし、タブ形式でデータを表示し、任意の列の値をその場所で直接編集する場合は、リストビューコントロールではなくテーブルコントロールの使用を検討する必要があります。

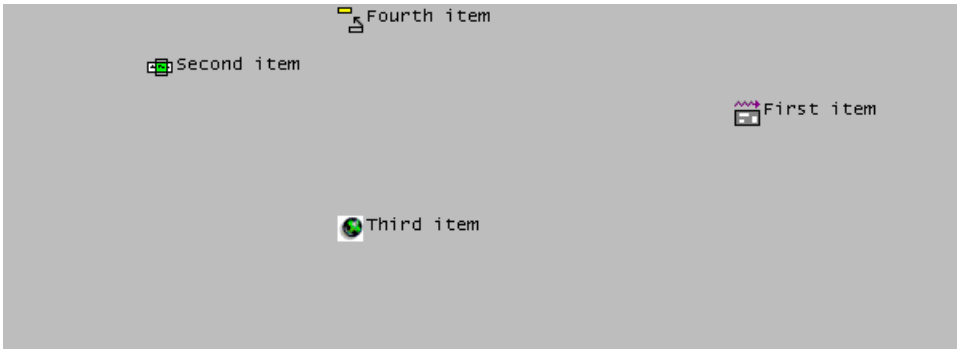
ビューモード

Naturalのリストビューコントロールは、アイコン、小さいアイコン、リスト、レポートの4つのビューモードでデータを表示できます。

アイコンビューモードでは、データは大きいアイコン形式で表示されます。

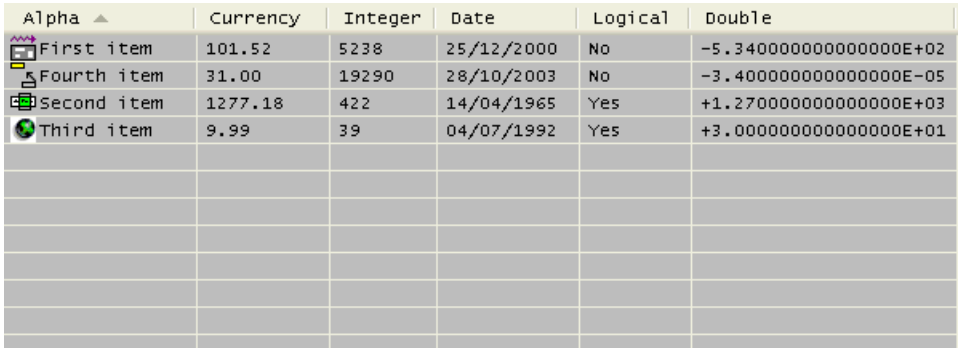


小さいアイコンのビューモードでは、項目ラベルがアイコンと一緒に表示されます。 アイコンビューでは、項目を任意の位置に表示できます。



リストビューモードでは、小さいアイコンのビューモードと同じように項目が表示されますが、自由に配置することはできません。 代わりに、整列して表示されます。

レポートビューモードでは、項目ごとに 1 行が割り当てられ、項目に関連付けられている他のデータが同じ行の個別の列に表示されます。 通常は、以下の例のように、列ヘッダーも表示されます（リストビューコントロールの［ヘッダーなし（x）］ STYLE フラグを設定することにより、任意で非表示にできます）。



レポートビューでは、列の仕切りをドラッグすることにより、列のサイズを変更できます。 代わりに、列ヘッダー内の列の右側の区切りをダブルクリックすると、列内で最も長いテキストに合わせて列の幅が調整されます。

上記の例は、リストビューコントロール自体、4 個のリストビュー項目、および 6 個のリストビュー列の合計 11 個のダイアログエレメントから構成されていることに注意してください。 リストビュー項目とリストビュー列の両方とも、PARENT としてリストビューコントロールを持ち、同じ SUCCESSOR チェーンに格納されます。 リストビュー項目とリストビュー列を混在させることは可能ですが、構成とパフォーマンスの両方の観点から、すべてのリストビュー列がすべてのリストビュー項目の前に来るように設定することをお勧めします。 例えば、空でないリストビューコントロールに最後の列として新しい列をダイナミックに挿入する場合、列の SUCCESSOR 属性に何も指定しないのではなく、最初のリストビュー項目のハンドルを明示的に設定するか、または新しい列がチェーンの最後、つまりすべてのリストビュー項目の後に配置される NULL-HANDLE を設定します。

リストビューコントロールに対して作成される最初のリストビュー列には特別な意味があります。この列を（ここでは）*プライマリ列*と呼びます。 プライマリ列には常にリストビュー項目ラベル（つまり、STRING 属性値）が表示されます。 その他の列には、*サブ項目データ*と呼ばれるものが表示されます。 例えば、「Currency サブ項目」は "Currency" 列に格納されているデータを参照します（上記の例を参照）。 列内の特定の値を参照するには、さらに厳密に指定する必要があります。 例えば、上記の "1277.18" という値は、"Second item" 項目の "Currency サブ項目" として参照されています。 サブ項目は、Natural のダイアログエレメントのタイプではありません。これについては、後で説明します。

リストビュー列が作成されていない場合、レポートビューモードではリストビューコントロールに情報が表示されません。 ただし、表示モードを切り替える唯一の方法は、プログラムでリストビューコントロールの VIEW-MODE 属性値を明示的に変更することです。 したがって、アプリケーションで複数の表示モードをサポートする場合、表示モードを切り替えるメカニズム（コンテキストメニューなど）を用意する必要があります。実際にはこのような場合、アプリケーションがレポートビューモードへの切り替えを認めないため、通常はユーザーが列のないリストビューコントロールをレポートビューモードで見ることはありません。

項目イメージの設定

リストビュー項目のイメージは、「*イメージリストコントロールの操作*」で説明しているように、リストビューコントロールにイメージリストコントロールを作成および関連付け、（項目ごとに）必要なイメージを、インデックスおよび／またはイメージハンドルを使用してイメージリストから選択することによって定義できます。

サポートする表示モードに合わせて、イメージリストコントロールの［大きいイメージ（L）］スタイルおよび［小さいイメージ（S）］スタイルを設定する必要があることに注意してください。 アイコンビューモードでは、大きいアイコンを表示できる必要があります。一方、その他の表示モードでは、小さいアイコンを表示できる必要があります。

項目の配置

アイコンビューモードおよび小さいアイコンのビューモードでは、RECTANGLE-X および／または RECTANGLE-Y の各属性値を設定することによって、リストビュー項目を（再）配置できます。項目の作成時に位置が明示的に指定されなかった場合、項目はデフォルトのグリッド間隔を持つ仮想グリッド上に配置されます。デフォルトのグリッド間隔は、リストビューコントロールの SPACING-X および SPACING-Y の各属性値を設定することにより上書きできます。 ARRANGE アクションを使用するといつでも、論理グリッドを基準にした連続する位置に項目を再整列させたり、最も近くの論理グリッド位置に項目の位置を合わせたりできます。

2つのアイコンビューモードでは、リストビュー項目の位置は、コントロールのクライアントエリアに対する相対位置（他の表示モードの場合に使用）ではなく、ビュー座標として解釈されることに注意してください。 クライアント座標とは異なり、アイコンビューがスクロールされて

も、項目のビュー座標は変わりません。 ビュー座標とクライアント座標を変換するには、リストビューコントロールの OFFSET-X 属性および OFFSET-Y 属性を使用する必要があります。これらの属性は、クライアントエリアの基点をビュー座標で返します。

2つのリストビューコントロールの STYLE フラグを使用すると、プログラムによって明示的に指定された位置を上書きできることに注意してください。 1 つ目は［自動整列（a）］スタイルフラグで、このスタイルフラグを指定すると、項目が追加または移動されるたびに、仮想グリッド上で項目が自動的に再整列されます。 この場合、明示的に指定された位置は、整列したアイコンリスト内の項目の位置を間接的に指定するにすぎません。 2 つ目は［グリッドに合わせる（r）］スタイルフラグで、このスタイルフラグを指定すると、プログラムによって明示的に指定された項目の位置は、仮想グリッド上の最も近くの整列位置に調整されます。［自動整列（a）］スタイルが設定されている場合、このスタイルは不要であることに注意してください。

ユーザーはリストビュー項目の位置をドラッグ&ドロップで変更できることに慣れているため、コントロールでもこの機能が自動的に提供されるものと期待している可能性があります。 ただし、この場合は当てはまりません。 アプリケーションでドラッグ & ドロップをサポートする場合は、明示的に処理する必要があります。これについては、次のセクションで説明します。

リストビューモードでは、すでに説明したように、項目は常に列内に表示され、再配置できません。 ただし、隣接する列の間隔は、SPACING 属性を使用して設定できます。

リストビューコントロールでは、表示モードを切り替えるときに項目の位置が記憶されないことに注意してください。 例えば、アイコンビューモードの 1 つから別のモードに切り替え、再度アイコンビューモードに戻ると、アイコンは常に整列しています。 この動作は、以下の例のように、項目の位置を保存およびリストアするプログラムコードを明示的に記述することにより回避できます。

```
DEFINE SUBROUTINE SAVE-ITEM-POSITIONS
   #ITEM := #CONTROL.FIRST-CHILD
   REPEAT WHILE #ITEM <> NULL-HANDLE
     IF #ITEM.TYPE = LISTVIEWITEM
       #ITEM.CLIENT-KEY := 'RECTANGLE-X'
       #ITEM.CLIENT-VALUE := #ITEM.RECTANGLE-X
       #ITEM.CLIENT-KEY := 'RECTANGLE-Y'
       #ITEM.CLIENT-VALUE := #ITEM.RECTANGLE-Y
     END-IF
     #ITEM := #ITEM.SUCCESSOR
   END-REPEAT
END-SUBROUTINE
*
DEFINE SUBROUTINE RESTORE-ITEM-POSITIONS
   #ITEM := #CONTROL.FIRST-CHILD
   REPEAT WHILE #ITEM <> NULL-HANDLE
     IF #ITEM.TYPE = LISTVIEWITEM
       #ITEM.CLIENT-KEY := 'RECTANGLE-X'
       IF #ITEM.CLIENT-VALUE <> ' '
       #ITEM.RECTANGLE-X := VAL(#ITEM.CLIENT-VALUE)
```

```
      END-IF
      #ITEM.CLIENT-KEY := 'RECTANGLE-Y'
      IF #ITEM.CLIENT-VALUE <> ' '
        #ITEM.RECTANGLE-Y := VAL(#ITEM.CLIENT-VALUE)
      END-IF
    END-IF
    #ITEM := #ITEM.SUCCESSOR
  END-REPEAT
END-SUBROUTINE
*
DEFINE SUBROUTINE SWITCH-VIEW-MODE
  IF #VIEW-MODE <> #CONTROL.VIEW-MODE
    IF #CONTROL.VIEW-MODE = VM-ICON OR
       #CONTROL.VIEW-MODE = VM-SMALLICON
      PERFORM SAVE-ITEM-POSITIONS
    END-IF
    #CONTROL.VIEW-MODE := #VIEW-MODE
    IF #VIEW-MODE = VM-ICON OR
       #VIEW-MODE = VM-SMALLICON
      PERFORM RESTORE-ITEM-POSITIONS
    END-IF
  END-IF
END-SUBROUTINE
```

ローカルデータの定義は以下のように想定されています。

```
01 #CONTROL HANDLE OF GUI
01 #ITEM HANDLE OF GUI
01 #VIEW-MODE (I4)
```

実際の表示モードの切り替えは、#VIEW-MODE を必要な表示モード（ローカルデータエリア NGULKEY1 で定義されている VM-* 定数）に、#CONTROL をリストビューコントロールのハンドルに設定し、SWITCH-VIEW-MODE サブルーチンを呼び出すことにより、実行できます。

項目の選択

項目を選択するには、項目をクリックするか（Ctrl キーを押したままクリックすると拡張選択を実行可能）、またはリストビューコントロール内をクリックし、マウスの第1ボタンを押したままドラッグして選択リージョンを指定します。 後者の手法は Marquee 選択と呼ばれ、コントロールの［Marquee 選択（m）］STYLE フラグが設定されている場合にのみ使用できます（デフォルトで設定）。 コントロールの［ホットトラック選択（t）］スタイルフラグが設定されている場合、項目を選択するのにクリックは必要ないことに注意してください。 マウスカーソルを項目の上にしばらく置いておくだけで選択されます。

また、プログラムで項目の SELECTED 属性を設定したりクリアしたりすることにより、項目を選択したり選択解除したりできます。

どちらの場合でも、コントロールの MULTI-SELECTION 属性が TRUE に設定されている場合は、拡張選択を実行できます。 拡張選択とは、既存の選択をクリアすることなく、選択項目を追加したり、選択済みの項目を選択解除したりするプロセスのことです。これにより、複数の項目を選択できます（まったく選択しないことも可能）。 単一選択リストビューの場合、新しい項目を選択するとそれまで選択されていた項目が暗黙的に選択解除されるという方法のみを使用できます。 この場合、Marquee 選択も使用できません。

最初に（または唯一）選択された項目は（存在する場合）、リストビューコントロールの SELECTED-SUCCESSOR 属性をクエリすると確認できます。何も項目が選択されていない場合は、NULL-HANDLE が返されます。 次に選択されている項目は（存在する場合）、選択している項目の SELECTED-SUCCESSOR 属性をクエリすると確認できます。 アプリケーションでこの手法を繰り返し使用すると、以下の「ドロップアンドドラッグ」で説明されているように、選択されているすべての項目を列挙できます。

項目が選択または選択解除されるたびに、リストビューコントロールの CLICK イベントが発生します（抑制されていない場合）。このとき、対応する項目のハンドルがコントロールの ITEM 属性に設定されます。ごく短時間に連続して多数の項目が選択される場合があるため（Marquee 選択など）、このイベントには時間のかかる処理を記述しないようにします。 例えば、CLICK イベントハンドラでは単に論理変数を TRUE に設定してさらに処理を実行する必要があることを指定するのみにとどめ、実際の処理はダイアログの IDLE イベントハンドラでこのフラグの設定に応じて実行するようにします。 処理が終了したら、このフラグをクリアすることを忘れないようにしてください。

リストビューコントロールの［チェックボックス（c）］スタイルフラグが設定されている場合、各項目と一緒にチェックボックスが表示されます。 項目の CHECKED 属性は、プログラムで項目のチェックステータスを取得および設定するために使用できます。 チェックされている最初の項目は（存在する場合）、リストビューコントロールの CHECKED-SUCCESSOR 属性をクエリすると確認できます。また、チェックされている項目のこの属性をクエリすると、次にチェックされている項目のハンドルが返されます（存在する場合）。これにより、以下に示す、チェックされている項目数をカウントする簡単な例のように、チェックされているすべての項目を列挙できます。

```
RESET #COUNT
#ITEM := #LV-1.CHECKED-SUCCESSOR
REPEAT WHILE #ITEM <> NULL-HANDLE
   ADD 1 TO #COUNT
```

```
   #ITEM := #ITEM.CHECKED-SUCCESSOR
END-REPEAT
```

ローカルデータの定義は以下のように想定されています。

```

01 #LV-1 HANDLE OF LISTVIEW
01 #ITEM HANDLE OF LISTVIEWITEM
01 #COUNT (I4)
```

項目が選択または選択解除されるたびに、SUPPRESS-CHECK-EVENT 属性で抑制されていなければ、リストビューコントロールの CHECK イベントが発生します。このとき、対応する項目のハンドルがコントロールの ITEM 属性に設定されます。

項目の有効化

項目をダブルクリックすると、リストビューコントロールの ACTIVATE イベントが発生します（抑制されていない場合）。 このイベントをアプリケーションで制御する場合、通常は選択された各コントロール上でデフォルトのアクションを実行します。デフォルトのアクションはユーザー定義で、項目ごとに異なる処理を設定できます。 例えば、テキストファイルを表す項目を有効化すると、エディタ上でそのファイルが開かれます。一方、オーディオファイルを表す項目を有効化すると、そのファイルが再生されます。1つ以上の選択された項目に対し、デフォルトでない他のアクションも使用できることに注意してください。ただし、これらのアクションは通常、他のメカニズムによってアクセスされます。 例えば、これらはアプリケーションが表示するコンテキストメニューに（通常、デフォルトのアクションと一緒に）リストされます。

複数選択が可能な場合に（上記参照）Ctrl キーを押したまま項目をダブルクリックすると、ACTIVATE イベントが発生する前にその項目の選択状態が切り替えられます。

コントロールの［下線ホット（u）］または［下線コールド（U）］の各 STYLE フラグのどちらかが設定されている場合、項目を 1 回クリックするだけでその項目が有効化されます。

また、ACTIVATE イベントは、キーボードを使用してトリガすることもできます。 このためには、次の 2 つの方法のいずれかを使用します。

1. リストビューコントロールの ACCELERATOR 属性に定義されているキーまたはキーの組み合わせを押す。 現時点で、コントロールにフォーカスがなくてもかまいません。
2. リストビューコントロールにフォーカスがある場合、Enter キーを押す。 この方法は、ダイアログにデフォルトのプッシュボタンが含まれておらず、プッシュボタンに［OK ボタン (O)］の STYLE フラグが設定されていない場合にのみ機能します。

どちらの場合も、項目が選択されていないと ACTIVATE イベントは発生しません。

リストビュー列とサブ項目

リストビューの各列（リストビューコントロールがレポートビューモードのときに表示）は、リストビュー項目ごとに（最大）1つの項目データを持ちます。このデータは（存在する場合）、そのリストビュー項目の該当する列に表示されます。前述のとおり、このデータをサブ項目データと呼びます。

適切にソートできるようにするために（次のセクションを参照）サブ項目データを英数字にする必要はありません。デフォルトは英数字ですが、列の FORMAT 属性でサポートされている定義済みのタイプを使用できます。 また、列の EDIT-MASK 属性を設定することによって、列に編集マスクを適用できます。 ユーザーに表示されるレポートビュー列の値は、関連付けられている編集マスクが適用された（存在する場合）、その列に対するサブ項目の英数字表記になります。編集マスクが使用されている場合、サブ項目データと表示データ間の変換は、MOVE EDITED ステートメントと互換性があります。 それ以外の場合、内部データと表示されたデータ間およびその逆の変換は、データを Natural スタックにコピーしたり、Natural スタックからコピーする場合に発生する変換と互換性があります（STACK TOP DATA および INPUT ステートメントを参照）。 例えば、数値は、現在の小数点文字を使用して（DC パラメータで定義されたように）表示されます。必要に応じて、負の場合は先頭にマイナスを付けることができます。日付値は、DTFORM パラメータで定義されたフォーマットで表示されます。論理値は、真の場合は "X" として、偽の場合は空白として表示されます。

リストビューコントロールの最初に定義された列（*プライマリ列*）には、常に項目のラベルが表示されるという、特別な意味があります。 したがって、最初の列に対する項目のサブ項目が変更されると、自動的に項目のラベルが変更されます。その逆もまた同様です。 その他のすべての列の場合、サブ項目データを更新する唯一の方法は SET-SUBITEM-DATA アクションを使用することです。 このアクションを呼び出す場合、サブ項目データは、列の FORMAT 属性値（デフォルトでは英数字）で指定されている*内部*データタイプと互換性のあるフォーマットで受け渡す必要があります。

例えば、以下の例のように、リストビューコントロールに列を追加するとします。

```
PROCESS GUI ACTION ADD WITH
PARAMETERS
   HANDLE-VARIABLE = #LVCOL-DATE
   TYPE = LISTVIEWCOLUMN
   STRING = 'Date'
   PARENT = #LV-1
   RECTANGLE-W = 83
   STYLE = '1'
   FORMAT = FT-DATE
```

```
   EDIT-MASK = 'YYYY/MM/DD'
END-PARAMETERS GIVING *ERROR
```

特定のリストビュー項目#LVITEM-1（例）に対する列のサブ項目データを、以下のように現在日付に設定します。

```
PROCESS GUI ACTION SET-SUBITEM-DATA WITH
  #LVITEM-1 #LVCOL-DATE *DATX GIVING *ERROR
```

Naturalの時刻値は自動的に日付値に変換されるため、*DATXの代わりに*TIMXも使用できることに注意してください。 どちらの場合も、YYYY/MM/DDフォーマットの現在日付として値が表示されます。 プライマリ列の場合、表示される文字列は項目のラベルです。つまり、リストビューコントロールがレポートビューモードでなくても、その変更を見ることができます。

データは表示フォーマットでは渡せないことにも注意してください。 例えば、以下の例は動作しません。

```
PROCESS GUI ACTION SET-SUBITEM-DATA WITH
  #LVITEM-1 #LVCOL-DATE '2004/11/03' GIVING *ERROR  /* Does NOT work!
```

ただし、列がプライマリ列の場合は、項目のラベルを設定することにより、間接的に列データを更新できます。 次に例を示します。

```
#LVITEM-1.STRING := '2004/11/03'
```

サブ項目データの取得は、SET-SUBITEM-DATAアクションと同じパラメータを受け渡すGET-SUBITEM-DATAアクションを呼び出すことによって実現できます。 次に例を示します。

```
PROCESS GUI ACTION GET-SUBITEM-DATA WITH
  #LVITEM-1 #LVCOL-DATE #DATE GIVING *ERROR
```

#DATEは、以下のように定義されています。

```
01 #DATE (D)
```

取得するサブ項目データがない場合があるという事実に注意してください。 例えば、サブ項目データがまだ作成されていない、または削除されている場合（下記参照）があります。ただし、Naturalでは、空値はサポートされていません。したがって、デフォルトでは、空値が返されるとNaturalによって受け取りフィールドはリセットされます（RESETステートメントを参照）ただし、リストビュー列にデフォルト値が設定されている場合、このデフォルト値が代わりに返されます。 列のデフォルト値を設定するには、リストボックス項目のハンドルの代わりに

NULL-HANDLE を指定して SET-SUBITEM-DATA を呼び出します。 例えば、正数のみを使用できる数値列 #LVCOL-NUM に対し、以下のようにデフォルト値を -1 に設定しようとする場合があります。

```
#NUM := -1
PROCESS GUI ACTION SET-SUBITEM-DATA WITH
  NULL-HANDLE #LVCOL-NUM #NUM GIVING *ERROR
```

#NUM は、任意の符号付き数値フォーマット（I2 など）です。

デフォルト値を使用すると、プログラムで使用できない値を指定できます。 必要に応じて、デフォルト値に明示的に値が入力されないように、プログラムを変更する必要があります。

SET-SUBITEM-DATA および GET-SUBITEM-DATA の両アクションでは、単一のステートメントで（特定の項目に対する）複数列のサブ項目データをそれぞれ設定または取得できます。 次に例を示します。

```
PROCESS GUI ACTION SET-SUBITEM-DATA WITH
  #LVITEM-1 #LVCOL-NUM #NUM #LVCOL-DATE #DATE GIVING *ERROR
```

つまり、オペランドのペア [列ハンドル, 受け取りフィールド] を複数指定できます。

サブ項目データを削除するには、DELETE-SUBITEM-DATA アクションを使用します（これにより、データが設定されていなかったかのように内部的に空値が格納されます）。 次に例を示します。

```
PROCESS GUI ACTION DELETE-SUBITEM-DATA WITH
  #LVITEM-1 #LVCOL-DATE GIVING *ERROR
```

ここでもまた、複数の列ハンドルを指定することにより、特定の項目に対する複数のサブ項目を単一のステートメントで削除できます。

```
PROCESS GUI ACTION DELETE-SUBITEM-DATA WITH
  #LVITEM-1 #LVCOL-DATE #LVCOL-NUM GIVING *ERROR
```

リストビュー項目が削除されると、その項目に関連付けられていたサブ項目データ（存在する場合）も一緒に削除されます。 リストビュー列を残したままリストビューコントロール内のすべての項目を削除する場合、CLEAR アクションを使用します。

```
PROCESS GUI ACTION CLEAR WITH #LV-1 GIVING *ERROR
```

#LV-1 は、リストビューコントロールのハンドルです。

ソート

リストビュー列のサブ項目データのソートは、ユーザーが列ヘッダーをクリックするか（リストビューコントロールの［ヘッダーなし（x）］および［ソートヘッダーなし（y）］の各 STYLE フラグが設定されていない場合）、またはプログラムで SORT-ITEMS アクションを呼び出すことによって実現できます。 SORT-ITEMS アクションを使用する場合、リストビュー列ではなくリストビューコントロール自体のハンドルを渡すことによって、有効な列がない場合でも項目がソートされます。 詳細については、SORT-ITEMS アクションのドキュメントを参照してください。

リストビューコントロールの列ヘッダー内の列をクリックすることによるソートは通常、Natural によって暗黙的に実行されます。ただし、ソートが実行される前に、Natural によってそのリストビュー列の CLICK イベントが起動されます（抑制されていない場合）。 このイベントから復帰すると、列がすでに必要な方向にソートされているかどうかが Natural によってチェックされます。すでにソート済みの場合、これ以上アクションは実行されません。 つまり、ソート方向のルールに従っている限り（つまり、列が昇順でソートされている場合に降順を指定する、またはその逆）、Natural の代わりにアプリケーションでソートを実行できます。ソートオプション（大文字と小文字の区別など）をダイナミックに指定する必要がある場合や、ソート後にアプリケーション固有のコードを実行する必要がある場合、これは便利です。 明示的にソートを実行する、列の CLICK イベントハンドラ例を以下に示します。

```
#CONTROL := *CONTROL
T1. SETTIME
IF #CONTROL.SORTED AND NOT #CONTROL.DESCENDING
  PROCESS GUI ACTION SORT-ITEMS WITH #CONTROL TRUE
    GIVING *ERROR
ELSE
  PROCESS GUI ACTION SORT-ITEMS WITH #CONTROL
    GIVING *ERROR
END-IF
```

```
COMPRESS 'Sort took' *TIMD(T1.) 'tenths of a second'
  INTO #DLG$WINDOW.STATUS-TEXT
```

ただし、ほとんどの場合は、Natural の暗黙的なソートを使用すれば十分です。

英数字データの場合、ソート列の［大文字／小文字の区別なし（i）］および［単語比較（w）］の各STYLEフラグによって、値を比較する方法が変わります。 ソートを明示的に実行する場合、対応するオプションパラメータをSORT-ITEMSアクションに指定すると、これらのデフォルト値は上書きされます。 これらのオプションの詳細については、このアクションのドキュメントを参照してください。

欠損値（「空値」）は、低く評価されます。 つまり、この値は、列を降順でソートすると列の下部に、昇順でソートすると列の上部に表示されます。 さらに、2 つの列エントリが同じ場合は、該当する 2 つの項目のそれまでの相対位置が保持されます。

リストビューコントロールがアイコンビューモードの1つを使用している場合、項目をソートすると項目が再整列されることに注意してください。 したがって、アイコンビューモードのどちらかで明示的に項目の位置を指定している場合、ソートコマンドを無効にすることをお勧めします。

また、リストビューコントロールもSORTED属性を持っています。これにより、新しい項目は、項目リストの最後に追加されるのではなく、コントロールのDESCENDING属性の値に応じて昇順または降順のソート位置に挿入されます。 この機能を期待どおりに動作させるには、あらかじめ適切な方向で項目がソートされている必要があります。 例えば、項目のラベル（つまり、項目のSTRING属性）が変更された場合、必要に応じて、ソートされた順序をリストが維持していることをアプリケーション自体で確認する必要があります。 これを実行する方法の例については、次のセクションで紹介します。

SORTED属性は、アイコンビューモードのどちらかで表示されている項目の位置には影響しないことに注意してください。 ただし、SORT-ITEMSアクションを使用してソートを明示的に実行すると、ソートされた順序で項目が再整列されます。 ソートを回避できず、かつ、アイコンビューモードで明示的に項目の位置を指定している場合、ソート前にアイコンの位置を明示的に保存しておき、ソート後に元の位置に戻す必要があります。 次に例を示します。

```
PERFORM SAVE-ITEM-POSITIONS
PROCESS GUI ACTION SORT-ITEMS WITH #CONTROL GIVING *ERROR
IF #CONTROL.VIEW-MODE = VM-ICON OR
   #CONTROL.VIEW-MODE = VM-SMALLICON
  PERFORM RESTORE-ITEM-POSITIONS
END-IF
```

#CONTROLはリストビューコントロールのハンドルです。また、項目の位置を保存およびリストアするために前述の例で定義したサブルーチンが使用されています。アイコンが（元の位置で）再描画されるため、画面がちらつく場合があることに注意してください。 したがって、可能であれば、アイコンビューモードの1つでリストビューコントロールを使用している間は、ソート

が実行されないようにしてください。 これを実行する方法の例については、以下のラベル編集についてのセクションを参照してください。

ラベル編集

リストビューコントロールのラベル編集プロセスは、ツリービューコントロールと同じです。 このため、この内容の詳細については、「*ツリービューおよびリストビューコントロールでのラベル編集*」を参照してください。

SORTED 属性が設定されている場合でも、ラベル編集の操作が完了した後は、項目の順序付けが自動的に再実行されることはありません。 これが必要な場合は、以下の例に従って操作してください。 まず、後で使用する変数をいくつか定義します。

```
01 #SORTOBJ HANDLE OF GUI
01 #SORT (L)
01 #AUTO-ARRANGE (I4)
```

また、リストビューコントロールの名前は #LV-1 であるとします。

リストビューコントロールの AFTER-EDIT イベントでは、（まだ完了していない）編集プロセスに影響があるため、非同期で順序付けを再実行できません。代わりに、#SORT フラグを設定し、編集プロセスの完了後に順序付けを再実行するよう指定します。

```
#SORT := TRUE
```

項目の順序付けの再実行を実行するかどうかを判断するために、項目がすでにソートされているかどうかを確認する必要があります。 これは、リストビューに列がある場合はプライマリ列の SORTED 属性と DESCENDING 属性、および列がない場合はリストビューコントロール自体のこれらの属性をクエリすることによって実行できます。 適切なオブジェクトハンドルをダイアログの AFTER-OPEN イベントで設定します。

```
IF #LV-1.COLUMN-COUNT <> 0
  #SORTOBJ := #LV-1.FIRST-CHILD
ELSE
  #SORTOBJ := #LV-1
END-IF
*
EXAMINE #LV-1.STYLE FOR 'a' GIVING NUMBER #AUTO-ARRANGE
IF #LV-1.SORTED AND #AUTO-ARRANGE <> 0 AND
  (#LV-1.VIEW-MODE = VM-ICON OR
   #LV-1.VIEW-MODE = VM-SMALLICON)
```

```
 #SORT := TRUE
END-IF
```

また、リストビューコントロールの［自動整列（a）］STYLE フラグの設定を取得します。 このフラグが設定されている場合、アイコンの位置を修正する必要がないため、コントロールがアイコンビューモードであっても項目をソートできます。 さらに、コントロールがソートされている場合、まず項目の順序付けを再実行するように指定します。 これは、コントロールの SORTED フラグが設定されている場合、アイコンビューモードで項目を挿入しても項目の位置が更新されないという（前述の）事実に基づいています。 したがって、この場合、アプリケーションで初期ソート自体を実行する必要があります。

項目の順序付け再実行の実際の作業は、ダイアログの IDLE イベントで非同期的に行われます。

```
IF #SORT
  IF #AUTO-ARRANGE <> 0 OR
    (#LV-1.VIEW-MODE <> VM-ICON AND
     #LV-1.VIEW-MODE <> VM-SMALLICON)
    IF #SORTOBJ.SORTED
     PROCESS GUI ACTION SORT-ITEMS WITH
       #SORTOBJ #SORTOBJ.DESCENDING GIVING *ERROR
    END-IF
    RESET #SORT
  END-IF
END-IF
```

アイコンビューモードのリストビューコントロールに対しては、自動整列が設定されている場合にのみソートが実行されることに注意してください。自動整列が設定されていない場合、#SORT フラグはリセットされず、コントロールがアイコンビューモード以外のモードに切り替えられて最初の IDLE イベントが発生するまで、順序付けの再実行は保留されます。

マルチプルコンテキストメニュー

コントロールで単一のコンテキストメニューを使用する場合、表示するコンテキストメニューのハンドルをコントロールのCONTEXT-MENU属性に設定してそのままにしておきます。ただし、特定のコントロールで複数のコンテキストメニューを表示する必要があることがよくありますが、このアプローチでは柔軟性が足りません。

上記の問題に対応するために、CONTEXT-MENU イベントが導入されました（上記の同じ名前の属性と混同しないように注意してください）。 このイベント（無効にされていない場合）は、CONTEXT-MENU属性が評価される直前にターゲットのコントロールで発生します。これによって、アプリケーションは最初に、適切なコンテキストメニューのハンドルにこの属性をダイナミックに設定できます。

例として、ダイアログエディタで2つのコンテキストメニューを定義したとします。1つは項目に関連するコマンドが組み込まれている #CTXMENU-ITEMS で、もう1つは汎用コマンド（リストビューコントロールの表示モードの切り替えなど）が組み込まれている #CTXMENU-DEFAULT です。 この場合、次の CONTEXT-MENU イベントを使用することができます。

```
#CONTROL := *CONTROL
IF #CONTROL.SELECTED-SUCCESSOR <> NULL-HANDLE
   #CONTROL.CONTEXT-MENU:= #CTXMENU-ITEMS
ELSE
   #CONTROL.CONTEXT-MENU := #CTXMENU-DEFAULT
END-IF
```

ローカルデータの定義は以下のように想定されています。

```
01 #CONTROL HANDLE OF GUI
```

この例では、項目が占めている位置をユーザーが右クリックすると、コンテキストメニュー #CTXMENU-ITEMS が表示されます。それ以外の場合は、#CTXMENU-DEFAULT が表示されます。

この手法をさらに活用して、選択された項目のタイプに固有のコンテキストメニューを表示することもできます。

ドラッグ & ドロップ

ドラッグ & ドロップは、他のウィンドウとのデータ転送だけでなく、リストビューコントロール内の項目を再配置するために使用できます。 この 2 つのケースに特に違いはありません。 再配置の場合、ドラッグを開始したリストビューコントロール内でドロップ操作を行うという点が特殊なだけです。

ドラッグ & ドロップをサポートするための基本的な手法については、「*クリップボードおよびドラッグ & ドロップの使用*」に記載されています。 特に、コントロール内の項目を再配置をするだけの場合であっても、ドラッグ & ドロップを開始するよう Natural に通知するために、ドラッグ & ドロップクリップボードにデータを転送する必要があることに注意してください。 次に、リストビューコントロール固有の注意点があります。項目を再配置するとリストビューコントロールの基点が変更され、リストビューコントロールの表示エリアの近い方の上隅の座標がデフォルトの基点の (0, 0) で始まらなくなる場合があります。

以下の例は、次の操作をサポートするためにリストビューコントロールでドラッグ & ドロップを使用するコードを示しています。

1. リストビュー項目を再配置する。

2. 別のアプリケーション（ワードパッドなど）からリストビュー項目にテキストをドラッグ & ドロップしてラベルを変更する。

まず最初に、ドラッグ & ドロップモードをリストビューコントロールに適切に設定します。 ダイアログエディタの［リストビューコントロール属性］ウィンドウで、［ドラッグモード］選択ボックスを"［移動］"に、［ドロップモード］選択ボックスを"［コピー+移動］"に設定します。これにより、コントロールの DRAG-MODE 属性および DROP-MODE 属性がそれぞれ DM-MOVE および DM-COPYMOVE に、ダイアログの生成ソースコード内で設定されます。

次に、以下で使用される必要なローカルの変数を定義する必要があります。

```
01 #CONTROL HANDLE OF GUI
01 #DROP-ITEM HANDLE OF GUI
01 #ITEM HANDLE OF GUI
01 #SELECTED (L)
01 #AVAIL (L)
01 #X (I4)
01 #Y (I4)
01 #ORIG-X (I4)
01 #ORIG-Y (I4)
```

これを行った後、必要なイベントハンドラを記述することができます。 処理を開始する論理的な場所は、BEGIN-DRAG イベントです。

```
#CONTROL := *CONTROL
*
PROCESS GUI ACTION INQ-CLICKPOSITION WITH
  #CONTROL #ORIG-X #ORIG-Y GIVING *ERROR
*
ADD #CONTROL.OFFSET-X TO #ORIG-X
ADD #CONTROL.OFFSET-Y TO #ORIG-Y
*
PROCESS GUI ACTION SET-CLIPBOARD-DATA WITH
  'DUMMYPRIVFMT' 0 GIVING *ERROR
```

このコードは以下の 3 つの部分で構成されています。

1. クリック位置をクライアント座標で取得する。
2. クリック位置をビュー座標に変換する（前述のとおり、リストビューウィンドウの基点は常に "(0, 0)" とは限りません）。
3. ダミーデータをドラッグ & ドロップクリップボードに転送する。転送しないと、Natural はドラッグ & ドロップ操作を開始しません。 使用するのはダミーデータのみであるため、他のアプリケーションで認識されないように、ここではプライベートクリップボードフォーマットを選択します。

次に、DRAG-ENTER イベントのハンドラを用意します。

```
PROCESS GUI ACTION INQ-DRAG-DROP WITH
  1x #CONTROL GIVING *ERROR
*
IF #CONTROL = *CONTROL
  IF #CONTROL.VIEW-MODE = VM-ICON OR
     #CONTROL.VIEW-MODE = VM-SMALLICON
    #CONTROL.SUPPRESS-DRAG-DROP-EVENT := NOT-SUPPRESSED
  ELSE
    #CONTROL.SUPPRESS-DRAG-DROP-EVENT := SUPPRESSED
  END-IF
ELSE
  #CONTROL := *CONTROL
  PROCESS GUI ACTION INQ-FORMAT-AVAILABLE WITH CF-TEXT #AVAIL GIVING *ERROR
  IF #AVAIL
    #CONTROL.SUPPRESS-DRAG-DROP-EVENT := NOT-SUPPRESSED
  ELSE
    #CONTROL.SUPPRESS-DRAG-DROP-EVENT := SUPPRESSED
  END-IF
END-IF
```

上記のコードは、以下の 3 つの部分で構成されています。

1. INQ-DRAG-DROP アクションを呼び出し、ドラッグソースコントロールのハンドルを確認します。
2. ドラッグソースが現在のコントロールの場合、ドラッグソースとドロップターゲットは同じです。つまり、これは項目の再配置のケースです。項目の再配置はアイコンビューモードの 1 つでのみ実行可能なため、この場合は DRAG-DROP イベントを抑制せずにドロップを実行可能にします。それ以外の場合、このイベントを無効にしてドロップを禁止し、ドロップアンドドラッグカーソルが表示され「ない」ようにします。この場合、ドラッグ & ドロップクリップボードにデータを何も転送しないことで、ドラッグがまったく発生しないようにすることもできたことに注意してください。ただし、この例では示されていませんが、ソースコントロールがリストビューモードまたはレポートビューモードであっても、1つ以上の項目をリストビューコントロールから別のウィンドウにドラッグする必要があることはよくあります。その場合、上記のコードを基本として使用できます。
3. ドラッグソースとドロップターゲットが異なる場合、別のウィンドウからのデータ転送を試みます。この場合、必要なフォーマット CF-TEXT でデータを使用できるかどうかを確認し、使用可能な場合はドロップを可能にします。そうでない場合、ドロップは禁止されます。

外部データをリストビューコントロール上でドラッグしているときにドロップの強調表示を行うには、DRAG-OVER イベントハンドラを使用します。

```
PROCESS GUI ACTION INQ-DRAG-DROP WITH
  1X #CONTROL 1X #X #Y GIVING *ERROR
*
IF #CONTROL <> *CONTROL
  #CONTROL := *CONTROL
  IF #CONTROL.SUPPRESS-DRAG-DROP-EVENT = NOT-SUPPRESSED
    PROCESS GUI ACTION INQ-ITEM-BY-POSITION WITH
      #CONTROL #X #Y #ITEM GIVING *ERROR
    IF #ITEM <> #DROP-ITEM
      IF #DROP-ITEM <> NULL-HANDLE
        #DROP-ITEM.SELECTED := #SELECTED
      END-IF
      #DROP-ITEM := #ITEM
      IF #DROP-ITEM <> NULL-HANDLE
        #SELECTED := #DROP-ITEM.SELECTED
        #DROP-ITEM.SELECTED := TRUE
      END-IF
    END-IF
  END-IF
END-IF
```

上述したコードにより、次の処理が実行されます。

1. INQ-DRAG-DROP アクションが呼び出され、ドラッグソースコントロールのハンドルおよび現在のドラッグの位置が判断されます。
2. ドラッグソースとドロップターゲットが同じ場合（項目の再配置）、ドロップの強調表示は不要なため、これ以上アクションは実行されません。 ドラッグソースとドロップターゲットが異なる場合、INQ-ITEM-BY-POSITION アクションを使用して、現在のドロップ位置にあるリストビュー項目（存在する場合）を確認します。
3. 現在のターゲット項目（「ドロップ項目」）を #DROP-ITEM に保存します。 ドロップ項目が変わるたびに、まず新しいドロップ項目の現在の選択状態を #SELECTED に保存します。その後、項目の SELECTED 属性を TRUE に設定することにより、項目を選択してドロップの強調表示を行います。 また、元のドロップ項目（存在する場合）の選択状態は、前に保存した値に戻されます。
4. ドロップを実行できない場合（つまり、SUPPRESS-DRAG-DROP-EVENT 属性が SUPPRESSED に設定されている場合）、ドロップの強調表示は適用されないことに注意してください。

実際のドロップを実行するために、DRAG-DROP イベントハンドラが提供されています。

```
PROCESS GUI ACTION INQ-DRAG-DROP WITH
  1x #CONTROL 1x #X #Y GIVING *ERROR
*
IF #CONTROL = *CONTROL
  ADD #CONTROL.OFFSET-X TO #X
  ADD #CONTROL.OFFSET-Y TO #Y
  SUBTRACT #ORIG-X FROM #X
  SUBTRACT #ORIG-Y FROM #Y
  #ITEM := #CONTROL.SELECTED-SUCCESSOR
  REPEAT WHILE #ITEM <> NULL-HANDLE
    ADD #X TO #ITEM.RECTANGLE-X
    ADD #Y TO #ITEM.RECTANGLE-Y
    #ITEM := #ITEM.SELECTED-SUCCESSOR
  END-REPEAT
ELSE
  IF #DROP-ITEM <> NULL-HANDLE
    PROCESS GUI ACTION GET-CLIPBOARD-DATA WITH CF-TEXT #DROP-ITEM.STRING
      GIVING *ERROR
    #DROP-ITEM.SELECTED := #SELECTED
    RESET #DROP-ITEM
  END-IF
END-IF
```

上述したコードにより、次の処理が実行されます。

1. INQ-DRAG-DROP アクションが呼び出され、ドラッグソースコントロールのハンドルおよび現在のドラッグの位置が判断されます。
2. このイベントハンドラは、ドラッグソースとドロップターゲットが同じ場合（項目の再配置）と異なる場合（外部ソースからのドラッグ）を区別します。
3. 項目の再配置の場合、ドロップ位置をビュー座標に変換して基点からの変位を "(x, y)" で取得します。その後、この変位を選択された各項目に適用して移動させます。
4. 外部ソースからのドラッグの場合、クリップボードから取得したテキストデータを現在のドロップ項目（存在する場合）の STRING 属性に直接設定して、ラベルを更新します。 ドラッグ & ドロップクリップボードでテキストを使用できるかどうかの確認は、すでに DRAG-ENTER イベントで実行して、使用できない場合はドロップおよび関連付けられている DRAG-DROP イベントが禁止されているため、最初に実行する必要はありません。ラベルの更新後は、ドロップ項目の選択状態を元（ドラッグ前の選択状態）に戻し、次のドラッグ操作（もしあれば）に備えて #DROP-ITEM をリセットします。

最後に、ユーザーがドラッグ操作をキャンセルした場合、またはドロップを実行せずにリストビューコントロールの境界線の外に出た場合は、DRAG-LEAVE イベントハンドラを使用します。

```
IF #DROP-ITEM <> NULL-HANDLE
  #DROP-ITEM.SELECTED := #SELECTED
  RESET #DROP-ITEM
END-IF
```

上記のコードは、ドロップ項目の選択状態を元に戻して、ドロップの強調表示（行われている場合）を単にクリアしています。他のイベントハンドラのロジックと合わせるために、#DROP-ITEM をリセットして、ドロップ項目がないことを示します。 DRAG-LEAVE イベントはドロップでは呼び出されないため、このコードは DRAG-DROP イベントの最後に実行されるコードと基本的に同じコードになっています。 したがって、ドロップの強調表示のリセットはこの両方で行う必要があります。

100 ネスト構造のコントロールの操作

このchapterでは、次のトピックについて説明します。

はじめに

ダイアログの直下の子である 「最上位」 コントロール以外にも、他のコントロールの子としてコントロールを作成できます。 このようなコントロールはネスト構造コントロールと呼ばれ、親コントロールはコンテナと呼ばれます。 同一の親を持つ子コントロール集合を表すのに兄弟という用語を使います。 明らかに、コントロール階層内には多くの異なる兄弟コントロール集合が存在し得ます。

コントロール階層を作成することによって、Naturalプログラマはコントロールをグループ化でき、Naturalプログラム内で簡単かつ効率的に操作できるようになります。以下に、ネスト構造コントロールの特徴について説明します。

- ネスト構造コントロールの位置は、ダイアログではなくコンテナコントロールのクライアントエリアに対する相対位置です。
- ネスト構造コントロールの表示は、祖先のウィンドウの大きさで切り取られます。 つまり、コンテナの境界線の外にあるネスト構造コントロールのエリアは表示されません。 ダイアログエディタでは、ネスト構造コントロールをコンテナの外にドラッグできません。
- ネスト構造コントロールは、コントロールシーケンス内での位置に関係なく、常にコンテナコントロールの前面に表示されます。
- ネスト構造コントロールは、コンテナコントロールとともに移動されます。 これは、ダイアログエディタにおける編集時（コンテナをドラッグするとき）とランタイム時（コンテナの RECTANGLE-X 属性および／または RECTANGLE-Y 属性を変更するとき）の両方に当てはまります。
- コンテナコントロールが非表示になると、ネスト構造コントロールは、VISIBLE属性が変更されていなくても非表示になります。
- コンテナコントロールが使用不可になると、ネスト構造コントロールは、ENABLED属性が変更されておらずコントロールがグレー表示されていなくても、ランタイム時に使用不可になります。
- コンテナコントロールが削除されると、ネスト構造コントロールも削除されます。

Note: Natural はコントロール階層に含まれるレベル数を制限しません。 ダイアログエディタステータスバーのコンボボックスに、コントロール名とともにコントロールのレベルが表示されます。

コンテナになり得るコントロールタイプ

すべてのコントロールタイプがコンテナになれるわけではありません。 例えば、コントロールを入力フィールドの子として作成することはできません。 Naturalによってサポートされるコンテナコントロールには、現時点では以下の 3 つのタイプがあります。

- 新しい"コンテナ" スタイルが設定されるグループフレーム。 このスタイルは、グループフレームの作成後にダイアログエディタの属性ウィンドウで変更できます。 グループフレームがコンテナに変換されると、コンテナに含まれるすべてのコントロールは、グループフレームの子孫になるようにコントロール階層で移動されます。 グループフレームがコンテナ以外に変換されると、グループフレームの直下の子はグループフレームの兄弟になるように1つ上の階層レベルに移動されます。
- レジストリに "OLEMISC_SIMPLEFRAME" としてマークされる ActiveX コントロール。 このフラグは、設計によって特定の ActiveX コントロールクラスに固定されます。
- コントロールボックス。 このコントロールタイプは常にコントロールコンテナです。 実際、これがこのコントロールの目的のすべてです。 詳細については、「*コントロールボックスの操作*」を参照してください。

ネスト構造コントロールの作成

ネスト構造コントロールは、ネストしないコントロールと同じ方法でダイアログエディタで作成します。 コントロール挿入時に最初にマウスの左ボタンをコンテナコントロール上でクリックすると、Naturalによって新しいコントロールがコンテナの子として自動的に作成されます。 マウスボタンを挿入モードでクリックする前でも、マウスカーソルがダイアログ上を移動するのに合わせて、ダイアログエディタのステータスバーがコンテナに対する相対的なマウス座標で絶えず更新されます。

また、前述のようにグループフレームをコンテナに変換するときに、ネスト構造コントロールがダイアログエディタ内で間接的に作成されます。

ダイアログのハンドルではなく必要なコンテナコントロールのハンドルとして PARENT 属性を指定することによって、PROCESS GUI ACTION ADD ステートメントを使用してランタイム時にネスト構造コントロールをダイナミックに作成できます。 ネスト構造コントロールの位置（RECTANGLE-X 属性および RECTANGLE-Y 属性）は、コンテナのクライアントエリアに対して相対的に指定する必要があります。 コントロールのクライアントエリアは、3-D ボーダー、"フレーム" スタイルによって生じる 1 ピクセルフレーム、コントロールのスクロールバーなどのフレームコントロールを除いた、コントロールの内部エリアです。

複数選択、コントロールシーケンス、およびクリップボード操作

ダイアログエディタは、親が同じではない（つまり、兄弟でない）コントロールの複数選択を禁止します。 これは、複数のコントロールが 「rubber banding」（マウスの左ボタンを押したままドラッグして範囲を選択する）と拡張選択（Shift キーを押したままコントロールを選択する）のどちらで選択されたかに関係なく適用されます。 ただし、選択したコンテナコントロールを削除すると、その直接的および間接的な子（*子孫*）はすべて、明示的に選択していなくても削除されます。 そのため、クリップボードへの切り取り操作は、選択したコントロール*および*すべての子孫コントロールをクリップボードにコピーします。 クリップボードへのコピー操作はコンテナだけをコピーするかコンテナに加えてすべての子孫をコピーするか決まっていません。 この場合は、メッセージボックスが表示され、ユーザーはこれらの2つのオプションから選択できます。

クリップボードからのコントロールの貼り付けでは、新規作成されるコントロールと同じコントロールシーケンス（タブオーダー）挿入位置ロジックが使用されます。 どちらの場合も、新しいコントロールが、選択された兄弟およびその子孫のすぐ後のコントロールシーケンス位置に作成されます。 兄弟以外のコントロールが選択された場合は、選択された（アクティブな）コントロールのコントロールシーケンス内の位置に基づいて、「事実上の兄弟」 が代わりに使用されます。 選択された「アクティブ」 であるコントロールとは、黒（グレーではない）の選択ハンドルを使用して強調表示された選択済みコントロールを指します。 選択がアクティブでない場合、コントロールはコントロールシーケンス内で最初の兄弟コントロールの直前か、あるいはコンテナが空であればコンテナの直後（または最上位コントロールのコントロールシーケンスの前）に挿入されます。 ただし、コントロールシーケンスは階層に関係なくメンテナンスされることに注意してください。 コントロールが作成されたら、［ダイアログ］メニューの［操作順序設定］オプションを使用して、どのコントロールもコントロールシーケンス内の任意の位置に明示的に移動できます。

新規作成されるコントロールの階層内での位置の決定方法は、これら2つの場合で少し異なります。 コントロールを新しく作成する場合は、マウスの左ボタンを押した位置で最上位コンテナを探すとコンテナを特定できます。 ただし、コントロールをクリップボードから貼り付ける場合は、使用できる"（X,Y）"位置がないため、コントロールが選択されていればそのコンテナ、選択されていなければダイアログ自体がコンテナであるとみなされます。 つまり、例えばコントロールをあるコンテナから別のコンテナにコピーおよび貼り付けする場合、貼り付ける前に、コンテナ自体ではなく2つ目のコンテナ内のコントロールを選択する必要があります。2つ目のコンテナが空であれば、まずダミーの子コントロールを暫定的に作成する必要があります。ダミーの子コントロールは、貼り付け操作の完了後に削除できます。

コントロールを削除すると、その子孫コントロールも削除されます。

ネスト構造コントロールの導入によって、［すべてを選択］コマンドの動作が以下のように変わりました。

- コントロールが現在選択されていない場合、コマンドはすべての最上位コントロールを選択します。

- コントロールが現在選択されている場合、その兄弟である他のすべてのコントロールも追加選択されます。

したがって、階層の1レベルだけを使用する一般的な場合では、［すべてを選択］コマンドは以前と同様にすべてのダイアログコントロールを選択します。

101 ダイナミックな情報行の操作

フォーカスされているダイアログエレメントに関する説明テキストをダイナミック情報行（DIL）に表示すると、イベントドリブンアプリケーションがさらにユーザーフレンドリになります。ダイアログエレメントは、エンドユーザーがキーボード入力できる場合にフォーカスを獲得します。

ダイアログエレメントを DIL テキストに関係付けるには、以下の 2 つの方法があります。

- ダイアログエディタを使用します（一番簡単な方法であるため、使用される可能性が高い）。
- Natural コードを使用して、すべてをダイナミックに指定します。

ダイアログエディタ使用時の実行手順

1. ［ダイアログ属性］ウィンドウ内の［インフォメーション行］エントリをマークすることによって、ダイアログの HAS-DIL 属性を TRUE に設定します。
2. ダイアログエレメントの DIL-TEXT 属性を "*diltextstring*" に設定します。 属性ウィンドウで、［**DIL** テキスト］エントリの右にある［**...**］ボタンを押します。 ［属性ソース］ウィンドウが開きます。属性ソースの1つを選択して、［値］フィールドにテキストを入力します。ダイアログエレメントの使用方法が、"*diltextstring*" によって簡単に説明されていることを確認してください。

Natural コードを使用して上記 2 つの手順を実行する場合は、以下のようになります。

```
...
PERSDATA-DIALOG.HAS-DIL := TRUE  /* Set HAS-DIL To TRUE
#PB-1.DIL-TEXT := 'DILTEXTSTRING'   /* Assign the text string
...
```

Note: ダイアログにステータス行が含まれ、テキストが DIL に表示される場合、STATUS-TEXT および DIL-TEXT は同じエリアに表示されます。

102 スピンコントロールの操作

このchapterでは、次のトピックについて説明します。

はじめに

スピンコントロールは、「アップダウン」 コントロールと呼ばれる上下対になった矢印ボタンと、「バディ」 コントロールと呼ばれる、任意で関連付けられた入力フィールドコントロールによって構成されます。スピンコントロールには整数範囲と現在位置が関連付けられています。バディコントロール（存在する場合）には、現在位置に関連付けられている内容が表示されます。 現在位置は、矢印ボタンや↑キーおよび↓キーを使用したり、バディ入力フィールド（使用可能かつ変更可能な場合）に直接値を入力したりすることによって、変更できます。

アップダウンコントロール

アップダウンコントロールを使用すると、スピンコントロールの整数範囲を明示的にスクロールできます。 アップダウンコントロールのボタンは、↑キーおよび↓キーを使用すると暗黙的に押されます。 ボタンまたはキーを押し続けると、値が連続してインクリメントまたはデクリメントされます。コントロールの［折り返し（w）］スタイルが設定されている場合、範囲の上限または下限に到達すると、反対側の端の値がサイクリックに表示されます。 最初のインクリメント数またはデクリメント数は1ですが、数秒間続くとこの値は増加します。 SET-ACCELERATION アクションでは、この加速を無効にしたり、変更したりできます。

コントロールの範囲は、MIN 属性および MAX 属性を設定することによって定義できます。 コントロールの範囲内の現在位置は、POSITION 属性値を設定またはクエリすることによって、プログラムで設定したり取得したりできます。

現在位置がユーザーによって変更されると、コントロールの CHANGE イベントが発生します（抑制されていない場合）。 コントロールの値がプログラムによって変更された場合は、このイベントは発生しません。 代わりに、バディ入力フィールドコントロールが存在する場合、バディコントロールの CHANGE イベント を使用できます。

バディコントロール

コントロールの［左に配置（l）］または［右に配置（r）］の各スタイルフラグを設定すると、バディコントロールと呼ばれる入力フィールドがスピンコントロールに組み込まれます。バディコントロールは、スタイルフラグの設定に応じて、アップダウンコントロールの左または右に表示されます（配置は、バディコントロールではなくアップダウンコントロールに対する位置が基準になります）。

バディコントロールはスピンコントロールの子で、標準のNatural入力フィールドコントロールとして表示されます。 したがって、Naturalプログラマは、単独の入力フィールドコントロール

で使用できるすべての機能にアクセスできます。 例えば、バディコントロールは、数字のみを受け付けるようにしたり、変更不可能にしたりできます。

スピンコントロールの［バディの設定（s）］スタイルが設定されている場合、アップダウンコントロールの現在位置が変更されると、その現在位置に合わせてバディコントロールが自動的に更新されます。［バディの設定（s）］スタイルが設定されていない場合、スピンコントロールの CHANGE イベントに応じて、バディコントロールの内容を手動で更新する必要があります。

日付／時刻のフォーマット

デフォルトでは、日付／時刻の情報は、現在の地域の設定に定義されている日付／時刻のフォーマットに従って表示されます。 Windows には長いフォーマットと短いフォーマットの 2 つの代替日付フォーマットがあり（両方ともユーザーが変更可能）、また短い日付フォーマットには世紀の情報が含まれないことがあるため、3 つの STYLE フラグの 1 つにより、使用される標準の日付フォーマットが決定されます。 これらのフォーマットは以下のとおりです（相互排他的）。

- ［短い日付 (s)］。現在の地域の設定に短い日付フォーマットを使用することを意味します。
- ［世紀日付 (c)］。現在の地域の設定に短い日付フォーマットを使用することを意味しますが、世紀情報が含まれていない場合は、世紀情報を提供するように拡張されます。 多くの場合、短い日付フォーマットには世紀情報がすでに含まれています。この場合、このスタイルにより日付の外観が変わることはありません。
- ［長い日付（l）］。現在の地域の設定に長い日付フォーマットを使用することを意味します。

また、［時刻（t）］スタイルフラグは、コントロールで（日付ではなく）時刻情報を表示することを示すために用意されています。

これらの標準のフォーマットが十分でない場合は、EDIT-MASK 属性を使用して、これらのフォーマットよりも優先させるカスタムフォーマット文字列を指定することができます。 ただし、フォーマット文字列の指定子は、Natural 内の他の場所にある編集マスクに使用された指定子には対応していません。 次のテーブルに、使用可能な指定子とその意味を示します。

指定子	説明
d	1 桁または 2 桁の日付。
dd	2 桁の日付。 1 桁の日付の場合、先頭に 0 が付けられます。
ddd	3 文字の週日の略記。
dddd	完全な週日名。
h	12 時間フォーマットの 1 桁または 2 桁の時間。
hh	12 時間フォーマットの 2 桁の時間。 1 桁の値の場合、先頭に 0 が付けられます。
H	24 時間フォーマットの 1 桁または 2 桁の時間。
HH	24 時間フォーマットの 2 桁の時間。 1 桁の値の場合、先頭に 0 が付けられます。

指定子	説明
m	1 桁または 2 桁の分。
mm	2 桁の分。 1 桁の値の場合、先頭に 0 が付けられます。
s	1 桁または 2 桁の秒。
ss	2 桁の秒。 1 桁の値の場合、先頭に 0 が付けられます。
M	1 桁または 2 桁の月番号。
MM	2 桁の月番号。 1 桁の値の場合、先頭に 0 が付けられます。
MMM	3 文字の月の略記。
MMMM	完全な月名。
t	AM/PM 略記の 1 文字（つまり、AM は "A" と表示される）。
tt	AM/PM 略記の 2 文字（つまり、AM は "AM" と表示される）。
yy	年の下 2 桁（つまり、2005 年は "05" と表示される）。
yyyy	完全な年（つまり、2005 年は "2005" と表示される）。

また、引用符で囲まれた文字は、指定したとおりに表示されます。 引用符で囲まれた引用符を指定する場合は、2つの一重引用符を連続して使用する必要があります。 スペースおよび句読点の記号（コンマなど）は、引用符で囲む必要はありません。

例えば、"John's birthday is Friday, December 31, 1969" という文字列を表示するには、DTP コントロールの `EDIT-MASK` 属性を "'John''s birthday is' dddd, MMMM d, yyyy" に設定します。

日付および時刻の入力

DTP コントロールには、指定された情報を修正するための方法がいくつかあります。

- ユーザーが、数値情報（日付など）を直接入力します。
- ユーザーが、選択されたフィールド（日付、月名など）の値を、+ キーまたは - キーを使用して、それぞれ増やしたり減らしたりします。
- DTP コントロールのスタイルが［時刻 (t)］または［上下 (u)］のいずれかの場合、ユーザーが、必要なフィールドを選択し、上下（「スピン」）コントロールを使用して数値を増やしたり減らしたりします。
- DTP コントロールでカレンダーが使用されている場合、ユーザーが、下向きの矢印を押してカレンダーを開き、必要な日付に移動します。 上述した方法とは異なり、この方法ではすべての日付フィールドが同時に更新されます。
- プログラム的に、必要な日付または時刻で `TIME` 属性を更新します。

例えば、DTP コントロールで日付または時刻を現在の日付または時刻に設定するには、次の割り当てを使用します。

```
#DTP-1.TIME := *DATX
```

または

```
#DTP-1.TIME := *TIMX
```

それぞれ、#DTP-1 は DTP コントロールのハンドルであるとみなされます。

DTP コントロールでは、日付または時刻のコンポーネントの編集しか許可されませんが、コントロールのスタイルに応じて、日付および時刻の両方の情報を保存します。

空値

DTP コントロールに［値なし"" を許可 (n)］のスタイルが指定されている場合、コントロールにチェックボックスが表示されます。 このチェックボックスがオンになっていない場合、そのコントロールに関連付けられた日付または時刻はないと解釈されます。 アプリケーションは、コントロールの CHECKED 属性をクエリすることで、この状態をテストすることができます。また、CHECKED 属性を UNCHECKED に戻すことで、「値なし」の状態にコントロールを戻すこともできます。 ただし、CHECKED 属性を明示的に CHECKED に設定することはできません。これは、日付または時刻がコントロールに適用されるたびに、暗黙的に行われるためです。 さらに、CHECKED 属性は、［値なし"" を許可 (n)］スタイルを使用しない場合、DTP コントロールにまったく設定できません。

カレンダーのカラーおよびフォント

DTP コントロールと関連してカレンダー（存在する場合）が使用するカラーおよびフォントは、それぞれ SET-AUX-COLOR アクションおよび SET-AUX-FONT アクションを使用して変更できます。

103 ステータスバーの操作

ダイナミック情報行と同様に、ステータスバーもイベントドリブンアプリケーションをさらにユーザーフレンドリにします。

ダイアログエレメントをステータスバーに関係付けるには、以下の 2 つの方法があります。

- ダイアログエディタを使用します（一番番簡単な方法であるため、使用される可能性が高い）。
- Natural コードを使用して、すべてをダイナミックに指定します。

ダイアログエディタを使用する場合は、以下を実行する必要があります。

- ［ダイアログ属性］ウィンドウ内の［ステータスバー］エントリをマークすることによって、ダイアログの HAS-STATUS-BAR を TRUE に設定します。 HAS-STATUS-BAR 属性はステータスバーを変更できるかどうかを決めます。 HAS-STATUS-BAR が FALSE で HAS-DIL が TRUE の場合、ステータスバーは表示されますが、ダイナミック情報行としてのみ使用されます。

Natural コードを使用して上記の手順を実行する場合は、以下のようになります。

```
...
PERSDATA-DIALOG.HAS-STATUS-BAR := TRUE  /* Set HAS-STATUS-BAR To TRUE
PERSADTA-DIALOG.STATUS-TEXT := 'HELLO'  /* Set the text to 'Hello'
...
```

Note: ダイアログにステータス行が含まれ、テキストが DIL に表示される場合、STATUS-TEXT および DIL-TEXT は同じエリアに表示されます。

104 ステータスバーコントロールの操作

このchapterでは、次のトピックについて説明します。

はじめに

Note: ステータスバーコントロールと従来のダイアログステータスバー（ダイアログエディタの［ダイアログ属性］ウィンドウの［ステータスバー］チェックボックスを選択するか、またはランタイム時にダイアログの HAS-STATUS-BAR 属性を選択することによって作成される）を混同しないでください。 ステータスバーコントロールを使用している場合は、［ステータスバー］チェックボックスのチェックを解除し、HAS-STATUS-BAR 属性は設定しないでください。

ステータスバーコントロールの作成

ステータスバーコントロールは、他の標準コントロール（リストボックスやプッシュボタンなど）と同じようにダイアログエディタで作成します。 つまり、ダイアログエディタの［挿入］メニューを使用して、あるいは［挿入］ツールバーからのドラッグ & ドロップによってスタティックに作成するか、または TYPE 属性を STATUSBARCTRL に設定した PROCESS GUI ACTION ADD ステートメントを使用してランタイム時にダイナミックに作成します。

ステータスバーコントロールは、他のほとんどのコントロールタイプとは異なり、別のコントロール内にネストすることも MDI 子ダイアログ内に作成することもできません。 MDI アプリケーションでは、ステータスバーコントロールは MDI フレームダイアログに属す必要があります。

ステータスバーコントロールには、関連付けられたウィンドウを含めることもまったく含めないこともできます。ウィンドウは、ダイアログエディタでステータスバーコントロールの属性ウィンドウから定義するか、またはランタイム時に TYPE 属性を STATUSBARPANE に設定した PROCESS GUI ACTION ADD ステートメントを実行することによって定義できます。

ウィンドウのないステータスバーコントロールの使用

ステータスバーコントロールの拡張機能へのアクセスを提供するほとんどの属性は、ステータスバーウィンドウに対してのみサポートされるため、ウィンドウのないステータスバーコントロールでは、提供される機能が限定的となります。 ステータスバーコントロールでテキスト行の表示以外を実行したいが、ステータスバーコントロールを複数部分に分割する必要がない場合は、ステータスバーコントロールの幅全体を占める 1 つのウィンドウを作成することをお勧めします。

伸縮性のあるウィンドウと伸縮性のないウィンドウ

ステータスバーコントロールにウィンドウを定義する場合、ステータスバーコントロールを含むダイアログがサイズ変更されたときに各ウィンドウの幅が伸び縮みするか変わらないようにするかを決める必要があります。 幅が伸び縮みするものは「伸縮性のある」 ウィンドウ、幅が変わらないものは「伸縮性のない」 ウィンドウと呼ばれます。

［ステータスバーコントロール属性］ウィンドウには、ウィンドウの伸縮性をマークするための明確なフラグはありません。 フラグの代わりに、幅（RECTANGLE-W属性）に0が定義されたウィンドウは伸縮性があり、0以外が定義されたウィンドウは指定された幅（ピクセル数）で固定だと暗黙的にみなされます。 RECTANGLE-W 属性は 0 にデフォルト設定されるため、ダイアログエディタで定義した初期状態では、すべてのウィンドウに伸縮性があります。

伸縮性のある各ウィンドウの表示幅は、ステータスバーコントロールのウィンドウ表示に使用できる合計幅からすべての伸縮性のないウィンドウの幅を引いたものを、伸縮性のあるウィンドウ数で割って決められます。

Note: 通常、ステータスバーコントロールのウィンドウ表示に使用できる合計幅は、最後のウィンドウを表示する手前で止まっていることを示すサイジンググリッパがある場合は、それを含みません。 ただし、ステータスバーコントロールが伸縮性のある1つのウィンドウだけを持つ場合は、ダイアログの幅全体（サイジンググリッパがある場合はそれも含む）が使用されます。

ステータスバーコントロールへのテキスト出力

ステータスバーコントロールにテキストを出力するには、以下の 3 つの方法があります。

1. ウィンドウを含むステータスバーコントロールの場合、テキストを出力するウィンドウの STRING 属性を設定します。
2. ステータスバーコントロール自体のSTRING属性を設定します。これは、ウィンドウを含むステータスバーコントロールで最初の伸縮性のあるウィンドウ（存在する場合）のSTRING属性を設定することと同等です。
3. ダイアログの STATUS-TEXT 属性を設定します。 これは、ダイアログの STATUS-HANDLE 属性によって識別されるステータスバーコントロール（存在する場合）のSTRING属性を設定することと同等です。

多くの場合、ステータスバーコントロールやウィンドウのハンドルを知る必要がないため、最後の方法がメッセージテキストを設定するための一番便利な方法です。

例：

```
DEFINE DATA LOCAL
01 #DLG$WINDOW  HANDLE OF WINDOW
01 #STAT-1      HANDLE OF STATUSBARCTRL
01 #PANE-1      HANDLE OF STATUSBARPANE
END-DEFINE
...
#DLG$WINDOW.STATUS-HANDLE := #STAT-1
...
#PANE-1.STRING := 'Method 1'
...
#STAT-1.STRING := 'Method 2'
...
#DLG$WINDOW.STATUS-TEXT := 'Method 3'
```

Note: ダイアログエディタは、STATUS-HANDLE 属性に最初のステータスバーコントロール（存在する場合）を設定するコードを自動的に生成します。したがって、ステータスバーコントロールをダイナミックに作成している場合、またはダイアログに複数のステータスバーコントロールを定義済みでそれらを切り替える場合は、STATUS-HANDLE 属性を明示的に設定する必要があります。

MDI アプリケーションでのステータスバーの共有

MDI 子ダイアログにはステータスバーコントロールを作成できないため、MDI フレームダイアログに複数のステータスバーコントロールを定義しなくてもよいというのは好都合です。 代わりの方法として、ステータスバーを 1 つだけ定義して、それを子ダイアログ間で共有します。この方法は、以下のようにして実現できます。

1. MDIフレームダイアログで、アプリケーションで使用するすべてのウィンドウを単一のステータスバーコントロールに定義します。
2. すべてのウィンドウを "共有" としてマークします。
3. MDI 子ダイアログが直接アクセスできるように、すべてのウィンドウのハンドルを GDA 内の対応するシャドー変数にエクスポートします。
4. COMMAND-STATUS イベントハンドラで、ダイアログに表示するすべてのウィンドウの VISIBLE 属性を TRUE に設定します。 他のすべてのウィンドウは自動的に非表示になります。

Note: COMMAND-STATUS イベントでは、ダイアログと関連付けられるコマンド（SAME-AS 属性で他のオブジェクトを参照しないシグナル、メニュー項目、またはツールバー項目）の ENABLED 状態も設定する必要があります。設定しないと、これらは自動的に使用不可になります。 ダイアログと関連付けられるコマンドは、MDI フレームのすべての非共有コマンドであり、アクティブな MDI 子（アクティブな MDI 子ダイアログがない場合は MDI フレーム）のすべての共有コマンドです。

ウィンドウ固有のコンテキストメニュー

コンテキストメニューの定義は、ステータスバーコントロールに対して行い、ウィンドウごとには行いません。 ただし、特定のウィンドウを右クリックしたときにのみステータスバーコントロールのコンテキストメニューが表示されるようにする場合、コンテキストメニューをステータスバーコントロールに関連付けると、そのウィンドウ以外でクリックされてもコンテキストメニューの表示を抑制できます。

例：

```
DEFINE DATA LOCAL
01 #CTXMENU-1   HANDLE OF CONTEXTMENU
01 #STAT-1      HANDLE OF STATUSBARCTRL
01 #PANE-1      HANDLE OF STATUSBARPANE
01 #PANE-2      HANDLE OF STATUSBARPANE
01 #PANE-3      HANDLE OF STATUSBARPANE
01 #PANE        HANDLE OF STATUSBARPANE
01 #X (I4)
01 #Y (I4)
END-DEFINE
...
#STAT-1.CONTEXT-MENU := #CTXMENU-1
...
DECIDE ON FIRST *CONTROL
   ...
   VALUE #CTXMENU-1
       DECIDE ON FIRST *EVENT
           ...
             VALUE 'BEFORE-OPEN'
             /* Get click position relative to status bar control
             PROCESS GUI ACTION INQ-CLICKPOSITION WITH
               #STAT-1 #X #Y GIVING *ERROR
             /* Get pane (if any) at specified position
             PROCESS GUI ACTION INQ-ITEM-BY-POSITION WITH
               #STAT-1 #X #Y #PANE
             /* Only show context menu if user clicked in second pane
             IF #PANE = #PANE-2
                #CTXMENU-1.ENABLED := TRUE
             ELSE
                #CTXMENU-1.ENABLED := FALSE
             END-IF
         ...
    END-DECIDE
 ...
END-DECIDE
```

```
...
END
```

Note: ステータスバーのウィンドウに応じて異なるコンテキストメニューを表示する場合は、コンテキストメニューの BEFORE-OPEN イベントでメニュー項目をダイナミックに作成する必要があります。

105 タブコントロールの操作

このchapterでは、次のトピックについて説明します。

タブコントロールの作成

タブコントロールは、他の標準コントロール（リストボックスやプッシュボタンなど）と同じようにダイアログエディタで作成します。 つまり、ダイアログエディタの［挿入］メニューを使用して、あるいは［挿入］ツールバーからのドラッグ & ドロップによってスタティックに作成するか、または TYPE 属性を TABCTRL に設定した PROCESS GUI ACTION ADD ステートメントを使用してランタイム時にダイナミックに作成します。

また、ダイアログウィザードを使用して、タブコントロールを持つダイアログを生成できます。 この場合、このセクションで説明する多くの手法がウィザードによって自動的に適用されます。 明示的な実装は不要で、展開したり 「入力」 したりするだけで、生成された構造が保持されます。 これにより、プログラミングにかかる労力を大幅に減少できます。

タブコントロールには、関連付けられたタブを含めることもまったく含めないこともできます。 タブは、ダイアログエディタでタブコントロールの属性ウィンドウから定義するか、またはランタイム時に TYPE 属性を TABCTRLTAB に設定した PROCESS GUI ACTION ADD ステートメントを実行することによって定義できます。

コントロールのタブへの割り当て

タブコントロールはコンテナです。 ただし、このコントロールの Natural の実装では、個別のタブはコンテナではありません。 ダイアログエディタでタブコントロール内にコントロールを作成すると、そのコントロールの PARENT 属性には、現在アクティブなタブ（存在する場合）のハンドルではなく、タブコントロールのハンドルが自動的に設定されます。 子コントロールを特定のタブに関連付けるには、関連付けるタブのハンドルを子コントロールの OWNER 属性に設定します。 設定後は、タブが非アクティブになるとコントロールが自動的に非表示になり、タブが再度アクティブになると再び表示されます。

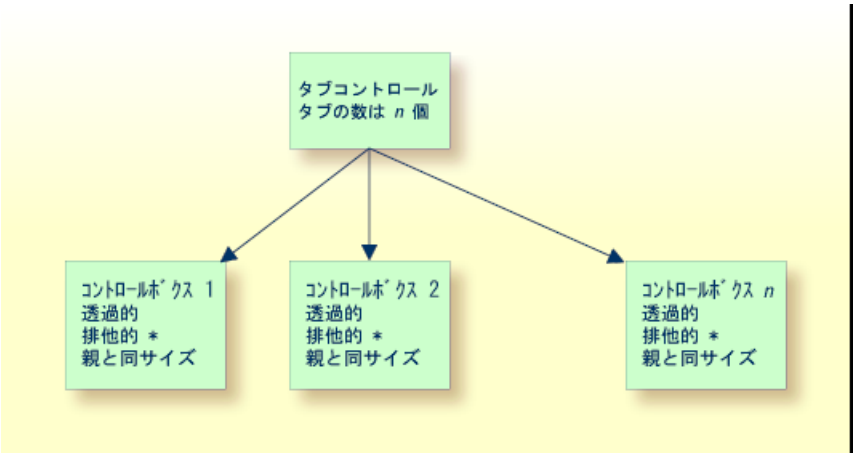
ただし、ダイアログエディタでは、タブコントロールの［UIアクティブ（U）］ STYLE フラグが設定されている場合にのみ（デフォルトでは設定されています）、子コントロールの OWNER 属性が自動的に設定されることに注意してください。 この STYLE フラグが設定されていない場合、子コントロールの OWNER 属性は設定されません（つまり、NULL-HANDLE）。 後者の場合、タブを切り替えても、子コントロールの表示／非表示の切り替えは自動的には行われません。［UIアクティブ（U）］ STYLE は、ランタイム時には効力がないことに注意してください。

タブコントロールページとしてのコントロールボックスの使用

前述のとおり、タブコントロール内の子コントロールはすべて、属しているタブに関係なくタブコントロールを親として持ちます。これでも十分ですが、個別のサブ階層に各タブのコントロールを分けることをお勧めします。

これを実現する最も便利な方法は、タブ「ページ」を表す子コントロールボックスをタブごとに作成することです。その後、他のすべての子コントロールを、対応するコントロールボックスの子コントロールとして作成します。タブコントロールの［UI アクティブ（U）］STYLE フラグが設定されている場合、タブを切り替えるとコントロールボックスの表示／非表示の切り替えが自動的に行われます。したがって、対応する子コントロールの表示／非表示もコントロールボックスと一緒に切り替えられます（祖先のウィンドウが非表示になると、子コントロールは自動的に非表示になります）。そうでない場合、プログラムで明示的にページを切り替える必要があります。これについては、次のセクションで説明します。

以下の図は、コントロールの構成を示しています。



この図のように、各子コントロールボックスは、タブコントロールの背景テクスチャ（存在する場合）が表示されるように、透過にする必要があります。また、タブコントロールのサイズが変更されると必ず直ちにタブコントロールの内部エリアがコントロールボックスによって自動的かつ正確に埋められるように、［親と同サイズ（z）］STYLE を設定する必要があります。また、タブコントロールが UI アクティブでない場合、常に 1 つの子コントロールボックスのみが表示されるように、各コントロールボックスは「排他的」である必要があります。

異なるタブに属するコントロール間の切り替え

Note: このセクションでは、UIアクティブではないタブコントロールについてのみ説明します。 UI アクティブの場合、コントロールの切り替えは Natural によって自動的に実行されます。

ダイアログエディタ上で、現在表示されていない、排他的なコントロールボックスに属するコントロール（またはコントロールボックス自体）を選択すると、ダイアログエディタによって自動的に切り替えが行われます。 ダイアログエディタによって、現在表示されている排他的なコントロールボックス（存在する場合）が非表示にされ（このコントロールボックス内に配置されているコントロールも非表示になります）、選択したコントロールが含まれているコントロールボックスが表示されます（このコントロールボックス内に配置されているコントロールもすべて表示されます）。 このプロセスは、選択方法（明示的、ステータスバーの選択ボックスから、暗黙的、Tab キーでコントロール間を移動してなど）の影響を受けません。

Note: ステータスバーが表示されていない場合、［ツール］メニューの［オプション］コマンドで開く［オプション］ダイアログの［ダイアログエディタ］タブにある［ステータスバー］を設定します。

ランタイム時は、当然ながら、ユーザーによるタブの選択に応じて、対応するコントロールボックスが表示されたり、非表示になったりします。 これは、タブコントロールの CHANGE イベントでアクティブなタブをクエリし、対応するコントロールボックスの VISIBLE 属性を TRUE に設定することによって実現できます。 タブとコントロールボックスを関連付ける方法の 1 つは、ダイアログの AFTER-OPEN イベントで、コントロールボックスのハンドルを対応するタブの CLIENT-HANDLE 属性に設定することです。

次に例を示します。

```
/* Map control boxes to tabs:
#TAB-1.CLIENT-HANDLE := #CTLBOX-1
#TAB-2.CLIENT-HANDLE := #CTLBOX-2
```

```
..
#TAB-N.CLIENT-HANDLE := #CTLBOX-N
```

#CONTROL が HANDLE OF GUI として定義されている場合、タブコントロール（#TABCTRL-1）の CHANGE イベントは次のようになります。

```
/* Get active tab
#CONTROL := #TABCTRL-1.SELECTED-SUCCESSOR
/* Switch to control box belonging to active tab:
#CONTROL := #CONTROL.CLIENT-HANDLE
IF #CONTROL <> NULL-HANDLE
  #CONTROL.VISIBLE := TRUE
END-IF
```

タブ依存コントロールとタブ非依存コントロールの混在

状況によって、現在どのタブが選択されているかに関係なく、コントロールを表示させておきたい場合があります。

これを実現するには、以下の 2 つの方法があります。

1. コントロールボックスを使用している場合、タブコントロールの内部エリアの一部だけを占めるようにコントロールボックスを小さくして、常に表示するコントロールのためのスペースを残すことができます。 この場合、コントロールボックスの［親と同サイズ（z）］STYLE をオフにする必要があります。
2. コントロールボックスを使用せず、タブコントロールがUIアクティブの場合、常に表示する子コントロールを作成している間、タブコントロールの［UI アクティブ（U）］STYLE を一時的にオフにすることにより、常に表示されるコントロールを作成できます。

タブコントロールがUIアクティブでない場合、2階層のコントロールボックスを使用できます。この場合、前述の子コントロールボックスを透過の最上位コントロールボックス（タブコントロールの子として作成）の子コントロールとして作成します。 これにより、（［親と同サイズ（z）］フラグが設定されていない）最上位コントロールボックスで適切な位置とサイズを指定して、置換可能なリージョンを定義できます。

Note: これは、ダイアログウィザードに使用されている手法によく似ています。 詳細については、「*コントロールボックスの操作*」を参照してください。

キーボードナビゲーション

タブコントロール内のタブ間をキーボードで移動するには、以下の3つの方法があります。これらの方法は同時に組み合わせて使用できます。

1. タブコントロールの［ブラウズ可能（z）］STYLE フラグが設定されている場合、そのタブコントロールはタブシーケンスに含まれています（つまり、Tab キーで移動できます）。タブコントロールがフォーカスを受け取ると、矢印キーでタブ間を移動できます。 この場合、最初のタブと最後のタブとの間は 「ラップアラウンド」 されません。
2. タブのキャプション（STRING 属性）には、次の文字がニーモニック文字であることを示すアンパサンド記号（&）を含むことができます。 Alt キーを押したままニーモニック文字を押すと、該当するタブが選択されます。これにより、希望するタブにキーボードで「直接」移動できます。
3. ダイアログの［プロパティシート（p）］STYLE が設定されている場合、キーボードショートカットの Ctrl + Tab および Ctrl + Shift + Tab を使用して、それぞれ次のタブおよび前のタブにラップアラウンド付きで移動できます。

最後の手法は、タブコントロールまたはタブコントロール内のダイアログエレメントにフォーカスがなくても実行できることに注意してください。フォーカスのあるコントロールから始まり、1 つ以上のタブコントロールを持つコンテナが見つかるまで、Natural によって各コンテナ（祖先）が確認されます。 見つかったコンテナにタブコントロールが 1 つだけ含まれている場合、キーボードショートカットはこのタブコントロールに適用されます。 複数のタブコントロールが含まれている場合、これらのショートカットは無効になります。ダイアログにタブコントロールが含まれていない場合、ショートカットは、［プロパティシート（p）］STYLE が設定されていなかったかのように、通常の機能を実行します。

キーの組み合わせ Ctrl + Tab および Ctrl + Shift + Tab のデフォルトの処理は、ACCELERATOR 属性で再定義することによって上書きできることに注意してください。

タブの切り替えイベント

タブの切り替えが実行されるたびに（実行したのがユーザーとプログラムのどちらであっても）、以下の一連のイベントが発生します。

1. 現在選択されているタブが LEAVE イベントを受け取ります（抑制されていない場合）。 このイベントは通常、タブページ上のデータを検証および／または確定するために使用します。
2. タブコントロールの MODIFIABLE 属性が確認されます。 この属性が FALSE に設定されている場合、タブの切り替えは実行されません。現在選択されているタブが選択されたままになり、これ以上イベントは発生しません。LEAVE イベント中に実行したデータ検証でエラーがあり、処理を続行する前にユーザーに訂正してもらう必要がある場合、これは有効です。

3. 現在選択されているタブを OWNER として持つ、すべての直接的な子コントロール（存在する場合）が自動的に非表示になります。
4. 新しいタブが選択されます。
5. 新しく選択されたタブを OWNER として持つ、すべての直接的な子コントロール（存在する場合）が、VISIBLE 属性が TRUE に設定されている場合、自動的に表示されます。
6. 新しく選択されたタブが ENTER イベントを受け取ります（抑制されていない場合）。このイベントは通常、新しいタブページ上のコントロールを初期化するために使用します。
7. タブコントロールが CHANGE イベントを受け取ります。これは、各タブのイベントハンドラを変更せずにタブの切り替えを追跡し、対応するのに便利です。

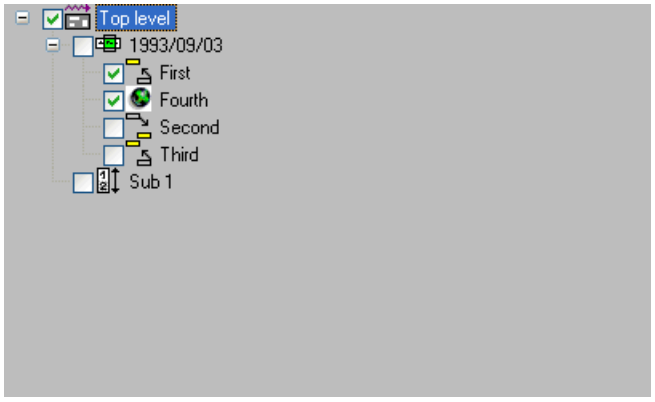
コントロールの作成時は、選択されているタブは最初の ENTER イベントを受け取らず、タブコントロールも最初の CHANGE イベントを受け取らないことに注意してください。また、タブコントロールを含むダイアログが閉じられるときは、現在選択されているタブは LEAVE イベントを受け取りません。

106 ツリービューコントロールの操作

このchapterでは、次のトピックについて説明します。

はじめに

ツリービューコントロールは、階層形式でデータを表示するために使用します。階層内の各ノードは、ツリービュー項目として内部的に表されます。以下の図は、7つのツリービュー項目を持つ3つの階層を表示する、簡単なツリービュー（任意のチェックボックス付き）です。



上記のツリービューコントロールには、［+／-ボタン（b）］、［行（l）］、［ルート行（r）］、および［チェックボックス（c）］の各 STYLE フラグが設定されています。

階層間の各項目の高さとインデントは、それぞれ ITEM-H 属性および SPACING 属性で設定できます。 これらのどちらかがゼロの場合、デフォルト設定が使用されます。

項目イメージの設定

ツリービュー項目のイメージは、「*イメージリストコントロールの操作*」で説明しているように、ツリービューコントロールにイメージリストコントロールを作成および関連付け、（項目ごとに）必要なイメージを、インデックスおよび／またはイメージハンドルを使用してイメージリストから選択することによって定義できます。

ツリービューコントロールでは小さいイメージしか使用されないため、リストビューコントロールと同じイメージリストを使用していない限り、イメージリストコントロールの［大きいイメージ（L）］スタイルを設定する必要はないことに注意してください。

項目の選択

リストビューコントロールとは異なり、ツリービューコントロールで選択できる項目は1つのみです。 現在の選択項目（存在する場合）は、ツリービューコントロールの（読み取り専用の）SELECTED-SUCCESSOR 属性をクエリすることによって取得できます。

また、ユーザーによる選択の他に、プログラムで項目の SELECTED 属性を設定したりクリアしたりすることにより、項目を選択したり選択を解除したりできます。

項目が選択されると、ツリービューコントロールの CLICK イベントが発生します（抑制されていない場合）。このとき、対応する項目のハンドルがコントロールの ITEM 属性に設定されます。

項目の有効化

項目をダブルクリックすると、ツリービューコントロールの ACTIVATE イベントが発生します（抑制されていない場合）。 このイベントをアプリケーションで制御する場合、通常は選択された項目上でユーザー定義のデフォルトのアクションを実行します。この項目のハンドルは、ツリービューコントロールの ITEM 属性または SELECTED-SUCCESSOR 属性で取得できます。 選択された項目に対し、デフォルトでない他のアクションも使用できることに注意してください。ただし、これらのアクションは通常、他のメカニズムによってアクセスされます。 例えば、これらはアプリケーションが表示するコンテキストメニューに（通常、デフォルトのアクションと一緒に）リストされます。

［ダブルクリック展開（d）］STYLE フラグが設定されている場合、子項目を持つツリービュー項目をダブルクリックすると、子項目が展開されアクティブ化されます。

また、ACTIVATE イベントは、キーボードを使用してトリガすることもできます。 このためには、次の 2 つの方法のいずれかを使用します。

1. ツリービューコントロールの ACCELERATOR 属性に定義されているキーまたはキーの組み合わせを押す。 現時点で、コントロールにフォーカスがなくてもかまいません。
2. ツリービューコントロールにフォーカスがある場合、Enter キーを押す。 この方法は、ダイアログにデフォルトのプッシュボタンが含まれておらず、プッシュボタンに［OK ボタン (O)］の STYLE フラグが設定されていない場合にのみ機能します。

どちらの場合も、項目が選択されていないと ACTIVATE イベントは発生しません。

項目データ

適切にソートできるようにするために（次のセクションを参照）ツリービュー項目のデータを英数字にする必要はありません。デフォルトは英数字ですが、項目の FORMAT 属性でサポートされている定義済みのタイプを使用できます。 また、項目の EDIT-MASK 属性を設定することによって、項目に編集マスクを適用できます。 ユーザーに表示される項目のラベルは、関連付けられている編集マスク（存在する場合）が適用された、項目の内部データの英数字表記になります。編集マスクが使用されている場合、項目の内部データと表示データ間の変換は、MOVE EDITED ステートメントと互換性があります。 それ以外の場合、内部データと表示されたデータ間およびその逆の変換は、データを Natural スタックにコピーしたり、Natural スタックからコピーする場合に発生する変換と互換性があります（STACK TOP DATA および INPUT ステートメントを参照）。 例えば、数値は、現在の小数点文字を使用して（DC パラメータで定義されたように）表示されます。必要に応じて、負の場合は先頭にマイナスを付けることができます。日付値は、DTFORM パラメータで定義されたフォーマットで表示されます。論理値は、真の場合は "X" として、偽の場合は空白として表示されます。

英文字以外のツリービュー項目の使用例を以下に示します。

```
PROCESS GUI ACTION ADD WITH
PARAMETERS
   HANDLE-VARIABLE = #TVITEM-DATE
   TYPE = TREEVIEWITEM
   PARENT = #TV-1
   STRING = '+123.456'
   FORMAT = FT-DECIMAL
   EDIT-MASK = '+ZZZ,ZZ9.999'
END-PARAMETERS GIVING *ERROR
```

この場合、指定した項目ラベル（STRING 属性値）は、指定した編集マスク（+ZZZ,ZZ9.999）と互換性のあるフォーマットの有効な数値である必要があります。 ラベルは内部的に、指定したフォーマット FT-DECIMAL（= P10.5）に対応するデータタイプと長さに変換されます。

ツリービュー項目の基となるデータには直接アクセスできないことに注意してください。 項目のデータは、項目のラベル経由でのみ設定または取得できます。

ソート

ツリービュー項目は、SORT-ITEMS アクションを呼び出すことにより、ソートされた順序で挿入することも、挿入後に明示的にソートすることもできます。 ツリービューコントロールまたは挿入するツリービュー項目の SORTED 属性が TRUE に設定されている場合、項目はアルファベットの昇順で挿入されます。 挿入後にソートする場合、内部データに応じて、項目は任意で昇順または降順でソートされます（最後のセクションを参照）。 詳細については、SORT-ITEMS アクションのドキュメントを参照してください。ソート対象の項目の基本となるフォーマット（FORMAT 属性で定義）を比較可能なタイプに設定するのは、プログラマの責任であることに注意してください。 例えば、整数値と浮動小数点値は一緒に使用できますが、整数値と日付値は一緒に使用できません。

ラベル編集

ツリービューコントロールのラベル編集プロセスは、リストビューコントロールと同じです。このため、この内容の詳細については、「*ツリービューおよびリストビューコントロールでのラベル編集*」を参照してください。

SORTED 属性が設定されている場合でも、ラベル編集の操作が完了した後は、項目の順序付けが自動的に再実行されることはありません。 これが必要な場合は、以下の例に従って操作してください。 まず、後で使用する変数をいくつか定義します。

```
01 #CONTROL HANDLE OF GUI 01 #ITEM HANDLE OF TREEVIEWITEM
01 #SORTITEM HANDLE OF TREEVIEWITEM
```

また、ツリービューコントロールの名前は #TV-1 であるとします。

ツリービューコントロールの AFTER-EDIT イベントでは、（まだ完了していない）編集プロセスに影響があるため、非同期で順序付けを再実行できません。 代わりに、#SORTITEM 変数を設定し、編集プロセスの完了後に順序付けを再実行するよう指定します。この処理は、ツリービュー項目をソートする場合にのみ実行する必要があります。

```
#CONTROL := *CONTROL #ITEM := #CONTROL.ITEM IF #CONTROL.SORTED OR #ITEM.SORTED
#SORTITEM := #ITEM ELSE #SORTITEM := NULL-HANDLE END-IF
```

この例では、ツリービューコントロール自体が提供しているデフォルトのソート（アルファベットの昇順）を使用することが想定されていることに注意してください。

項目の順序付け再実行の実際の作業は、ダイアログの IDLE イベントで非同期的に行われます。

```
IF #SORTITEM <> NULL-HANDLE PROCESS GUI ACTION SORT-ITEMS WITH #SORTITEM.PARENT
GIVING *ERROR RESET #SORTITEM END-IF
```

マルチプルコンテキストメニュー

コントロールに単一のコンテキストメニューのみを使用する場合、表示するコンテキストメニューのハンドルをコントロールのCONTEXT-MENU属性に設定し、そのままにしておきます。ただし、ツリービューコントロールに複数のコンテキストメニューを表示する必要があることがよくありますが、このアプローチでは柔軟性が足りません。

上記の問題に対応するために、CONTEXT-MENU イベントが導入されました（上記の同じ名前の属性と混同しないように注意してください）。 このイベント（無効にされていない場合）は、CONTEXT-MENU属性が評価される直前にターゲットのコントロールで発生します。これによって、アプリケーションは最初に、適切なコンテキストメニューのハンドルにこの属性をダイナミックに設定できます。

例として、ダイアログエディタで2つのコンテキストメニューを定義したとします。1つは項目に関連するコマンドが組み込まれている #CTXMENU-ITEMS で、もう 1 つは汎用コマンド（表示モードの切り替えなど）が組み込まれている #CTXMENU-DEFAULT です。 この場合、次の CONTEXT-MENU イベントを使用することができます。

```
#CONTROL := *CONTROL IF #CONTROL.SELECTED-SUCCESSOR
<> NULL-HANDLE #CONTROL.CONTEXT-MENU := #CTXMENU-ITEMS ELSE #CONTROL.CONTEXT-MENU
:= #CTXMENU-DEFAULT END-IF
```

ローカルデータの定義は以下のように想定されています。

```
01 #CONTROL HANDLE OF GUI
```

この例では、項目が占めている位置をユーザーが右クリックすると、コンテキストメニュー #CTXMENU-ITEMS が表示されます。それ以外の場合は、#CTXMENU-DEFAULT が表示されます。

この手法をさらに活用して、選択された項目のタイプに固有のコンテキストメニューを表示することもできます。

項目のダイナミックな作成

ツリービュー項目が展開または圧縮されると、イベントが抑制されていなければ、コントロールの EXPAND イベントまたは COLLAPSE イベントがそれぞれ発生します。 多数あるイベントの中のこれらのイベントにより、ツリービュー項目をオンデマンドでダイナミックに追加したり削除したりできます。

例えば、以下のコードは、EXPAND イベントに応答して 3 つの項目をダイナミックに作成する方法を示しています。このコードでは、項目を入力する位置に、空の STRING 属性値を持つダミーのプレースホルダ項目がすでに配置されているものとしています。 プレースホルダは、ダイアログエディタでスタティックに定義することも、ツリービュー項目を最初に作成するときにダイナミックに定義することもできます。 プレースホルダの目的は、"+" ボタン（［+／- ボタン（b）］ STYLE によってボタン表示が有効に設定されている場合）が親ノードの隣に確実に表示されるようにすることです。

以下の EXPAND イベントコードでは、変数 #TGT-ITEM にツリービュー項目のハンドルが設定されているものとしています。このツリービュー項目の下に、ツリービュー項目がダイナミックに作成されます。

```
#CONTROL := *CONTROL
#ITEM := #CONTROL.ITEM
IF #ITEM = #TGT-ITEM
  #TVITEM-DYN := #ITEM.FIRST-CHILD
  IF #TVITEM-DYN <> NULL-HANDLE AND
     #TVITEM-DYN.STRING = ' '
    PROCESS GUI ACTION DELETE WITH #TVITEM-DYN GIVING *ERROR
    FOR #I 1 3
      COMPRESS 'Dynamic Item' #I INTO #A
      PROCESS GUI ACTION ADD WITH PARAMETERS
        HANDLE-VARIABLE = #TVITEM-DYN
        TYPE = TREEVIEWITEM
        PARENT = #ITEM
        STRING = #A
      END-PARAMETERS GIVING *ERROR
    END-FOR
```

```
  END-IF
END-IF
```

ローカルデータの定義は以下のように想定されています。

```
01 #CONTROL HANDLE OF GUI
01 #ITEM HANDLE OF TREEVIEWITEM
01 #TVITEM-DYN HANDLE OF TREEVIEWITEM
01 #TGT-ITEM HANDLE OF TREEVIEWITEM
01 #I (I4)
01 #A (A) DYNAMIC
```

上記のコードではまず、展開する項目がターゲット項目かどうかを確認します。 ターゲット項目の場合、最初の子の STRING 属性をクエリして、プレースホルダが存在するかどうかを確認します。プレースホルダが存在する場合、そのプレースホルダは削除され、3つのツリービュー項目がダイナミックに作成されます。 挿入される項目の STRING 属性値は、SORTED 属性が設定されたツリービューでもコードが正しく動作するように、後で変更するのではなく、作成時に設定します。

リソースを節約するために、ダイナミックに作成した項目を COLLAPSE イベントハンドラ内で削除して、別のプレースホルダ項目で置き換えることもできます。

```
#CONTROL := *CONTROL
#ITEM := #CONTROL.ITEM
IF #ITEM = #TGT-ITEM
  PROCESS GUI ACTION DELETE-CHILDREN WITH #ITEM GIVING *ERROR
  PROCESS GUI ACTION ADD WITH PARAMETERS /* placeholder
    TYPE = TREEVIEWITEM
    PARENT = #ITEM
  END-PARAMETERS GIVING *ERROR
END-IF
```

ドラッグ & ドロップ

ドラッグ & ドロップをサポートするための基本的な手法については、「*クリップボードおよびドラッグ & ドロップの使用*」に記載されています。

マウスカーソルの下にある項目（存在する場合）の SELECTED 属性を（必要に応じて）設定して強調表示し、後で元の選択状態（設定されていた場合）に戻すのは、プログラマの責任であることに注意してください。

以下の例は、ラベルを変更するための、別のアプリケーション（ワードパッドなど）からツリービュー項目へのテキストのドラッグ & ドロップをサポートする、ドロップターゲットとして動作するツリービューコントロールのコードを示しています。

まず最初に、ドロップモードをツリービューコントロールに適切に設定します。 ダイアログエディタの［ツリービューコントロール属性］ウィンドウで、［ドロップモード］選択ボックスを［コピー+移動］に設定します。 これにより、ダイアログの生成ソースコード内で、コントロールの DROP-MODE 属性が DM-COPYMOVE に設定されます。

次に、以下で使用される必要なローカルの変数を定義する必要があります。

```
01 #CONTROL HANDLE OF GUI
01 #DROP-ITEM HANDLE OF GUI
01 #ITEM HANDLE OF GUI
01 #SELECTED HANDLE OF GUI
01 #AVAIL (L)
01 #X (I4)
01 #Y (I4)
```

これを行った後、必要なイベントハンドラを記述することができます。 処理を開始する論理的な場所は、DRAG-ENTER イベントです。

```
#CONTROL := *CONTROL
#CONTROL.CLIENT-HANDLE := #CONTROL.SELECTED-SUCCESSOR
PROCESS GUI ACTION INQ-FORMAT-AVAILABLE WITH CF-TEXT #AVAIL GIVING *ERROR
IF #AVAIL
  #CONTROL.SUPPRESS-DRAG-DROP-EVENT := NOT-SUPPRESSED
ELSE
  #CONTROL.SUPPRESS-DRAG-DROP-EVENT := SUPPRESSED
END-IF
```

上記のコードではまず、後で項目の選択状態を元に戻すために、現在選択されている項目のハンドルをコントロールの CLIENT-HANDLE 属性に保存します。 その後、イベントハンドラはドラッグドロップクリップボードでテキストを使用できるかどうかを確認します。 使用できる場合、ドロップを可能にするために DRAG-DROP イベントの抑制が解除されます。 それ以外の場合、このイベントを無効にしてドロップを禁止し、ドロップアンドドラッグカーソルが表示され「ない」ようにします。

外部データをツリービューコントロール上にドラッグしているときにドロップの強調表示を行うには、DRAG-OVER イベントハンドラを使用します。

```
#CONTROL := *CONTROL
IF #CONTROL.SUPPRESS-DRAG-DROP-EVENT = NOT-SUPPRESSED
  PROCESS GUI ACTION INQ-DRAG-DROP WITH
    3X #X #Y GIVING *ERROR
  PROCESS GUI ACTION INQ-ITEM-BY-POSITION WITH
    #CONTROL #X #Y #ITEM GIVING *ERROR
  IF #ITEM <> #DROP-ITEM
    #DROP-ITEM := #ITEM
```

```
    IF #DROP-ITEM <> NULL-HANDLE
      #DROP-ITEM.SELECTED := TRUE
    END-IF
  END-IF
END-IF
```

上述したコードにより、次の処理が実行されます。

1. DRAG-ENTER イベントでドロップが許可されていない場合、ドロップの強調表示は不要なため、このイベントは無視されます。
2. ドロップが許可されている場合、INQ-DRAG-DROP アクションを呼び出して現在のドロップ位置を把握します。
3. INQ-ITEM-BY-POSITION アクションを使用して、現在のドロップ位置にあるツリービュー項目（存在する場合）を把握します。 この項目を #DROP-ITEM に保存します。
4. カーソルの下にあるツリービュー項目（存在する場合）の SELECTED 属性を TRUE に設定して、この項目を選択し、ドロップの強調表示を行います。

実際のドロップを実行するために、DRAG-DROP イベントハンドラが提供されています。

```
IF #DROP-ITEM <> NULL-HANDLE
  PROCESS GUI ACTION GET-CLIPBOARD-DATA WITH CF-TEXT #DROP-ITEM.STRING
    GIVING *ERROR
END-IF
#CONTROL := CONTROL  /* "parameter" for subroutine below
PERFORM RESET-SELECTION
```

ドロップターゲットのツリービュー項目が存在する場合、上記のコードはドラッグドロップクリップボードからテキストを取得して項目のキャプションに直接設定します。 その後、ツリービューコントロールの選択状態を元に戻します。 このためには、以下に記述するコードのように、RESET-SELECTION サブルーチンを使用します。

```
#ITEM := #CONTROL.CLIENT-HANDLE
IF #ITEM <> NULL-HANDLE
  /* Restore original selection:
  #ITEM.SELECTED := TRUE
ELSE
  /* Nothing was originally selected,
  /* so clear any existing selection:
  #ITEM := #CONTROL.SELECTED-SUCCESSOR
  #ITEM.SELECTED := FALSE
END-IF
RESET #DROP-ITEM
```

他のイベントハンドラのロジックと合わせるために、#DROP-ITEM もリセットして、ドロップ項目がないことを示します。

最後に、ユーザーがドラッグ操作をキャンセルした場合、またはドロップを実行せずにツリービューコントロールの境界線の外に出た場合は、DRAG-LEAVE イベントハンドラを使用します。

```
#CONTROL := CONTROL  /* "parameter" for subroutine below
PERFORM RESET-SELECTION
```

上記のコードは、ドロップの強調表示（行われている場合）をクリアして元の選択状態に戻すために、前述のインラインサブルーチンを呼び出しているだけです。

107 ダイナミックな情報行およびステータスバーの操作

ダイナミック情報行（DIL）とステータスバーの両方を使用する場合、DIL-TEXT 値と STATUS-TEXT 値のどちらがいつ表示されるかは、HAS-DIL 属性および HAS-STATUS-BAR 属性の組み合せによって決まります。

HAS-DIL	HAS-STATUS-BAR	DIL-TEXT	STATUS-TEXT
TRUE	TRUE	表示	表示
TRUE	FALSE	-	-
FALSE	TRUE	-	表示
FALSE	FALSE	-	-

HAS-DIL と HAS-STATUS-BAR の両方が TRUE の場合、DIL-TEXT 値と STATUS-TEXT 値は、どちらが最後に変更されたかに応じて一方が他方を書き換えます。

108 ［最大化］／［最小化］／［システム］ボタンの追加

手順 108.1. ダイアログに［最大化］／［最小化］／［システム］ボタンを追加するには

■ ［ダイアログ属性］ウィンドウを開きます。［システムボタン］、［最大化］、または［最小化］の各エントリをチェックします。

［システムボタン］エントリをチェックすると、ダイアログの標準コントロールメニューを使用できます。 標準コントロールメニューには、コントロールメニューボックス（ダイアログを閉じるため）、タイトルバー、最大化ボタン、および最小化ボタンが含まれます。

109 カラーの定義

ダイアログおよびダイアログエレメントに対してカラーを定義できます。 カラーには、前景色と背景色があります。 カラーを定義するには、以下の属性を使用します。

- BACKGROUND-COLOUR-NAME
- BACKGROUND-COLOUR-VALUE
- FOREGROUND-COLOUR-NAME
- FOREGROUND-COLOUR-VALUE

NAME で終わる属性には標準色のみを割り当てることができます。 一方、VALUE で終わる属性には RGB モデルに従ってカスタマイズしたカラーを割り当てることができます。

カラーを設定するには、以下の 2 つの方法があります。

- 属性ウィンドウで設定する。
- イベントハンドラコードで設定する。

NAME で終わる属性には値を直接割り当てることができます。 VALUE で終わる属性に値を割り当てる場合は、NAME 属性に値 CUSTOM を設定する必要があります。 NAME 属性に値 CUSTOM を設定しない場合、VALUE 属性は無視されます。

例：

```
#DIA.BACKGROUND-COLOUR-NAME:= MAGENTA     /* Assign a value to a NAME
                                          /* attribute
#DIA.BACKGROUND-COLOUR-NAME:= CUSTOM      /* Set NAME to CUSTOM
#DIA.BACKGROUND-COLOUR-VALUE:= H'FF0000'  /* Then assign Red, Green, and
```

```
                                        /* Blue values to the VALUE
                                        /* attribute (hexadecimal)
```

Note: ユーザーインターフェイスのすべての部分ですべてのカスタマイズしたカラーを使用できるわけではありません。 例えば、テキストのカラーは常に白黒である必要があります。

属性ウィンドウでカラーを設定するときは、以下の 3 つの指定方法があります。

- NAME で終わる属性の値を DEFAULT にしておきます。 コードでもこのようにできます。 これにより、ウィンドウシステム内のカラー設定によってカラーが決まります。
- リストボックスの NAME で終わる属性を使用して、事前定義されたカラーの 1 つを選択します。
- VALUE で終わる属性を使用することによって、カスタマイズされたカラーを定義します。

手順 109.1. カラーを定義するには

1 ［背景色］エントリの右にある［カスタム］プッシュボタンコントロールを選択します。 ダイアログボックスが表示されます。

2 事前定義されたカラーの1つを選択するか、または［色の作成］プッシュボタンコントロールを選択します。 赤、緑、青の値を設定するには、カーソルを使用して希望する色を選択するか、または赤、緑、青の値表示フィールドに 1～253 の値を入力します。

3 ［色の追加］プッシュボタンコントロールを選択します。 ダイアログボックスの［**OK**］ボタンをクリックすると、新規定義したカラーが保存されます。 これで、新規定義したカラーがデフォルトで選択されます。

4 属性ウィンドウを閉じると、新規定義したカラーが設定されます。

110 特定のフォントのテキストの追加

手順 110.1. ダイアログエレメントに割り当てるテキスト（例：プッシュボタンコントロールのキャプション）に特定のフォントを指定するには

1 ダイアログエレメントの属性ウィンドウを使用します。

2 ［フォント］エントリの右にある［...］プッシュボタンを選択します。ダイアログボックスが開きます。

3 使用可能なフォントのリストから、フォントタイプ（例：［**Times New Roman**］）を選択します。

4 フォントタイプに使用可能なスタイルのリストから、フォントスタイル（例：［斜体］）を選択します。

5 フォントタイプおよびスタイルに使用可能なサイズのリストから、フォントサイズ（例：［**10**］）を選択します。 選択したフォントのサンプルが表示されます。

6 属性ウィンドウを閉じると、選択したフォントが設定されます。

Note: ダイアログエレメントに中央揃えまたは右揃えのテキストを追加すると、ダイアログエレメントの RECTANGLE-H 属性にその要素の最小の高さ（4 ポイントのフォント - 高さ 8、8 ポイントのフォント - 高さ 22、12 ポイントのフォント - 高さ 24）が設定されます。

また、ダイアログエディタでは、ダイアログ属性ウィンドウでダイアログ全体に1つのフォントを指定できます。 このフォントは FONT-STRING 属性に定義され、ダイアログとその子すべてに有効です。 ダイアログ全体にフォントを指定する主な利点は、指定したフォントがダイアログレイアウトにとって大き過ぎるか小さ過ぎる場合、ダイアログのすべての子ではなく FONT-STRING 属性だけを変えればよいということです。

FONT-STRING 属性は、最初は、ダイアログを PROCESS GUI ステートメントの ADD アクションで作成するときにパラメータとして設定する必要があります。 ダイアログ内のダイアログエレメントのテキストに特定のフォントが割り当てられていない場合は、このテキストが FONT-STRING で指定されたフォントで表示されます。

111 オンラインヘルプの追加

ダイアログエディタで作成されたアプリケーションから、指定されたヘルプトピック ID に対するヘルプを起動できます。 ヘルプトピック ID に関連付けるヘルプ部分は、Natural 開発環境外で作成する必要があることに注意してください。 また、プラットフォーム固有のヘルプコンパイラを使用してコンパイルする必要もあります。

さまざまなヘルプセクションすべての概要をアプリケーションに維持するため、Naturalにはヘルプオーガナイザが用意されています。 このオーガナイザを使用すると、次の操作を実行できます。

- ヘルプ ID（HELP-ID 属性値）を特定のダイアログエレメントに割り当てる。
- ヘルプトピックに関連付けるヘルプテキストを記述する。このテキストは、ヘルプコンパイラによって処理される *.rtf* ファイルに変換されます。
- ヘルプトピックのキーワードを定義する（オプション）。
- ヘルプコンパイラマクロをヘルプトピックに割り当てる（オプション）。
- 任意で、内部ドキュメントのためのコメントを追加する。

手順 111.1. ヘルプトピックを作成するには

1 ヘルプオーガナイザのメインダイアログを起動する。
2 特定のダイアログエレメントを選択する。
3 新しいヘルプトピック ID を生成する。
4 ヘルプオーガナイザのメインダイアログに戻る。
5 生成されたヘルプトピック ID を割り当てる。
6 ヘルプトピックテキストやトピック名など、ヘルプトピック ID に対する外部定義を入力する。
7 ヘルプオーガナイザのメインダイアログに戻る。

8 トピックリストに移動し、この新しいヘルプトピックが作成するヘルプファイルの一般的な編成に適しているかどうかを確認する。

9 ヘルプオーガナイザのメインダイアログに戻る。

10 すべてを保存する。

ダイアログまたはダイアログエレメントに、ヘルプオーガナイザとは関係なく HELP-ID 値を割り当てることもできます。

手順 111.2. ヘルプオーガナイザとは関係なく HELP-ID 値を割り当てるには

■ 対応する属性ウィンドウを開きます。［ヘルプ **ID**］エントリに値を入力します。

ヘルプトピックの *.h* ファイルを使用して、ここで入力する数値 ID を、対応するヘルプトピック ID（*.hlp* ファイルのマークアップで作成されます）にマッピングします。

Natural は、ヘルプファイルが現在のライブラリのリソース（RES）サブディレクトリ、STEPLIB の 1 つのリソース（RES）サブディレクトリ、または環境変数 NATGUI_BMP が指すディレクトリにあることを想定しています。 デフォルトでは Natural は現在のライブラリと同じ名前のヘルプファイルを検索しますが、HELP-FILENAME 属性を使用してヘルプファイルの名前を明示的に設定できます。 ファイル拡張子を指定しないと、Natural はまずコンパイルされた HTML ヘルプファイル（拡張子 ".chm"）を探し、見つからなければ次に WinHelp ヘルプファイル（拡張子 ".hlp"）を探します。 このように、ファイル拡張子を指定しなければ、Natural プログラムを変更せずに、使用するファイルを WinHelp から HTML ヘルプに変えることができます。ただし、ヘルプオーガナイザは WinHelp だけをサポートすることに注意してください。 HTML ヘルプ内容を作成する場合は、外部のヘルプ作成ツールを使用する必要があります。

アクティブなダイアログでエンドユーザーが F1 キーを押すと、Natural はまず HELP-FILENAME 属性値と拡張子 ".h*nn*"（*nn* は Natural 言語コード）でファイルを探します。 このようなファイルが見つからない場合、次は HELP-FILENAME 属性値と拡張子 ".hlp" でファイルを探します。

ダイアログエレメントにフォーカスがあるときにエンドユーザーが F1 キーを押すと、Natural はそのヘルプ ID を探します。

Note: アプリケーションにオンラインヘルプを追加する場合、各ダイアログに HELP-ID 番号を割り当て、ダイアログのためのヘルプテキストを記述することをお勧めします。 エンドユーザーが、HELP-ID が割り当てられていないダイアログエレメントを選択して、ヘルプを要求するために F1 キーを押すと、現在のダイアログのヘルプが表示されます。 Natural は、ダイアログエレメントに HELP-ID が割り当てられていないと、ダイアログエレメントの親、つまりダイアログが HELP-ID を持っているかどうかを調べます。 ダイアログが HELP-ID を持っていないと、次はダイアログの親、つまり 1 レベル上位のダイアログが HELP-ID を持っているかどうかを調べるというように、最上位のダイアログに達するまでこの作業を繰り返します。

手順 111.3. ヘルプファイルを構築するには

1 コマンドプロンプトの画面に変更します。

2 環境変数 `$NATGUI_BMP` が指すディレクトリに移動します。

3 コマンド `HCRTF -X helpfilename` を発行します。

Note: これは、*HCRTF.EXE* を含むディレクトリが `PATH` 環境変数に指定されているものとします。

手順 111.4. ヘルプファイルをテストするには

1 アプリケーション内のダイアログを起動します。

2 F1 キーを押します。

ダイアログに対するヘルプトピックが表示されます。

別の方法としてヘルプコンパイラワークショップ（*HCW.EXE*）で *.hpj* ファイルを開くことによって、ヘルプファイルを簡便に作成して対話的にテストできます。

手順 111.5. ポップアップウィンドウにヘルプを表示するには

1 ダイアログ属性ウィンドウで［ポップアップヘルプ］オプションをチェックします。

2 ダイアログを実行します。

3 ヘルプ ID が関連付けられているコントロールにカーソルを置いて F1 キーを押します。

カーソルが置かれたコントロールに関連付けられたヘルプトピックがポップアップウィンドウに表示されます。

112 ニーモニックキーおよびアクセスキーの定義

このchapterでは、次のトピックについて説明します。

はじめに

キーボードコマンドを提供するには、以下の 2 つの方法があります。

- ニーモニックキーは、下線付き文字によってメニュー項目などの表示可能ダイアログエレメントに定義されます。エンドユーザーは、「Alt+ニーモニックキー」（例：Alt+A）を押すことによってメニュー項目を選択できます。
- アクセスキーは ACCELERATOR 属性に定義されます。 使用可能なダイアログエレメントであれば表示可能かどうかに関係なく、エンドユーザーはこのキーを押すことによって、ダイアログエレメントに対するダブルクリックイベントまたはクリックイベントを起こすことができます。

ニーモニックキーの定義

ニーモニックキーは、希望する文字の前に "&" を付けたものをダイアログエレメントの STRING 属性に指定することによって定義します。 ランタイム時にこの文字が下線付きで示されます。 例えば、STRING 属性値 "E&xplanation" は、ランタイム時に以下のように表示されます。

Explanation

テキスト定数コントロールまたはグループフレームコントロールにニーモニックキーを定義した場合、ランタイム時にニーモニックキーが押されると、コントロールシーケンスの次のダイアログエレメントがフォーカスを得ます。 例えば、テキスト定数コントロールの次のダイアログエレメントが入力フィールドコントロールの場合、テキスト定数コントロールのニーモニックキーが押されると入力フィールドコントロールがフォーカスが移動します。 ランタイム時にこの入力フィールドコントロールを使用不可にする場合は、対応するテキスト定数コントロールも使用不可にする必要があります。

ニーモニックキーは、グループフレームコントロール、メニュー項目、プッシュボタンコントロール、ラジオボタンコントロール、テキスト定数コントロール、トグルボタンコントロール、ツールバー項目の各タイプのダイアログエレメントの STRING 属性に定義できます。

STRING 属性に "&&" と指定すると、ランタイム時に "&" も表示できます。 例えば、"A&&B" は "A&B" と表示されます。

Note: 最新の Windows バージョン（Windows 2000 など）のデフォルトでは、Alt キーが押されるまでニーモニック文字に下線が付けられません。 ただし、この新しい動作は、常にニーモニック文字に下線が付けられるようにユーザーによって変更できます。 例えば、英語版の Windows 2000 では、［**Start**］、［**Control Panel**］、［**Display**］、

［**Effects**］の［**Hide Keyboard navigation indicators until I use the Alt key**］オプションをオフにすることによって、これを実現できます。

アクセスキーの定義

アクセスキーは、F6 や Ctrl+1 のように、ダイアログエレメントに対するキーまたはキーの組み合せを ACCELERATOR 属性に設定することによって定義します。 エンドユーザーがアクセスキーを押すと、ダイアログエレメントに対するダブルクリックイベントか、ダブルクリックイベントがない場合はクリックイベントが発生します。 対応するイベントが抑制されているか、またはダイアログエレメントが使用不可の場合は、アクセスキーは動作しません。

Alt + Esc、 Ctrl + Esc、 Alt + Tab、および Ctrl + Alt + Del などの標準システムアクセラレータをアクセスキーとして定義できますが、これらはダイアログエレメントのクリックイベントまたはダブルクリックイベントを起動するのではなく、関連付けられたシステム機能を起動します。 MDI アプリケーションで使用される標準 MDI アクセラレータ（Ctrl + F4 や Ctrl + F6 など）、およびアクティブなサーバー（例：現在フォーカスがある ActiveX コントロール）に属するアクセラレータについても同様です。

ユーザー定義アクセスキーは、デスクトップ項目に関連付けられた同じユーザー定義ショートカットキーを書き換えることに注意してください。

同一のアクセスキーを複数のダイアログエレメントに関連付けると、クリックイベントまたはダブルクリックイベントが起動されるダイアログエレメントは定義されません。

SAME-AS 属性によって別のダイアログエレメントを参照するダイアログエレメントは、参照されるオブジェクトのアクセラレータを継承します。 例えば、メニュー項目が 1 つのシグナルを参照し、そのシグナルのアクセラレータが Ctrl + Alt + X である場合、メニュー項目の ACCELERATOR 属性をクエリした場合も Ctrl + Alt + X が返されます。 ただし、アクセラレータを押すと、参照されるダイアログエレメント（この例では、シグナル）のクリックイベントだけが起動されます。

Alt + X 形式（"X" は 1 つのアルファベット文字）のアクセラレータは、キーボードニーモニックとして予約されているため、使用しないでください。

アクセスキーのメニューへの表示

メニュー項目のアクセラレータを表示するためには、メニューテキストにまずタブ（h'09'）文字を追加して、次にアクセラレータ用のテキストを追加する必要があります。 ダイアログエディタのメニューエディタには文字列定義にタブ文字を入力する方法がないため、このエディタではスタティックに定義できません。 ただし、ダイアログのすべてのメニュー項目で繰り返し実行される共通コードを使用して、アクセラレータをダイナミックに追加できます。 これを説明する、ダイアログの AFTER-OPEN イベントから簡単に呼び出すことのできる外部サブルーチンを以下に示します。

```
DEFINE DATA
PARAMETER
  1 #DLG$WINDOW  HANDLE OF WINDOW
LOCAL
  1 #CONTROL     HANDLE OF GUI
  1 #COMMAND     HANDLE OF GUI
LOCAL USING NGULKEY1
END-DEFINE
*
DEFINE SUBROUTINE APPEND-ACCELERATORS
#CONTROL := #DLG$WINDOW.FIRST-CHILD
REPEAT UNTIL #CONTROL = NULL-HANDLE
  IF #CONTROL.TYPE = SUBMENU OR #CONTROL.TYPE = CONTEXTMENU
    #COMMAND := #CONTROL.FIRST-CHILD
    REPEAT UNTIL #COMMAND = NULL-HANDLE
      IF #COMMAND.ACCELERATOR <> ' '
       COMPRESS #COMMAND.STRING H'09' #COMMAND.ACCELERATOR INTO
         #COMMAND.STRING LEAVING NO SPACE
      END-IF
      #COMMAND := #COMMAND.SUCCESSOR
    END-REPEAT
  END-IF
 #CONTROL := #CONTROL.SUCCESSOR
END-REPEAT
END-SUBROUTINE
END
```

このダイナミック手法は、結果的にアクセラレータを2回（つまり、メニュー項目の ACCELERATOR 属性と STRING 属性に対して）定義しなくてもよいという利点があります。

使用言語が英語でない場合は、メニュー項目文字列に追加する前に ACCELERATOR 属性値の翻訳が必要となる場合があります。

113 ダイナミックデータ交換 - DDE

このchapterでは、次のトピックについて説明します。

概念

DDE は、異なるアプリケーション間でデータを交換できるようにするために Microsoft によって定義されているプロトコルです。 つまり、例えば Natural で作成されたアプリケーションとスプレッドシートの両方が DDE プロトコルを処理できるため、両者間でデータを交換できます。 DDE プロトコルを処理するアプリケーションは、標準化されたメッセージを使用して別の DDE アプリケーションと通信します。 一方のアプリケーションはクライアントとして定義され、他方はサーバーとして定義されます。 クライアントおよびサーバーは DDE 会話を保ちます。

Note: DDE の概念および用語に関する説明については、Microsoft Windows のドキュメントを参照してください。

DDE 会話のデータは、以下の 3 レベルの階層によって識別されます。

- サービス
- トピック
- 項目

DDE 会話は、クライアントが DDE サーバーに*サービス*を要求したときに確立されます。 1 つの DDE サーバーはすべてのアクティブなアプリケーションに 1 つまたは複数の*サービス*を提供します。

DDE サーバーは、各サービスに対して*トピック*をいくつでも提供できます。 DDE クライアントは、*サービス*の*トピック*に対して会話を要求します。

*サービス*の*トピック*上の会話で、DDE クライアントおよび DDE サーバーは交換されるデータを*項目*名によって一意に識別します。

DDE サーバーは、多くのトピックを含むことのできる多くのサービスをサポートします。さらに、トピックも多くの項目を含むことができます。

Natural を使用すると、DDE クライアントアプリケーションと DDE サーバーアプリケーションの両方を開発できます。 例えば、DDE サーバーとして機能するスプレッドシートにデータを要求する Natural DDE クライアントアプリケーションを作成することも、ワードプロセッサ（DDE クライアント）にデータを提供する Natural DDE サーバーアプリケーションを作成することもできます。

DDE クライアントアプリケーションおよび DDE サーバーアプリケーションを開発するために、以下の機能が提供されています。

- ライブラリ SYSTEM 内の接頭語 NGU で始まる多くのサブプログラム。これらのサブプログラムは、パラメータデータエリア NGULDDE1 で定義されたメッセージおよびデータを送信します。
- サブプログラムによって DDE 会話（DDE-VIEW）で使用されるパラメータを記述したパラメータデータエリア（NGULDDE1）。
- DDE メッセージを操作する DDE-CLIENT イベントおよび DDE-SERVER イベント。

DDE サーバーアプリケーションを開発するには、DDE-SERVER イベントに応答し、SYSTEM ライブラリ内の接頭語NGU-SERVERで始まるサブプログラムを使用してサービスおよびトピックの登録と DDE クライアントアプリケーションへのメッセージとデータの送信を行います。

DDE クライアントアプリケーションを開発するには、DDE-CLIENT イベントに応答し、SYSTEM ライブラリ内の接頭語 NGU-CLIENT で始まるサブプログラムを使用して会話の初期化と DDE サーバーアプリケーションへの要求およびその他の DDE コマンドの送信を行います。

常に、クライアントダイアログまたはサーバーダイアログにパラメータデータエリアNGULDDE1とローカルデータエリア NGULFCT1 を含める必要があります （SYSTEM ライブラリ内の接頭語 NGU- で始まるサブプログラムを使用するために NGULFCT1 が必要です）。

DDE サーバーアプリケーションの開発

以下では次のトピックについて説明します。

サービスおよびトピックの登録／登録解除

DDE サーバーアプリケーションを DDE クライアントアプリケーションから指定できるようにするには、DDE サーバーアプリケーションのサービス名とサービスでサポートされるすべてのトピックを登録する必要があります。 サブプログラム NGU-SERVER-REGISTER を使用して、DDE サーバーがサポートするサービス／トピックごとに登録を行います。 登録は、通常、基本ダイアログの 「AFTER OPEN」 イベントで処理されます。

サービス／トピックを初めて登録するときは、サーバーとして機能してクライアントからすべてのDDEメッセージを受け取るダイアログのダイアログIDを、Naturalに提供します。 これは、`DDE-VIEW.CONV-ID` にダイアログ ID を設定し、`DDE-VIEW.MESSAGE` にも文字列 "DLGID" を設定することによって行います。

後でサービスにトピックを追加したり、完全に新しいサービスを追加したりすることもできます。サブプログラムNGU-SERVER-UNREGISTERを使用してトピックを登録解除することもできます。

クライアントからのデータの取得

サービスおよびトピックを正常に登録できたら、DDE サーバーアプリケーションは、サービスの登録されたトピック上で会話を確立している DDE クライアントアプリケーションから DDE メッセージを受け取ることができます。

この DDE サーバーへのメッセージは、ダイアログの DDE-SERVER イベントで受信されます。イベントハンドラセクションの初めに、サブプログラム NGU-SERVER-GET-DATA を使用して DDE-VIEW にクライアントのメッセージデータを格納する必要があります。データを読み終えたら、受け取ったクライアントメッセージに基づいて行動する必要があります。有効なメッセージとその意味が、サブプログラム NGU-SERVER-GET-DATA の記述で説明されています。

クライアントへのデータの送信

クライアントメッセージは最終的にはサーバーがクライアントにデータを送ることを要求します。クライアントへのデータ送信は、サブプログラム NGU-SERVER-DATA を使用して行われます。

DDE サーバー操作の終了

DDE サーバー操作を終了する場合は、常にサブプログラム NGU-SERVER-STOP を使用します。これは、すべてのサービスを登録解除して、すべての確立された会話を終了します。そして、`CLOSE DIALOG` ステートメントでサーバーアプリケーションを終了します。

DDE クライアントアプリケーションの開発

以下では次のトピックについて説明します。

- DDE サーバーアプリケーションとの接続
- DDE サーバーアプリケーションのサービスの使用
- DDE サーバーアプリケーションからのデータの受信
- DDE サーバーアプリケーションからの切断

DDE サーバーアプリケーションとの接続

DDE クライアントアプリケーションは、DDE サーバーアプリケーションとの会話を確立するために、接続するサーバーのサービス名およびトピック名を指定してサブプログラム NGU-CLIENT-CONNECT を呼び出す必要があります。 サーバーから適切な DDE イベントを受け取るために、`DDE-VIEW.CONV-ID` にクライアントのダイアログ ID を設定し、`DDE-VIEW.MESSAGE` にも文字列 "DLGID" を設定する必要があります。 サブルーチンは一意な会話 ID を `DDE-VIEW.CONV-ID` に返します。 サーバーとの今後のすべての通信にこの値を使用する必要があります。

DDE サーバーアプリケーションのサービスの使用

会話が確立すると、サーバーのサービスを使用するためのいくつかのオプションがクライアントに提供されます。 以下の処理を実行できます。

- 特定の項目でデータを要求する（NGU-CLIENT-REQUEST を使用）。
- サーバーにデータを送信する（NGU-CLIENT-POKE を使用）。
- サーバーにコマンドの実行を依頼する（NGU-CLIENT-EXECUTE を使用）。
- サーバーとのウォームリンクまたはホットリンクを確立する（NGU-CLIENT-ADVISE-HOT、NGU-CLIENT-ADVISE-WARM、および NGU-CLIENT-ADVISE-TERM を使用）。

DDE サーバーアプリケーションからのデータの受信

DDE クライアントは、DDE サーバーからのデータまたはその他のメッセージをクライアントダイアログの DDE-CLIENT イベントを通して受け取ります。 サーバーがメッセージを送信すると、常にこのイベントが発生します。 まず NGU-CLIENT-GET-DATA を使用してこのメッセージの内容を取り出す必要があります。 こうすることによって、DDE-VIEW 構造に適切に格納されます。 クライアントは、どのメッセージ（DDE-VIEW.MESSAGE）が届いたかを識別して、適切に応答する必要があります。 有効なメッセージが、サブプログラム NGU-CLIENT-GET-DATA の記述にリストされています。

DDE サーバーアプリケーションからの切断

クライアントがこれ以上は会話が必要ないと判断したら、NGU-CLIENT-DISCONNECT を呼び出してサーバーに会話を終了するように通知する必要があります。

DDE クライアント操作の終了

クライアントアプリケーションが終了するか、またはDDEの使用を停止するときは、常にNGU-CLIENT-STOP を呼び出す必要があります。 これにより、Natural にクライアントのすべてのアクティブな会話を終了してアプリケーションの DDE 操作を停止するように通知します。

リターンコード

このセクションでは、リターンコードについて説明します。

Note: エラーコードの説明が完全でないものがあります。 この場合はアスタリスク（*）を付記してあります。

コード	意味
-1	無効なコマンドまたはコマンドパラメータが指定されました。 DDE データエリアが正しいタイプで、コマンドが正しいことを確認してください。
0	関数が正しく処理されました。
1	アプリケーションが DDEML ライブラリを 2 回以上初期化しようとしたときに、この値が返されます。 プログラムのロジックを確認してください。 プログラムの最後の実行中に DDEML が正しく終了したことも確認してください。
2	前に DDEML を正しく終了しないでプログラムを実行した場合に、この値がサーバーの初期化関数から返されます。 要求されたサービスが異常終了した場合も、常にこの値がコールバック関数によって返されます。
	基本となるレイヤでエラーが発生しました。 *
3	参照された会話 ID を持つアクティブな会話がありません。 正しいサービス名を指定したかどうかを確認してください。
4	最大インスタンス数が接続されているため、アプリケーションが DDE ライブラリを初期化できませんでした。
5	DDEML 通信が初期化されていません。 DDE 処理を行うには、先に DDEML を初期化する必要があります。
6	メモリの割り当てに問題が発生しました。 会話のどちら側かのメッセージキューが長くなり過ぎた場合に、このエラーが発生します。 *
7	サービス、トピック、または項目の名前が 255 文字を超えました。 フィールドが DDE-VIEW を正しく指定しているかどうかを確認して、これらの変数のいずれも 255 文字を超える文字列を含まないことを確実にしてください。
8	DDE ライブラリでエラーが発生しました。 Software AG サポートに連絡してください。 *
9	これは関数呼び出しによって返されます。 関数に渡されたパラメータが無効です。 パラメータを確認してください。
10	「Server Type Link」 がサポートされますが、UNLINK に対するコールバック関数は関数 `PIDsRegisterTopic` に渡されません。 *

コード	意味
11	少なくとも 1 つの会話がアクティブであるトピックを削除しようとしました。 これは、会話がまだ存続しているトピックを登録解除しようとした場合も含みます。
12	参照されているサービス／トピックが関数 `PIDsRegisterTopic` に登録されていません。
13	NGU-SERVER-DATA サブプログラムが使用されたときに DDE-VIEW.SERVICE へのリンクがアクティブではありませんでした。 サービス名を調べ、SDK ツールキット内の DDE-SPY を使用して使用可能なサービスを確認してください。
14	要求されたリンクタイプが無効です。
15	トランザクション ID が不正です。 DDE ビューでトランザクション ID 値を確認してください。
16	クライアントアプリケーションが会話を要求しましたが、その前に、リンクに対してデータを送信するための関数が指定されませんでした。
17	非同期トランザクションが要求されましたが、クライアントアプリケーションは完了したトランザクションの詳細を送信するための関数を指定しませんでした。 会話の初期化時にこの関数を指定する必要があります。
18	同期トランザクションでタイムアウトが発生しました。 トランザクションが完了するのにかかる時間が、DDE-VIEW 構造内の TIMEOUT 値よりも長くなりました。 TIMEOUT 値を増やすか、無限に待つことを表す "-1" を設定してください。
19 - 24	内部使用専用。

114 オブジェクトのリンクおよび埋め込み - OLE

このchapterでは、次のトピックについて説明します。

Natural における OLE とは

Natural は、以下の OLE テクノロジをサポートします。

- OLE ドキュメント
- OLE ビジュアル編集（インプレースアクティベーション）
- ActiveX コントロール

OLE に詳しくない場合は、Microsoft Win32 ソフトウェア開発キットのドキュメントを始めとする各種資料の 1 つを参照して、まず基本的な知識を得ることを強くお勧めします。

OLE ドキュメントサポート

OLE ドキュメントとは、エンドユーザーが異なるアプリケーションの操作に気をとられずにデータに集中できるように、異なる Windows アプリケーションをシームレスに統合するテクノロジです。 OLE を使用すると、例えば Natural のダイアログに Word for Windows のドキュメントを埋め込むことができます。 エンドユーザーがドキュメントを編集するためにテキストコンテナに入ると、Word 機能全体が使用できます。 したがって、エンドユーザーが Word を起動する必要はありません。

OLE ドキュメントサポートは Natural のダイアログエレメント OLE コンテナコントロールによって提供されます。

OLE ドキュメントテクノロジは、コンテナアプリケーションおよびサーバーアプリケーションを定義します。 コンテナアプリケーションとは、サーバーアプリケーションによって作成されたオブジェクトを使用できるアプリケーションです。 これらのオブジェクトはオブジェクトのリンクまたは埋め込みによって使用されます。 ダイアログエディタは OLE コンテナコントロールを提供するので、この意味で Natural はコンテナアプリケーションです。 典型的なサーバーアプリケーションは Microsoft Word で、Word ドキュメントは Natural によって使用されるオブジェクトになります。

埋め込みとリンク

- リンクとは、外部ファイルへのリンクを介してドキュメントの内容にアクセスすることを意味します。 このファイルは、サーバーの形式で保存されます（例えば、*.rtf* 形式のファイルは Natural 外のファイルシステムに保存されます。外部ファイルシステムに存在するサーバーとしては Microsoft Word などがあります。）
- 埋め込みとは、ドキュメントの内容がコンテナアプリケーションで管理され、コンテナの内部形式で保存されることを意味します。 埋め込みドキュメントは、以下のいずれかの方法で作成されます。
  - ドキュメントを新規にコンテナアプリケーションに構築する。
  - 外部ドキュメントをロードする。

埋め込みオブジェクトはビジュアル編集（「インプレースアクティベーション」）で編集されます。これに対して、リンクオブジェクトを編集するには特別のサーバーウィンドウを開く必要があります。

Natural は、ドキュメントの埋め込みおよびリンクのためのダイアログエレメントとして OLE コンテナコントロールを提供します。 さらに、埋め込みドキュメントを内部 Natural 形式で保存およびロードするためのアクションも提供します。 デフォルトでは、内部形式で埋め込まれるオブジェクトは、%NATGUI_BMP% ディレクトリにデフォルト拡張子 *.neo*（Natural Embedded Object）で保存されます。

手順 114.1. ダイアログの開始時に埋め込みオブジェクトを OLE コンテナに表示するには

1 コンテナコントロールの属性ウィンドウを起動します。

2 ［タイプ］エントリを［既存 OLE オブジェクト］に設定します。

3 ［名前］フィールドでファイル指定を選択します。

手順 114.2. ランタイム時に埋め込みオブジェクトをダイナミックに表示するには

- `PROCESS GUI` ステートメントの `OLE-READ-FROM-FILE` アクションを使用します。

手順 114.3. ダイアログの開始時にリンクオブジェクトを OLE コンテナに表示するには

1 コンテナコントロールの属性ウィンドウを起動します。

2 ［タイプ］エントリを［OLE サーバー］に設定します。

3 表示された［**Select OLE Server or Document**］ダイアログで、［ファイルから作成］を選択してファイル指定を選択します。

手順 114.4. ランタイム時にリンクオブジェクトをダイナミックに表示するには

- SERVER-OBJECT 属性に外部ドキュメントのファイル指定を割り当てます。

ビジュアル編集 - インプレースアクティベーション

インプレースアクティベーションとは、エンドユーザーがコンテナアプリケーションのウィンドウでサーバーアプリケーションを起動できることを意味します。 このサーバーアプリケーションは、Natural ダイアログの OLE コンテナコントロールに埋め込まれるオブジェクトに関連付けられます。 OLE コンテナコントロールをダブルクリックすると、サーバーアプリケーションが起動されます。 Natural ダイアログのツールバーおよびメニューバーコントロールはサーバーアプリケーションのツールバーおよびメニューとマージされます。 その結果、ダイアログに、サーバーの機能を使用したオブジェクトの編集を可能にするツールバー項目およびメニュー項目が組み込まれます。

ActiveX コントロールサポート

ActiveX コントロールサポートによって、Natural プログラマは Natural ダイアログ内で多くのサードパーティ製 ActiveX コントロールを使用できます。 Natural では、ActiveX コントロールのプロパティおよびメソッドに直接アクセスすることも、ActiveX コントロールのイベントをプログラムすることもできます。

ActiveX コントロールサポートは Natural のダイアログエレメント「ActiveX コントロール」によって提供されます。詳細については、「*ActiveX コントロールの操作*」を参照してください。

OLE コンテナコントロール

以下では次のトピックについて説明します。

- OLE コンテナコントロールの作成
- ダイアログエディタで OLE コンテナコントロールを作成する
- ランタイム時に OLE コンテナをダイナミックに作成する
- ランタイム時に OLE コンテナを消去または削除する
- OLE コンテナコントロールとダイアログのメニューバー

- その他の OLE コンテナコントロールの機能

OLE コンテナコントロールの作成

OLE コンテナコントロールは、ダイアログエディタでスタティックに作成することも、ランタイム時にダイナミックに作成することもできます。

ダイアログエディタで OLE コンテナコントロールを作成する

OLE コンテナコントロールを使用することによって、サーバーアプリケーションを統合できます。 サーバーアプリケーションは、OLE コンテナコントロールの属性ウィンドウの［タイプ］エントリに指定する［オブジェクト情報］グループフレームによって、以下の3つの方法で統合できます。

- タイプ：［新規OLE オブジェクト］。 挿入可能なオブジェクトのためのプレースホルダの働きをする OLE コンテナコントロールを作成します。 ランタイム時にサーバーアプリケーションを開始することによって埋め込みオブジェクトを作成できます。 埋め込みオブジェクトを Natural の埋め込みオブジェクト（*.neo* ファイル）として保存できます。
- タイプ：［既存 OLE オブジェクト］。 OLE コンテナコントロール内の既存の埋め込みオブジェクトを変更します。埋め込みオブジェクトはNaturalの埋め込みオブジェクト（*.neo* ファイル）として保存されます。
- タイプ：［OLE サーバー］。アプリケーション内にネイティブなOLEオブジェクトを作成するか、外部オブジェクトへのリンクを作成します。

手順 114.5. ダイアログエディタで OLE コンテナコントロールを作成するには

1 ダイアログエディタのメインメニューを［挿入］、［**OLE** コンテナ］の順に選択します。

2 マウスの右ボタンを押したまま縦／横方向にドラッグすることによって矩形を描画し、適切なところでボタンから手を離します。

空の OLE コンテナが作成されます。

手順 114.6. ダイアログの開始時に OLE コンテナにドキュメントを表示するには

1 OLE コンテナコントロールをダブルクリックして属性ウィンドウを開きます。

2 ［タイプ］選択ボックスで、外部ドキュメントをリンクするために［**OLE** サーバー］を選択します。 または、埋め込みオブジェクトを読み取るために［既存 **OLE** オブジェクト］を選択します。

3 ［..］ボタンを押して、外部または埋め込みオブジェクトファイルを選択します。

ランタイム時に OLE コンテナをダイナミックに作成する

イベントハンドラセクションで例を入力する前に、ダイアログのローカルデータエリアで OLE コンテナコントロールのハンドル変数を宣言します。

```
01 #OCT-1 HANDLE OF OLECONTAINER
```

ランタイム時に OLE コンテナコントロールを作成して外部ドキュメントにリンクする例は、以下のとおりです。

```
PROCESS GUI ACTION ADD WITH
PARAMETERS
   HANDLE-VARIABLE = #OCT-1
   TYPE = OLECONTAINER
   SERVER-OBJECT = 'PICTURE.BMP'
   RECTANGLE-X = 56
   RECTANGLE-Y = 32
   RECTANGLE-W = 336
   RECTANGLE-H = 160
   PARENT = #DLG$WINDOW
   SUPPRESS-CLICK-EVENT = SUPPRESSED
   SUPPRESS-DBL-CLICK-EVENT = SUPPRESSED
   SUPPRESS-CLOSE-EVENT = SUPPRESSED
   SUPPRESS-ACTIVATE-EVENT = SUPPRESSED
   SUPPRESS-CHANGE-EVENT = SUPPRESSED
END-PARAMETERS GIVING *ERROR
```

ランタイム時に OLE コンテナコントロールを作成して Natural 埋め込みオブジェクトを埋め込む例は、以下のとおりです。

```
PROCESS GUI ACTION ADD WITH
PARAMETERS
   HANDLE-VARIABLE = #OCT-1
   TYPE = OLECONTAINER
   EMBEDDED-OBJECT = 'SLIDE.NEO'
   RECTANGLE-X = 56
   RECTANGLE-Y = 32
   RECTANGLE-W = 336
   RECTANGLE-H = 160
   PARENT = #DLG$WINDOW
   SUPPRESS-CLICK-EVENT = SUPPRESSED
   SUPPRESS-DBL-CLICK-EVENT = SUPPRESSED
   SUPPRESS-CLOSE-EVENT = SUPPRESSED
   SUPPRESS-ACTIVATE-EVENT = SUPPRESSED
```

```
  SUPPRESS-CHANGE-EVENT = SUPPRESSED
  END-PARAMETERS GIVING *ERROR
```

ランタイム時に OLE コンテナを消去または削除する

このセクションでは、ランタイム時に OLE コンテナを消去または削除する例について説明します。

イベントハンドラセクションで例を入力する前に、ダイアログのローカルデータエリアで OLE コンテナコントロールのハンドル変数を宣言します。

```
01 #OCT-1 HANDLE OF OLECONTAINER
```

OLE コンテナコントロールを消去（含まれるドキュメントを削除）する例は、以下のとおりです。

```
PROCESS GUI ACTION CLEAR WITH #OCT-1
```

OLE コンテナコントロールを削除する例は、以下のとおりです。

```
PROCESS GUI ACTION DELETE WITH #OCT-1
```

OLE コンテナコントロールとダイアログのメニューバー

メニュー項目の属性 MENU-ITEM-OLE は、サーバーのインプレースアクティベーション中に当該メニュー項目を表示するかどうかと表示場所を決める 4 つの異なる値を取ることができます。

また、メニュー項目の属性 MENU-ITEM-TYPE も値 MT-OBJECTVERBS を持ちます。 これにより、OLE コンテナコントロールが使用できるサーバーアクション（コマンド）をこのメニュー項目に表示するようにできます。

その他の OLE コンテナコントロールの機能

OLE コンテナコントロールにドキュメントを表示中に OLE コンテナコントロールの矩形内をダブルクリックすると、サーバーのデフォルトコマンドを起動できます。 これは、PROCESS GUI ステートメントの OLE-ACTIVATE アクションを実行することと同等です。 さらに、エンドユーザーは、ポップアップメニューを表示して、そこからサーバーコマンドを選択することもできます。 ポップアップメニューは、OLE コンテナ内でマウスの右ボタンをクリックすると表示されます。 そこで希望するコマンドを選択してからマウスの右ボタンを離します。

OLE コンテナコントロールの MODIFIABLE 属性に FALSE を設定すると、コンテナをダブルクリックしてもサーバーのデフォルトコマンドは起動されません。また、マウスの右ボタンをクリック

しても使用可能なサーバーコマンドを含んだポップアップメニューは表示されません（「*標準化されたプロシージャの実行*」も参照）。

ビジュアル編集（インプレースアクティベーション）中、サーバーはドキュメントの編集にNaturalダイアログを使用します。サーバーが自分のタスクとして実行するのではなく、Natural処理が続きます。 したがって、イベントコードを実行して、例えばタイマのイベントセクションで `PROCESS GUI ACTION OLE-DEACTIVATE, WITH #OCT-1` を指定することによってビジュアル編集を一定時間に制限できます（「*標準化されたプロシージャの実行*」も参照）。

属性、イベント、および PROCESS GUI ステートメントアクション

以下のセクションに、OLEコンテナコントロールに特別に適用されるすべての属性、イベント、および `PROCESS GUI` ステートメントアクションをリストします。

属性

OLE コンテナコントロールに提供されている OLE 固有の属性は、以下のとおりです。

- `EMBEDDED-OBJECT`
- `ICONIZED`
- `OBJECT-SIZE`
- `SERVER-OBJECT`
- `SERVER-PROGID`
- `SUPPRESS-ACTIVATE-EVENT`
- `SUPPRESS-CLOSE-EVENT`
- `ZOOM-FACTOR`

イベント

サーバーアプリケーションが起動されると、以下の OLE 固有のイベントが発生します。

- Activate イベント

PROCESS GUI ステートメントアクション

OLE コンテナコントロールに提供されている OLE 固有の PROCESS GUI ステートメントアクションは、以下のとおりです。

- OLE-ACTIVATE
- OLE-DEACTIVATE
- OLE-GET-DATA
- OLE-INSERT-OBJECT
- OLE-READ-FROM-FILE
- OLE-SAVE-TO-FILE
- OLE-SET-DATA

115 結果インターフェイス

このchapterでは、次のトピックについて説明します。

結果インターフェイスの目的

結果インターフェイスにより、プログラマはNaturalスタジオの結果ウィンドウ内にデータを表示できるようになります。『*Natural スタジオの使用*』ドキュメントの「*結果ウィンドウ*」も参照してください。

Note: メニューコマンド［オブジェクト検索］および［**Catall**］の結果は、結果インターフェイスの影響を受けません。

結果ウィンドウのタブの設計と使用法は、アプリケーションプログラミングインターフェイス（API）経由で決定できます。 一般的に、桁と行を備えた詳細なビューが使用されます。

コンテキストメニューをエントリごとに作成して、ユーザー定義タブが表示された後、以降の処理のために使用できるようにすることができます。

この処理は 2 つのプログラム内で定義する必要があります。

1. コンテキストメニューが表示される前に更新コマンドハンドラで定義します。
2. 項目が選択される場合にコマンドハンドラで定義します。

結果インターフェイス用のアプリケーションプログラミングインターフェイスは `USR5001N`～`USR5017N` で、ライブラリ `SYSEXT` にあります。

さまざまな機能の例は`USR5001P`にあり、更新コマンドハンドラは`USR5001A`、コマンドハンドラは `USR5001B` にあります。

Notes:

1. このインターフェイスでは、［オブジェクト検索］タブおよび［**Catall**］タブなどの事前定義されたタブの修正はできません。
2. 結果ウィンドウおよび結果インターフェイスは、Natural スタジオからのみアクセスできます。

結果ウィンドウコントロールバーのアクセス

以下のアプリケーションプログラミングインターフェイスを使用すると、結果ウィンドウコントロールバーにアクセスできます。

インターフェイス	機能
USR5001N	結果ウィンドウの ON と OFF を切り換えます。 結果ウィンドウの表示をチェックします。

タブ処理

以下に挙げるアプリケーションプログラミングインターフェイスを使用すると、一般的なタブのレイアウトを定義できます。

タブには、以下のすべてまたは 1 つを含ませることができます。

- チェックボックス
- 全行選択
- 単一行選択
- イメージ

タブは、以下の属性で定義できます。

- ビューのレイアウト（大小アイコン、リスト、または詳細ビュー）
- 複数の使用法（チェックボックス、イメージ、グリッド線、全行または単一行選択、ビュー変更）
- タブラベルのレイアウト（テキスト、ビットマップ、またはアイコン）

インターフェイス	機能
USR5004N	タブのレイアウトの追加、置換、削除、および管理。
USR5005N	アクティブタブの設定および取得。 タブをアクティブに設定し、このタブにフォーカスを設定します。

イメージ処理

以下のアプリケーションプログラミングインターフェイスを使用すると、ビットマップ（*.bmp*）およびアイコン（*.ico*）を事前定義したタブに指定できます。

インターフェイス	機能
USR5002N	指定したタブに対してビットマップおよびアイコンを追加および削除します。

コンテキストメニュー処理

以下のアプリケーションプログラミングインターフェイスを使用すると、ユーザー定義コンテキストメニューを指定できます。

インターフェイス	機能
USR5003N	タブのコンテキストメニューを追加および削除します。
USR5007N	コンテキストメニュー項目のチェック済みたは使用可能状態を設定および取得します。

コンテキストメニューの階層は手動で定義する必要があります。

以下の配列コンポーネントを定義できます。

配列コンポーネント	値	説明
タイプ	1～4	1 - コンテキストメニュー処理。 2 - セパレータ行。 3 - サブメニューの開始。 4 - サブメニューの終了。
コマンド ID	1～255	コンテキストメニューの特定の項目を識別する自由選択可能な番号。コマンドハンドラ内で使用されます。
ラベル	英数字テキスト	タイプ 1 および 3 のコンテキストメニュー項目のテキスト。 ステータスバーのテキストは、"H'0'A" で分割できます。
イメージ	イメージのハンドル	事前に定義したイメージのハンドル（ビットマップまたはアイコン）。イメージはコンテキストメニュー項目のテキストの前に配置されます。

コマンド処理

プログラムは、更新コマンドハンドラまたはコマンドハンドラとして割り当てることができます。

ユーザー定義データは、コマンドハンドラの内部ワークエリアに保存またはリストアすることができます。

例：タブのハンドル

以下のアプリケーションプログラミングインターフェイスを使用できます。

インターフェイス	機能
USR5006N	更新コマンドハンドラおよびコマンドハンドラを定義します。
USR5016N	コマンドハンドラワークエリアのデータを設定および取得します。

列処理

以下のアプリケーションプログラミングインターフェイスを使用すると、一般的な列のレイアウトを定義できます。

列には、以下のすべてまたは 1 つを含ませることができます。

- タイトル
- 幅
- データ位置
- 列ソート

さらに、列のデフォルト幅と指定幅を個々に設定できます。

インターフェイス	機能
USR5008N	タブの列を追加、挿入、および削除します。
USR5009N	列の数をカウントします。
USR5010N	デフォルトの列幅および指定した列の幅を設定および取得します。

行処理

以下のアプリケーションプログラミングインターフェイスを使用すると、イメージおよびコンテキストメニュー付きの行を定義できます。

インターフェイス	機能
USR5009N	行の数をカウントします。
USR5011N	タブの行を追加、挿入、および削除します。
USR5015N	行を結果ウィンドウの表示可能エリアにスクロールできます。

データ処理

以下のアプリケーションプログラミングインターフェイスを使用すると、ユーザー定義データを定義済みの列および行に書き込むことができます。

タブにチェックボックスを定義している場合、行ごとにチェックボックスを有効化または無効化できます。

インターフェイス	機能
USR5012N	タブにデータを設定および取得します。
USR5013N	行のチェック済み状態を設定および取得します。

選択処理

以下のアプリケーションプログラミングインターフェイスを使用すると、行を個別に選択できます。

インターフェイス	機能
USR5014N	■ 選択した行を設定、リセット、および取得します。 ■ 選択した行の合計をカウントします。 ■ 行選択を設定およびリセットします。
USR5015N	フォーカス行を設定および取得します。
USR5017N	選択した行をクリップボードにコピーします。

116 アプリケーションに対するキャラクタユーザーインターフェイスの設計

アプリケーションのユーザーインターフェイス、つまり、アプリケーションをどのようにユーザーに見せるかは、アプリケーションを作成する際の重要な検討項目です。

ここでは、Naturalが提供する、統一された外観を持ち、ユーザーへの指示やユーザーとの対話を行うための強力なメカニズムであるキャラクタユーザーインターフェイスを設計するさまざまな方法について説明します。

Note: グラフィカルユーザーインターフェイス（GUI）の詳細については、「*イベントドリブンプログラミングについて*」を参照してください。

ユーザーインターフェイスを設計するときは、標準および標準化が重要な要素となります。

Naturalを使用すると、さまざまなハードウェアおよびオペレーティングシステムで共通の機能をエンドユーザーに提供できます。

Natural には、全般的な画面レイアウト（情報、データ、メッセージの各エリア）、ファンクションキーの割り当て、およびウィンドウのレイアウトが含まれています。

このセクションでは、次のトピックについて説明します。

- 画面設計
- ダイアログ設計

117 画面設計

このchapterでは、全般的な画面レイアウトを定義するオプションについて説明します。

メッセージ行の制御 - 端末コマンド %M

端末コマンド `%M` には、Natural メッセージ行を表示する方法と位置を定義するためのさまざまなオプションを使用できます。

以下に参考情報を示します。

- メッセージ行の位置設定
- メッセージ行の色

メッセージ行の位置設定

%MB
画面の一番下にメッセージ行を表示します。

%MT
画面の一番上にメッセージ行を表示します。

メッセージ行の位置設定に関する他のオプションについては、『*端末コマンド*』ドキュメントの「*%M - メッセージ行の制御*」を参照してください。

メッセージ行の色

%M=*color-code*
指定した色でメッセージ行を表示します。カラーコードの詳細については、『*パラメータリファレンス*』ドキュメントに記載されている、セッションパラメータ `CD` の説明を参照してください。

フィールドへの色の割り当て - 端末コマンド %=

端末コマンド `%=` を使用すると、もともと色の設定をサポートしていないプログラムのフィールド属性に色を割り当てることができます。 コマンドにより、指定の属性で定義されたすべてのフィールド／テキストが、指定の色で表示されます。

定義済みの色割り当てが使用している端末タイプに合わない場合は、このコマンドを使用して元の割り当てを新しい割り当てで上書きできます。

%=コマンドは、Naturalエディタ内で、例えばマップの作成中に色割り当てをダイナミックに定義するために使用することもできます。

コード	説明
空白	色変換テーブルをクリアします。
F	新たに定義した色で、プログラムで割り当てた色を上書きします。
N	プログラムで割り当てられた色属性は変更されません。
O	出力フィールド
M	変更可能なフィールド（出力および入力）
T	テキスト定数
B	点滅
C	斜体
D	デフォルト値
I	高輝度
U	下線付き
V	反転
BG	背景
BL	青
GR	緑
NE	デフォルト色
PI	ピンク
RE	赤
TU	空色
YE	黄色

例：

```
%=TI=RE,OB=YE
```

この例では、すべての高輝度テキストフィールドに赤を割り当て、すべての点滅出力フィールドに黄色を割り当てています。

情報行 - 端末コマンド %X

端末コマンド %X は、Natural 情報行の表示を制御します。

詳細については、『*端末コマンド*』ドキュメントに記載されている、端末コマンド `%X` の説明を参照してください。

ウィンドウ

以下に参考情報を示します。

ウィンドウとは

*ウィンドウ*とは、端末画面上に表示される、プログラムによって構築された論理ページのセグメントのことです。

*論理*ページとは、Natural の出力エリアのことです。つまり、論理ページには、Natural プログラムによって表示用に生成された現在のレポート／マップが含まれています。 この論理ページは物理画面よりも大きくすることができます。

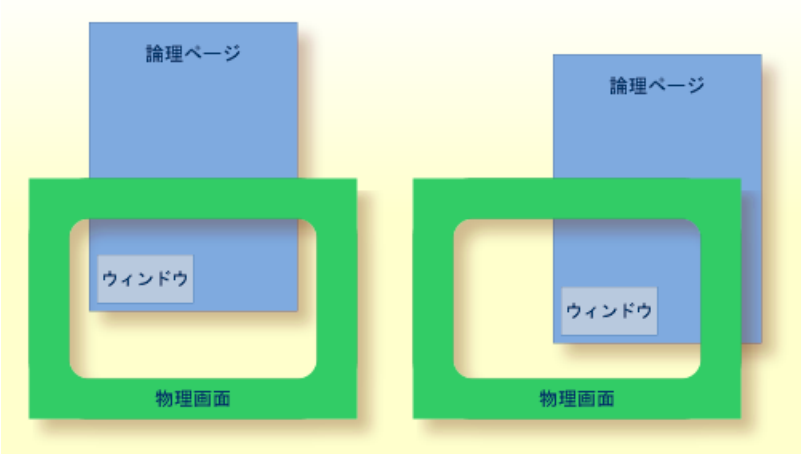
存在に気付かない場合もありますが、ウィンドウは常に存在しています。`DEFINE WINDOW` ステートメントで個別に指定されない限り、ウィンドウのサイズは端末画面の物理サイズと同一です。

ウィンドウは、以下の 2 つの方法で操作できます。

- *物理画面*上では、ウィンドウのサイズと位置を制御できます。
- *論理*ページ上では、ウィンドウの位置を制御できます。

物理画面上の位置

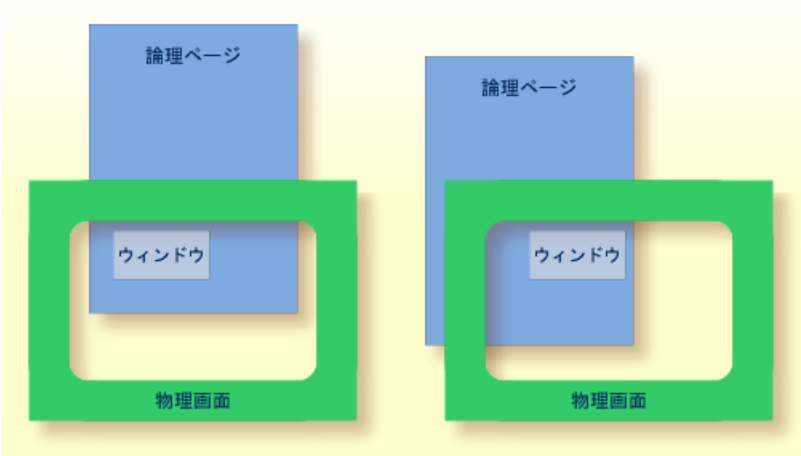
以下の図は、物理画面上のウィンドウの位置を示しています。 どちらの例にも、画面上のウィンドウの位置が違うだけで、論理ページの同じセクションが表示されていることに注意してください。



論理ページ上の位置

以下の図は、論理ページ上のウィンドウの位置を示しています。

*論理*ページでウィンドウの位置を変更しても、*物理画面*上のウィンドウのサイズおよび位置は変更されません。 つまり、ウィンドウがページ上で移動するのではなく、ページがウィンドウの「下」 で移動するということです。



DEFINE WINDOW ステートメント

DEFINE WINDOW ステートメントを使用して、*物理画面*上のウィンドウのサイズ、位置、属性を指定します。

DEFINE WINDOW ステートメントはウィンドウをアクティブ化しません。ウィンドウをアクティブ化するには、SET WINDOW ステートメントを使用するか、INPUT ステートメントの WINDOW 節を使用します。

DEFINE WINDOW ステートメントには、さまざまなオプションを使用できます。これらのオプションについて、以下の例で説明します。

以下のプログラムは、物理画面上のウィンドウの位置を定義しています。

```
** Example 'WINDX01': DEFINE WINDOW
************************************************************************
DEFINE DATA LOCAL
1 COMMAND (A10)
END-DEFINE
*
DEFINE WINDOW TEST
       SIZE 5*25
       BASE 5/40
       TITLE 'Sample Window'
       CONTROL WINDOW
       FRAMED POSITION SYMBOL BOTTOM LEFT
*
INPUT WINDOW='TEST' WITH TEXT 'message line'
      COMMAND (AD=I'_') /
      'dataline 1' /
      'dataline 2' /
      'dataline 3' 'long data line'
*
IF COMMAND = 'TEST2'
  FETCH 'WINDX02'
ELSE
  IF COMMAND = '.'
    STOP
  ELSE
    REINPUT 'invalid command'
  END-IF
END-IF
END
```

ウィンドウは、ウィンドウ名によって識別されます。名前の最大長は32文字です。ウィンドウ名には、ユーザー定義変数と同じ命名規則が適用されます。この例では、ウィンドウ名は"TEST"です。

ウィンドウのサイズは、SIZE オプションで設定します。 この例では、ウィンドウの高さは 5 行、幅は 25 列（ポジション）です。

ウィンドウの位置は、BASE オプションで設定します。 この例では、ウィンドウの左上隅は行 5、列 40 に位置します。

TITLE オプションを使用すると、ウィンドウフレームに表示するタイトルを定義できますが、これはウィンドウのフレームを定義した場合にのみ有効です。

CONTROL 節を使用して、PF キー行、メッセージ行、および統計行をウィンドウに表示するかフル物理画面に表示するかを指定します。 この例では、CONTROL WINDOW によって、ウィンドウ内にメッセージ行が表示されます。 CONTROL SCREEN を使用すると、ウィンドウの外側の物理画面全体に行が表示されます。 CONTROL 節を省略すると、デフォルトで CONTROL WINDOW が適用されます。

FRAMED オプションを使用すると、フレーム付きのウィンドウを定義できます。 このフレームは、カーソル依存です。 カーソル依存が適用可能な場所では、適切な記号（<、-、+、>。POSITION 節の説明を参照）の上にカーソルを置いて Enter キーを押すだけで、ウィンドウ内のページを上下左右に移動できます。 つまり、物理画面上のウィンドウの下にある*論理ページ*を移動できます。 記号が表示されない場合は、カーソルを枠線の最上部（前のページに戻る場合）または最下部（次のページに進む場合）に置いて Enter キーを押すことにより、ウィンドウ内でページを前後に移動することができます。

FRAMED オプションの POSITION 節を使用して、ウィンドウのフレームに表示する、論理ページ上のウィンドウの位置情報を定義します。 これは、論理ページがウィンドウよりも大きい場合にのみ適用されます。そうでない場合は、POSITION 節は無視されます。 位置情報は、論理ページが拡張する方向（現在のウィンドウの上、下、左、右）を示します。

POSITION 節を省略すると、デフォルトで POSITION SYMBOL TOP RIGHT が適用されます。

POSITION SYMBOL を使用すると、位置情報が記号形式（「More: <-+>」）で表示されます。 情報は、上部か下部のいずれか、または両方のフレームラインに表示されます。

TOP／BOTTOM で、位置情報を上のフレームラインまたは下のフレームラインのどちらに表示するかを指定します。

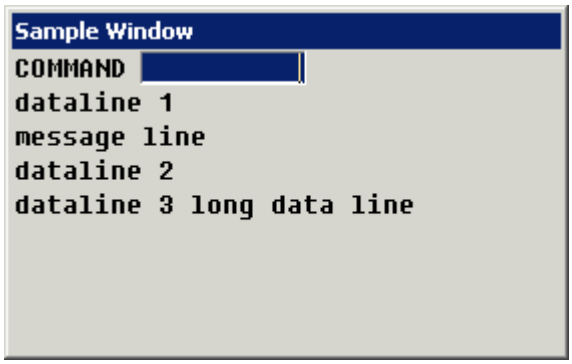
LEFT／RIGHT で、位置情報をフレームラインの左または右のどちらに表示するかを指定します。

INPUT WINDOW ステートメント

INPUT WINDOW ステートメントは、DEFINE WINDOW ステートメントで定義したウィンドウを有効にします。 以下の例では、ウィンドウ TEST が有効化されています。 ウィンドウにデータを出力する場合（WRITE ステートメントを使用する場合など）は、SET WINDOW ステートメントを使用します。

前述のプログラムを実行すると、入力フィールド COMMAND がウィンドウに表示されます。 セッションパラメータ AD を使用して、フィールドの値を高輝度で表示し、下線を充填文字として使用するよう定義されています。

プログラム WINDX01 の出力：



複数のウィンドウ

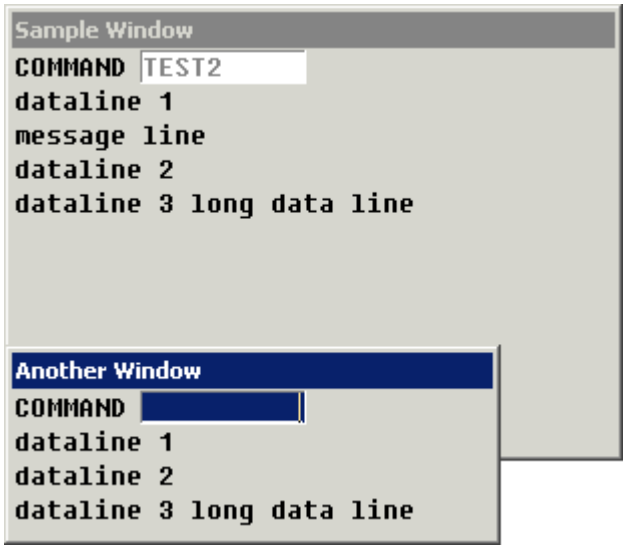
複数のウィンドウを開くことはもちろん可能です。ただし、Naturalウィンドウでは、アクティブにできるのは常に1つのウィンドウのみ、つまり最新のウィンドウのみです。画面に前のウィンドウが表示されていても、アクティブではないので、Naturalに無視されます。現在のウィンドウにのみ入力できます。 入力する十分なスペースがない場合は、まずウィンドウサイズを調整する必要があります。

COMMAND フィールドに「TEST2」と入力すると、プログラム WINDX02 が実行されます。

```
** Example 'WINDX02': DEFINE WINDOW
************************************************************************
DEFINE DATA LOCAL
1 COMMAND (A10)
END-DEFINE
*
DEFINE WINDOW TEST2
       SIZE 5*30
       BASE 15/40
       TITLE 'Another Window'
       CONTROL SCREEN
       FRAMED POSITION SYMBOL BOTTOM LEFT
```

```
*
INPUT WINDOW='TEST2' WITH TEXT 'message line'
      COMMAND (AD=I'_') /
      'dataline 1' /
      'dataline 2' /
      'dataline 3' 'long data line'
*
IF COMMAND = 'TEST'
  FETCH 'WINDX01'
ELSE
  IF COMMAND = '.'
    STOP
  ELSE
    REINPUT 'invalid command'
  END-IF
END-IF
END
```

2つ目のウィンドウが開きます。もう1つのウィンドウは表示されたままですが、非アクティブです。



新しいウィンドウのメッセージ行は、ウィンドウ上ではなく、出力ウィンドウの下部に表示されることに注意してください。これは、WINDX02プログラムの`CONTROL SCREEN`節によって定義されています。

`DEFINE WINDOW`、`INPUT WINDOW`、`SET WINDOW`の各ステートメントの詳細については、『ステートメント』ドキュメントの対応する説明を参照してください。

標準／ダイナミックレイアウトマップ

標準レイアウトマップ

*標準レイアウト*は、マップエディタで定義できます。このレイアウトによって、アプリケーション全体のすべてのマップの外観を統一できます。

標準レイアウトを参照するマップを初期化すると、その標準レイアウトはマップに固定的に組み込まれます。 つまり、標準レイアウトを変更する場合、変更を有効にするために影響を受けるマップをすべて再カタログ化する必要があります。

ダイナミックレイアウトマップ

標準レイアウトとは対照的に、*ダイナミックレイアウト*は、そのレイアウトを参照するマップに固定的に組み込まれるのではなく、ランタイム時に組み込まれます。

したがって、レイアウトマップをマップエディタの［マップエディタプロファイル（Define Map Settings For MAP）］画面で「ダイナミック」として定義すると、レイアウトに対する変更はすべて、そのレイアウトを参照するすべてのマップに引き継がれます。 マップを再カタログ化する必要はありません。

レイアウトマップの詳細については、『エディタ』ドキュメントの「マップエディタ」を参照してください。

多言語ユーザーインターフェイス

Natural を使用すると、各国で使用するための多言語アプリケーションを作成できます。

マップ、ヘルプルーチン、エラーメッセージ、プログラム、関数、サブプログラム、コピーコードは、ダブルバイト文字セットを使用する言語を含め、最大 60 の異なる言語で定義できます。

以下に参考情報を示します。

- 言語コード
- Natural オブジェクトの言語の定義
- ユーザー言語の定義
- 多言語オブジェクトの参照
- プログラム
- エラーメッセージ

- 日付／時刻フィールドの編集マスク

言語コード

Naturalでは、言語ごとに*言語コード*（1～60）が割り当てられています。以下の表は、言語コード表を一部抜粋したものです。 言語コードの完全な表については、『システム変数』ドキュメントに記載されている、システム変数 *LANGUAGE の説明を参照してください。

言語コード	言語	オブジェクト名のマップコード
1	英語	1
2	ドイツ語	2
3	フランス語	3
4	スペイン語	4
5	イタリア語	5
6	オランダ語	6
7	トルコ語	7
8	デンマーク語	8
9	ノルウェー語	9
10	アルバニア語	A
11	ポルトガル語	B

言語コード（左の列）が、システム変数 *LANGUAGE に含まれているコードです。 このコードはNaturalによって内部的に使用されます。このコードを使用してユーザー言語を定義します（以下の「*ユーザー言語の定義*」を参照）。 表の右の列のマップコードを使用して、Natural オブジェクトの言語を識別します。

例：

ポルトガルの言語コードは 「11」 です。 ポルトガル語版の Natural オブジェクトをカタログ化するときに使用するコードは 「B」 です。

言語コードの完全な表については、『システム変数』ドキュメントに記載されている、システム変数 *LANGUAGE の説明を参照してください。

Natural オブジェクトの言語の定義

Naturalオブジェクト（マップ、ヘルプルーチン、プログラム、関数、サブプログラム、コピーコード）の言語を定義するには、対応するマップコードをオブジェクト名に追加します。 マップコードを除くオブジェクト名は、すべての言語で同一である必要があります。

以下の例では、マップが英語とドイツ語で作成されています。マップの言語を識別するために、各言語に対応するマップコードがマップ名に組み込まれています。

多言語アプリケーションのマップ名の例

DEMO1 = 英語のマップ（マップコード 1）

DEMO2 = ドイツ語のマップ（マップコード 2）

英字のマップコードを持つ言語の定義

マップコードは、1〜9、A〜Z または a〜y の範囲内の値です。 英字のマップコードには、特別な操作が必要です。

小文字は大文字に自動的に変換されるため、通常は、名前に小文字があるオブジェクトをカタログ化することはできません。

しかし、例えば、言語コードが 59 でマップコードが "x" の漢字（日本語）をオブジェクト名に定義する場合、小文字を使用する必要があります。

このようなオブジェクトをカタログ化するには、まず正しい言語コード（ここでは 59）を端末コマンド `%L=nn` を使用して設定します。*nn* は言語コードです。

その後、実際のマップコードの代わりにアンパーサンド（&）文字をオブジェクト名に使用して、オブジェクトをカタログ化します。 したがって、マップ DEMO の日本語版を作成するには、DEMO& という名前でマップを STOW します。

ここで Natural オブジェクトのリストを確認すると、マップは DEMOx として正しくリストされます。

言語コード 1〜9 および大文字の A〜Z のオブジェクトは、アンパーサンド（&）表記を使用せずに直接カタログ化できます。

以下の例では、3 つのマップ DEMO1、DEMO2、および DEMOx. を確認できます。 マップ DEMOxを削除するには、作成時と同じ方法で、つまり、端末コマンド `%L=59` で正しい言語を設定し、アンパーサンド（&）表記（DEMO&）を使用して削除します。

ユーザー言語の定義

システム変数*LANGUAGEの定義として、プロファイルパラメータULANG（『パラメータリファレンス』を参照）または端末コマンド%L=*nn*（*nn*は言語コード）を使用して、ユーザーごとに使用する言語を定義します。

多言語オブジェクトの参照

多言語オブジェクトをプログラムで参照するには、オブジェクト名にアンパーサンド（&）文字を使用します。

以下のプログラムでは、マップDEMO1およびDEMO2が使用されています。 マップ名の最後のアンパーサンド（&）文字はマップコードを表し、*LANGUAGEシステム変数で定義されている現在の言語のマップを使用することを意味します。

```
DEFINE DATA LOCAL
1 PERSONNEL VIEW OF EMPLOYEES
  2 NAME (A20)
  2 PERSONNEL-ID (A8)
1 CAR VIEW OF VEHICLES
   2 REG-NUM (A15)
1 #CODE (N1)
END-DEFINE
*
INPUT USING MAP 'DEMO&' /* <--- INVOKE MAP WITH CURRENT LANGUAGE CODE
...
```

このプログラムを実行すると、英語のマップ（DEMO1）が表示されます。 これは、*LANGUAGEの現在の値が "1" = 英語であるためです。

```
                                                    MAP DEMO1


                              SAMPLE MAP



                    Please select a function!



                    1.) Employee information

                    2.) Vehicle information
```

```
                    Enter code here: _
```

以下の例では、端末コマンド %L=2 を使用して、言語コードが "2" = ドイツ語に変更されています。

プログラムを再度実行すると、ドイツ語のマップ（DEMO2）が表示されます。

```
                              BEISPIEL-MAP



                    Bitte wählen Sie eine Funktion!


                    1.) Mitarbeiterdaten

                    2.) Fahrzeugdaten


                 Code hier eingeben: _
```

プログラム

アプリケーションの中には、多言語プログラムを定義すると便利なものがあります。 例えば、一般的な請求書発行プログラムでは、請求書を発行する国ごとのさまざまな税制度に対応する、複数のサブプログラムを使用します。

多言語プログラムは、前述のマップの場合と同様の方法で定義します。

エラーメッセージ

Natural ユーティリティ SYSERR を使用すると、Natural エラーメッセージを最大 60 の言語に変換できます。また、独自のエラーメッセージを定義することもできます。

ユーザーに表示されるメッセージの言語は、*LANGUAGE システム変数によって決まります。

エラーメッセージの詳細については、『*ユーティリティ*』ドキュメントの「*SYSERR ユーティリティ*」を参照してください。

日付／時刻フィールドの編集マスク

編集マスクを使用して定義する日付フィールドおよび時刻フィールドに使用される言語もまた、システム変数 *LANGUAGE によって決まります。

編集マスクの詳細については、『*パラメータリファレンス*』に記載されている、セッションパラメータ EM の説明を参照してください。

スキル別ユーザーインターフェイス

同じアプリケーションを使用しつつ、ユーザーのスキルレベルに応じて（さまざまな詳細度の）異なるマップを使用したい場合があります。

アプリケーションを多言語で国際的に使用する必要がない場合、多言語のマップに対する手法を使用して、詳細度の異なるマップを定義できます。

例えば、初心者用に言語コード 1 を定義し、上級者用に言語コード 2 を定義できます。 この単純かつ有効な手法を以下に示します。

以下のマップ（PERS1）には、エンドユーザー向けに、メニュー機能を選択する方法の説明が含まれています。 非常に詳細な情報が記載されています。 マップ名には、マップコード 1 が含まれています。

```
                                                    MAP PERS1

                              SAMPLE MAP

                    Please select a function


                    1.) Employee information  _

                    2.) Vehicle information   _

                        Enter code:  _

 To select a function, do one of the following:

 - place the cursor on the input field next to desired function and press ENTER
 - mark the input field next to desired function with an X and press ENTER
```

```
- enter the desired function code (1 or 2) in the 'Enter code' field and press
  ENTER
```

詳細な説明が含まれていない同じマップが、同じ名前で保存されています。ただし、マップコードは 2 です。

```
                                                MAP PERS2


                          SAMPLE MAP

                  Please select a function


                  1.) Employee information  _

                  2.) Vehicle information   _

                      Enter code:  _
```

上記の例では、ULANG プロファイルパラメータの値が 1 の場合は詳細な説明付きのマップが出力され、値が 2 の場合は説明なしのマップが出力されます。『*パラメータリファレンス*』に記載されている、プロファイルパラメータ ULANG の説明も参照してください。

118 ダイアログ設計

このchapterでは、ユーザーとアプリケーションとの対話を簡単かつ柔軟に行うキャラクタユーザーインターフェイスの設計方法について説明します。

Note: グラフィカルユーザーインターフェイス（GUI）の詳細については、「*イベントドリブンプログラミングについて*」を参照してください。

以下のトピックについて説明します。

フィールドに基づいた処理

*CURS-FIELD および POS(*field-name*)

システム変数 *CURS-FIELD をシステム関数 POS(*field-name*) とともに使用すると、Enter キーが押された時点でカーソルが位置付けられているフィールドに基づいた処理を定義できます。

*CURS-FIELD には、カーソルが現在位置付けられている入力フィールドの内部 ID が格納されます。この ID は単体では使用できません。POS(*field-name*) と組み合わせてのみ使用できます。

*CURS-FIELD および POS(*field-name*) は、例えば、特定のフィールドにカーソルを配置して Enter キーを押すだけで機能を選択できるようにするために使用できます。

以下の例は、そのようなアプリケーションを示しています。

```
DEFINE DATA LOCAL
1 #EMP (A1)
1 #CAR (A1)
1 #CODE (N1)
END-DEFINE
*
INPUT USING MAP 'CURS'
*
DECIDE FOR FIRST CONDITION
  WHEN *CURS-FIELD = POS(#EMP) OR #EMP = 'X' OR #CODE = 1
    FETCH 'LISTEMP'
  WHEN *CURS-FIELD = POS(#CAR) OR #CAR = 'X' OR #CODE = 2
    FETCH 'LISTCAR'
   WHEN NONE
    REINPUT 'PLEASE MAKE A VALID SELECTION'
```

```
END-DECIDE
END
```

結果は以下のとおりです。

```
                                SAMPLE MAP

                  Please select a function


                  1.) Employee information  _
                  2.) Vehicle information   _  <== Cursor positioned
                                                    on field

                      Enter code:  _

To select a function, do one of the following:

- place the cursor on the input field next to desired function and press ENTER
- mark the input field next to desired function with an X and press ENTER
- enter the desired function code (1 or 2) in the 'Enter code' field and press
  ENTER
```

ユーザーがカーソルを［Employee information］の隣の入力フィールド（#EMP）に配置して Enter キーを押すと、プログラム LISTEMP によって従業員名のリストが表示されます。

```
Page      1                                              2001-01-22  09:39:32

        NAME
--------------------

ABELLAN
ACHIESON
ADAM
ADKINSON
ADKINSON
ADKINSON
ADKINSON
ADKINSON
ADKINSON
ADKINSON
ADKINSON
AECKERLE
AFANASSIEV
AFANASSIEV
```

```
  AHL
  AKROYD
```

Note: *CURS-FIELD および POS(*field-name*) の値は、フィールドの内部 ID の取得にのみ使用します。 算術演算には使用できません。

プログラミングの単純化

システム関数 POS

Natural システム関数 POS(*field-name*) は、指定された名前のフィールドの内部 ID を返します。

POS(*field-name*) を使用すると、マップ内の位置と関係なく、特定のフィールドを特定できます。 これは、マップ内のフィールドのシーケンスと数が変更されても、POS(*field-name*) は引き続き同じフィールドを一意に識別できることを意味します。 これにより、例えば、プログラムロジックに依存しているとフィールドに MARK を付ける場合、必要なのは 1 つの REINPUT ステートメントのみになります。

Note: POS(*field-name*) の値は、フィールドの内部 ID の取得にのみ使用します。 算術演算には使用できません。

例：

```
...
DECIDE ON FIRST VALUE OF ...
   VALUE ...
      COMPUTE #FIELDX = POS(FIELD1)
   VALUE ...
      COMPUTE #FIELDX = POS(FIELD2)
   ...
END-DECIDE
...
REINPUT ... MARK #FIELDX
...
```

*CURS-FIELD および POS(*field-name*) の詳細については、『システム変数』ドキュメントおよび『システム*関数*』ドキュメントを参照してください。

行に基づいた処理

システム変数 *CURS-LINE

システム変数 *CURS-LINE を使用すると、Enter キーが押された時点でカーソルが位置付けられている行に基づいた処理を作成できます。

この変数を使用することにより、ユーザーが使いやすいメニューを作成できます。 適切なプログラミングが行われていれば、ユーザーは実行したいメニューオプションの行にカーソルを移動して Enter キーを押すだけで、そのオプションを実行できます。

カーソルの位置は、画面上の物理的な位置に関係なく、その時点のアクティブなウィンドウ内の位置で定義されます。

Note: メッセージ行、ファンクションキー行、および情報行／統計行は、画面上のデータ行とは見なされません。

以下の例は、*CURS-LINE システム変数を使用した、行に基づいた処理を示しています。マップ上で Enter キーが押されると、オプション"［Employee information］"がある 8 行目にカーソルが配置されているかどうかがプログラムによって確認されます。8行目にカーソルがある場合、従業員名をリストするプログラム LISTEMP が実行されます。

```
DEFINE DATA LOCAL
1 #EMP (A1)
1 #CAR (A1)
1 #CODE (N1)
END-DEFINE
*
INPUT USING MAP 'CURS'
*
DECIDE FOR FIRST CONDITION
  WHEN *CURS-LINE = 8
     FETCH 'LISTEMP'
   WHEN NONE
     REINPUT 'PLACE CURSOR ON LINE OF OPTION YOU WISH TO SELECT'
```

```
END-DECIDE
END
```

出力：

```
                                  Company Information


                     Please select a function


                  [] 1.) Employee information

                     2.) Vehicle information









    Place the cursor on the line of the option you wish to select and press
    ENTER
```

"[]"で表示されているカーソルを希望するオプションに移動してEnterキーを押すと、対応するプログラムが実行されます。

列に基づいた処理

システム変数 *CURS-COL

システム変数*CURS-COLは、前述の*CURS-LINEと同様に使用できます。 *CURS-COLを使用すると、画面上でカーソルが位置付けられている列に基づいた処理を作成できます。

ファンクションキーに基づいた処理

システム変数 *PF-KEY

ユーザーが押したファンクションキーに応じて処理を行う必要がある場合は少なくありません。

これは、ステートメント SET KEY とシステム変数 *PF-KEY を使用し、デフォルトのマップ設定（Standard Keys = "Y"）を変更することによって実現できます。

SET KEY ステートメントは、プログラムの実行中に機能をファンクションキーに割り当てます。システム変数 *PF-KEY には、ユーザーが最後に押したファンクションキーの ID が格納されます。

以下の例は、*PF-KEY と組み合わせた SET KEY の使い方を示しています。

```
...
SET KEY PF1
*
INPUT USING MAP 'DEMO&'
IF *PF-KEY = 'PF1'
  WRITE 'Help is currently not active'
END-IF
...
```

SET KEY ステートメントによって、PF1 キーがファンクションキーとして有効化されています。

IF ステートメントは、ユーザーが PF1 キーを押したときに実行するアクションを定義しています。 システム変数 *PF-KEY の現在の値を確認し、その値が PF1 の場合は、対応するアクションを実行します。

ステートメント SET KEY およびシステム変数 *PF-KEY の詳細については、それぞれ『ステートメント』ドキュメントおよび『システム変数』ドキュメントを参照してください。

ファンクションキー名に基づいた処理

システム変数 *PF-NAME

ファンクションキーに基づいた処理を定義する場合、システム変数 *PF-NAME を使用するとさらに快適に定義できます。 この変数を使用すると、特定のキーではなく、ファンクションの名前に応じた処理を作成できます。

変数 *PF-NAME には、ユーザーが最後に押したファンクションキーの名前、つまり、SET KEY ステートメントの NAMED 節でキーに割り当てた名前が格納されます。

例えば、PF3 キーまたは PF12 キーが押されたらヘルプが起動されるようにする場合、両方のキーに同じ名前（以下の例では "INFO"）を割り当てます。ユーザーがこれらのどちらかのキーを押すと、IF ステートメントで定義した処理が実行されます。

```
...
SET KEY PF3  NAMED 'INFO'
        PF12 NAMED 'INFO'
INPUT USING MAP 'DEMO&'
IF *PF-NAME = 'INFO'
  WRITE 'Help is currently not active'
END-IF
...
```

NAMED 節で定義したファンクション名はファンクションキー行に表示されます。

アクティブなウィンドウの外部からのデータ処理

以下に参考情報を示します。

- システム変数 *COM
- *COM の使用例

- *COM へのカーソルの配置 - %T* 端末コマンド

システム変数 *COM

「*画面設計*」の「*ウィンドウ*」で説明されているとおり、アクティブになれるウィンドウは常に *1* つだけです。これは、通常は、その特定のウィンドウ内でのみ入力が可能だということです。

通信エリアと見なされる *COM システム変数を使用すると、アクティブなウィンドウの外でデータを入力できます。

ただし、マップに *COM が変更可能なフィールドとして含まれていることが前提条件です。この条件が満たされていれば、ウィンドウが画面上に表示されているときは、このフィールドにデータを入力できます。 *COM の内容に応じて、追加の処理を実行できます。

これにより、入力フィールドを持つウィンドウがアクティブな場合でも常にユーザーがコマンド行にデータを入力できる、Software AG のオフィスシステム Con-nect ですでに使用されているようなユーザーインターフェイスを実装できます。

*COM は、Natural セッションが終了するときにのみクリアされます。

*COM の使用例

以下の例では、プログラム ADD は、マップの入力データを使用して単純な加算処理を実行しています。このマップでは、拡張フィールド編集の AL フィールドで指定されている長さの変更可能なフィールドとして、（マップの最後に）*COM は定義されています。計算結果はウィンドウに表示されます。このウィンドウには入力できませんが、ウィンドウ外部のマップ内では引き続き *COM フィールドを使用できます。

プログラム **ADD**：

```
DEFINE DATA LOCAL
1 #VALUE1 (N4)
1 #VALUE2 (N4)
1 #SUM3 (N8)
END-DEFINE
*
DEFINE WINDOW EMP
  SIZE 8*17
  BASE 10/2
  TITLE 'Total of Add'
  CONTROL SCREEN
  FRAMED POSITION SYMBOL BOT LEFT
*
INPUT USING MAP 'WINDOW'
*
COMPUTE #SUM3 = #VALUE1 + #VALUE2
*
```

```
SET WINDOW 'EMP'
INPUT (AD=O) / 'Value 1 +' /
               'Value 2 =' //
               ' ' #SUM3
*
IF *COM = 'M'
  FETCH 'MULTIPLY' #VALUE1 #VALUE2
END-IF
END
```

プログラム ADD の出力：

```
                  Map to Demonstrate Windows with *COM

                                 CALCULATOR

                       Enter values you wish to calculate


                       Value 1:  12__
                       Value 2:  12__
   +-Total of Add-+
   !              !
   ! Value 1 +    !
   ! Value 2 =    !
   !              !
   !          24  !
   !              !
   +--------------+


    Next line is input field (*COM) for input outside the window:
```

この例では、値 "M" を入力すると、乗算が開始されます。入力マップの 2 つの値を乗算した結果が 2 つ目のウィンドウに表示されます。

```
                  Map to Demonstrate Windows with *COM

                                 CALCULATOR

                       Enter values you wish to calculate


                       Value 1:  12__
                       Value 2:  12__
   +-Total of Add-+                              +--------------+
   !              !                              !              !
   ! Value 1 +    !                              ! Value 1 x    !
```

```
   ! Value 2 =    !                                ! Value 2 =    !
   !              !                                !              !
   !          24  !                                !       144    !
   !              !                                !              !
   +--------------+                                +--------------+


    Next line is input field (*COM) for input outside the window:
                 M
```

COM へのカーソルの配置 - %T 端末コマンド

通常、アクティブなウィンドウが入力フィールド（AD=A、または AD=M）を持っていない場合、カーソルはウィンドウの左上隅に配置されます。

アクティブなウィンドウに入力フィールドがない場合、端末コマンド %T* を使用すると、ウィンドウの外にあるシステム変数 *COM にカーソルを配置できます。

%T* を再度使用すると、標準のカーソル位置に戻せます。

例：

```
...
INPUT USING MAP 'WINDOW'
*
COMPUTE #SUM3 = #VALUE1 + #VALUE2
*
SET CONTROL 'T*'
SET WINDOW 'EMP'
INPUT (AD=O) / 'Value 1 +' /
               'Value 2 =' //
               ' ' #SUM3
...
```

画面からのデータのコピー

以下に参考情報を示します。

- 端末コマンド %CS および %CC

端末コマンド %CS および %CC

これらの端末コマンドを使用すると、画面の一部を Natural スタック（%CS）またはシステム変数 *COM（%CC）にコピーできます。 特定の画面行の保護データがフィールドごとにコピーされます。

これらの端末コマンドのすべてのオプションについては、『*端末コマンド*』ドキュメントを参照してください。

スタックまたは *COM にコピーされたデータは、その後の処理で使用できます。 これらのコマンドを使用すると、以下の例のような、使いやすいインターフェイスを作成できます。

後の処理で使用する、レポート出力の行の選択

以下の例では、プログラム COM1 は、Abellan から Alestia までのすべての従業員名をリストしています。

プログラム **COM1**：

```
DEFINE DATA LOCAL
1 EMP VIEW OF EMPLOYEES
  2 NAME(A20)
  2 MIDDLE-NAME (A20)
  2 PERSONNEL-ID (A8)
END-DEFINE
*
READ EMP BY NAME STARTING FROM 'ABELLAN' THRU 'ALESTIA'
  DISPLAY NAME
END-READ
FETCH 'COM2'
END
```

プログラム COM1 の出力：

```
  Page      1                                          2006-08-12  09:41:21

           NAME
  --------------------

  ABELLAN
  ACHIESON
  ADAM
  ADKINSON
  ADKINSON
```

```
  ADKINSON
  ADKINSON
  ADKINSON
  ADKINSON
  ADKINSON
  ADKINSON
  AECKERLE
  AFANASSIEV
  AFANASSIEV
  AHL
  AKROYD
  ALEMAN
  ALESTIA
  MORE
```

プログラムの制御は COM2 に移されています。

プログラム **COM2**：

```
DEFINE DATA LOCAL
1 EMP VIEW OF EMPLOYEES
  2 NAME(A20)
  2 MIDDLE-NAME (A20)
  2 PERSONNEL-ID (A8)
1 SELECTNAME (A20)
END-DEFINE
*
SET KEY PF5 = '%CCC'
*
INPUT NO ERASE 'SELECT FIELD WITH CURSOR AND PRESS PF5'
*
MOVE *COM TO SELECTNAME
FIND EMP WITH NAME = SELECTNAME
  DISPLAY NAME PERSONNEL-ID
END-FIND
END
```

このプログラムでは、端末コマンド %CCC が PF5 に割り当てられています。 この端末コマンドは、カーソルが配置されている行のすべての保護データをシステム変数 *COM にコピーします。 コピーされた情報は、その後の処理で使用できます。 この処理は、太字で記述されているプログラム行に定義されています。

任意の名前にカーソルを合わせて PF5 キーを押すと、詳しい従業員情報が出力されます。

```
SELECT FIELD WITH CURSOR AND PRESS PF5                      2006-08-12  09:44:25

          NAME
  --------------------

  ABELLAN
  ACHIESON
  ADAM  <==  Cursor positioned on name for which more information is required
  ADKINSON
  ADKINSON
  ADKINSON
  ADKINSON
  ADKINSON
  ADKINSON
  ADKINSON
  ADKINSON
  AECKERLE
  AFANASSIEV
  AFANASSIEV
  AHL
  AKROYD
  ALEMAN
  ALESTIA
```

この場合、選択した従業員の個人 ID が表示されます。

```
  Page      1                                                 2006-08-12  09:44:52

          NAME          PERSONNEL
                           ID
  -------------------- ---------

  ADAM                 50005800
```

REINPUT／REINPUT FULL ステートメント

INPUT ステートメントに戻って再度実行する必要がある場合、REINPUT ステートメントを使用します。 これは通常、前の INPUT ステートメントの結果、データ入力が無効であったことを示すメッセージを表示するために使用します。

REINPUT ステートメントで FULL オプションを使用すると、対応する INPUT ステートメントが完全に再実行されます。

- FULL オプションを使用しない通常の REINPUT ステートメントでは、INPUT ステートメントと REINPUT ステートメントの間で変更された変数の内容は表示されません。つまり、画面上のすべての変数は、INPUT ステートメントが初めて実行されたときに保持していた内容を示します。
- REINPUT FULL ステートメントでは、INPUT ステートメントの最初の実行後に行われたすべての変更が、INPUT ステートメントの再実行時に INPUT ステートメント適用されます。つまり、画面上のすべての変数は、REINPUT ステートメントの実行時に保持していた値を含みます。
- カーソルを特定のフィールドに配置する場合、MARK オプションを使用できます。また、特定のフィールド内の特定の位置にカーソルを配置する場合は、MARK POSITION オプションを使用します。

以下の例は、MARK POSITION を使用した REINPUT FULL の使い方を示しています。

```
DEFINE DATA LOCAL
1 #A (A10)
1 #B (N4)
1 #C (N4)
END-DEFINE
*
INPUT (AD=M) #A #B #C
IF #A = ' '
  COMPUTE #B = #B + #C
  RESET #C
  REINPUT FULL 'Enter a value' MARK POSITION 5 IN *#A
END-IF
END
```

フィールド #B に「"3"」、フィールド #C に「"3"」と入力して、Enter キーを押します。

```
  #A            #B      3 #C      3
```

プログラムは、フィールド #A に空白以外の値を必要とします。 MARK POSITION 5 IN *#A を使用した REINPUT FULL ステートメントによって、制御が入力画面に戻ります。COMPUTE で計算が実行されているため、変更可能なフィールド #B に値 6 が設定されています。 カーソルはフィールド #A の 5 番目のポジションに配置され、すぐに入力できるようになっています。

```
Enter name of field
  #A    _       #B      6 #C      0
```

```
Enter a value
```

以下の画面は、FULL オプションを使用しなかった場合に、同じステートメントによって返される画面です。 変数 #B および #C が INPUT ステートメントの実行時の状態に戻されている（各フィールドの値が 3）ことに注意してください。

```
  #A     _        #B      3 #C      3
```

オブジェクト指向の処理 - Natural コマンドプロセッサ

Naturalコマンドプロセッサは、アプリケーション内のナビゲーション（移動操作）を定義および制御するために使用します。 Natural コマンドプロセッサは、開発部分とランタイム部分の 2 つの部分によって構成されています。

- 開発部分は、ユーティリティ SYSNCP です。 このユーティリティを使用すると、コマンド、およびこれらのコマンドの実行に対応するアクションを定義できます。 SYSNCP は、管理者が設定した定義により、コマンド入力時に実行する処理を判断するデシジョンテーブルを生成します。
- ランタイム部分は、ステートメント PROCESS COMMAND です。 このステートメントは、Natural プログラムからコマンドプロセッサを呼び出すために使用します。 このステートメントで、データ入力時の操作に使用する SYSNCP テーブルの名前を指定します。

Natural コマンドプロセッサの詳細については、『ユーティリティ』ドキュメントの「*SYSNCP ユーティリティ*」、および『ステートメント』ドキュメントの「PROCESS COMMAND」を参照してください。

119 Natural Native Interface

このセクションでは、次のトピックについて説明します。

120 はじめに

Natural Native Interfaceを使用すると、Cの呼び出し規則に従うファンクションコールを使用して、アプリケーション独自のプロセスコンテキストで Natural コードを実行できます。 このインターフェイスは、一連のインターフェイス関数が含まれている DLL で構成されています。 インターフェイス関数には、Natural セッションの初期化および初期化解除、特定の Natural ライブラリへのログオン、個々の Natural モジュールの実行などがあります。 呼び出し元のアプリケーションによって、オペレーティングシステムコールでインターフェイスライブラリがダイナミックにロードされ、その後インターフェイス関数が検索されて呼び出されます。

インターフェイスの使用法を示した C プログラムの例 *nnisample.c* は、*%NATDIR%\%NATVERS%\samples\sysexnni* にあります。

Cプログラム *nnisample.c* に呼び出される Natural モジュールは、Natural ライブラリ SYSEXNNI にあります。

121 インターフェイスライブラリおよび位置

このインターフェイスは、一連の関数をエクスポートするDLLで構成されています。 個々の関数については、「*インターフェイス関数*」を参照してください。インターフェイスDLLの名前は *natni.dll* で、Natural *bin* ディレクトリに含まれています。

Natural Native Interfaceを使用するプログラムを実行するときは、Natural *bin* ディレクトリを環境変数 `PATH` に定義し、呼び出し元のプログラムがインターフェイスライブラリおよび依存するすべてのライブラリを検索できるようにする必要があります。

122 インターフェイスバージョン管理

Natural Native Interface は、Natural の今後のバージョンで変更される可能性があります。 修正インターフェイスを提供する Natural バージョンでは、Software AG が決定し、近いうちに発表される時点までは以前のインターフェイスバージョンが並行してサポートされます。 インターフェイスの特定バージョンのインスタンスにアクセスするために、関数 `nni_get_interface` がアプリケーションによって呼び出されます。 アプリケーションでは、必要なインターフェイスバージョン番号を関数に渡し、代わりに関数ポインタが含まれる構造を受け取ります。インターフェイスバージョンを明示的に指定しないで最新のインターフェイスバージョンが要求される場合もあります。

123 インターフェイスアクセス

インターフェイスにアクセスするため、アプリケーションではプラットフォームに依存するシステムコールを使用して、インターフェイスライブラリがロードされます。

次に、プラットフォーム依存のシステムコールを再び使用して、関数 `nni_get_interface` のアドレスが検索されます。 中心関数 `nni_get_interface` のアドレスが検出されると、関数 `nni_get_interface` を呼び出して希望するインターフェイスバージョンを指定し、インターフェイスのインスタンスが要求されます。 この結果得られた構造には、インターフェイス関数ポインタが含まれています。

インターフェイス関数の使用が終わると、アプリケーションではプラットフォーム依存のシステムコールを使用して、インターフェイスライブラリがアンロードされます。

サンプルプログラム *nnisample.c* では、このインターフェイスを紹介しています。 また、インターフェイスライブラリをロードするためのプラットフォーム依存メカニズム、および関数 `nni_get_interface` へのアクセスも同じプログラムで説明されています。

124 インターフェイスインスタンスおよび Natural セッション

関数 nni_get_interface では、Natural Native Interface のインスタンスへのポインタが返されます。 1 つのインターフェイスインスタンスで、一度に 1 つの Natural セッションをホストできます。 アプリケーションでは、任意のインターフェイスインスタンス上で関数 nni_initialize を呼び出すことによって、Natural セッションが初期化されます。 次に、同じインターフェイスインスタンス上で nni_uninitialize を呼び出し、Natural セッションを初期化解除します。 この後は、この同じインターフェイスインスタンス上で新しい Natural セッションを初期化できるようになります。

複数のインターフェイスインスタンス、すなわち複数の Natural セッションをプロセスごとまたはスレッドごとに管理できるかどうかは、実装に依存します。 Windows および UNIX 上での Natural の現在の実装では、一度に 1 つの Natural セッションをホストできます。 そのため、1 つのプロセスで nni_get_interface が呼び出されるたびに、同じインターフェイスインスタンスが発生します。 ただし、この一意のインターフェイスインスタンスを、同時に実行される複数のスレッドで交互に使用することができます。 スレッドの同期化は、インターフェイス関数によって自動的、暗黙的に実行されます。 オプションとして、アプリケーションで明示的に実行することもできます。 このインターフェイスによって、必要な同期関数である nni_enter、nni_try_enter、および nni_leave が提供されます。

125 インターフェイス関数

nni_get_interface

構文

```
int nni_get_interface( int iVersion, void** ppnni_func );
```

この関数は、Natural Native Interface のインスタンスを返します。

この関数は、アプリケーションでプラットフォーム依存のシステムコールを使用してインターフェイスライブラリが取得され、ロードされた後に呼び出されます。 この関数を使用すると、個々のインターフェイス関数を示す関数ポインタが含まれている構造へのポインタが返されます。 構造内で返される関数は、インターフェイスのバージョン間で異なる場合があります。

インターフェイスバージョンを指定する代わりに、定数 NNI_VERSION_CURR を指定することもできます。この場合は、最新のインターフェイスバージョンが常に参照されます。特定の Natural バージョンに属するインターフェイスバージョン番号は、そのバージョンとともに提供されるヘッダーファイル *natni.h* 内で定義されています。 Natural バージョン *n.n* の場合、インターフェイスバージョン番号は NNI_VERSION_*nn* として定義されます。 NNI_VERSION_CURR も NNI_VERSION_*nn* として定義されます。 関数が呼び出された Natural バージョンで、要求されるインターフェイスバージョンがサポートされていない場合は、エラーコード NNI_RC VERSION_ERROR が返されます。 これ以外の場合のリターンコードは、NNI_RC_OK です。

関数によって返されるポインタは、インターフェイスの 1 つのインスタンスを表します。 このインターフェイスインスタンスを使用するため、アプリケーションではこのポインタが保持され、後続のインターフェイスコールに渡されます。

通常、アプリケーションで任意のインスタンスに対して nni_initialize を呼び出すことによって、Natural セッションがその後初期化されます。 アプリケーションで Natural セッションの使用が終了すると、同じインスタンスに対して nni_uninitialize が呼び出されます。 これ以降は、この同じインターフェイスインスタンス上で別の Natural セッションを初期化できるようになります。 アプリケーションでインターフェイスインスタンスの使用が完全に終了した後は、このインスタンスに対して nni_free_interface が呼び出されます。

パラメータ

パラメータ	意味
iVersion	インターフェイスバージョン番号 （NNI_VERSION_*nn* または NNI_VERSION_CURR）。
ppnni_func	リターン時に NNI インターフェイスインスタンスをポイントします。

リターンコード

リターンコードの意味については、「*リターンコード*」の説明を参照してください。

リターンコード	注釈
NNI_RC_OK	
NNI_RC_PARM_ERROR	
NNI_RC_VERSION_ERROR	

nni_free_interface

構文

```
int nni_free_interface(void* pnni_func);
```

この関数は、アプリケーションでインターフェイスインスタンスの使用が終了し、ホストされていたNaturalセッションが初期化解除された後に呼び出されます。この関数によって、インターフェイスインスタンスが占有していたリソースが解放されます。

パラメータ

パラメータ	意味
pnni_func	NNI インターフェイスインスタンスへのポインタ。

リターンコード

リターンコードの意味については、「*リターンコード*」の説明を参照してください。

リターンコード	注釈
NNI_RC_OK	

nni_initialize

構文

```
int nni_initialize(void* pnni_func, const char* szCmdLine, void*, void*);
```

この関数は、任意のコマンド行を使用して Natural セッションを初期化します。コマンド行の構文と語義は、Naturalを対話形式で開始した場合と同じです。すでに任意のインターフェイスインスタンスによってNaturalセッションが初期化されている場合は、新しいセッションが初期化される前に、そのセッションが暗黙的に初期化解除されます。

コマンド行は、ユーザーの操作がなくてもNaturalの初期化が完了するように指定する必要があります。 つまり、プログラムがスタックに渡されたり、起動プログラムが指定されたりしている場合には特に、満たされていない INPUT ステートメントをプログラムでスタックから実行ないでください。 実行した場合は、Natural セッションの後続の動作が不確定になります。

Natural セッションは、バッチセッションとしてサーバーモードで初期化されます。 このことは、実行されたNaturalモジュールにおける特定のステートメントおよびコマンドの使用が制限されることを意味します。 このような制限およびエラー状態は、『*オペレーション*』ドキュメントの「*NaturalX サーバー環境でのステートメントおよびコマンドの使用*」セクションの説明と同じです。

Natural Security で Natural セッションを初期化するときは、セッションを開始するために自由に選択したデフォルトライブラリに対する LOGON コマンド、および適切なユーザー ID やパスワードをコマンド行に含める必要があります。

例：

```
int iRes =
pnni_func->nni_initialize( pnni_func, "STACK=(LOGON,MYLIB,MYUSER,MYPASS)", 0, 0);
```

この後にアプリケーションで異なるユーザー ID を使用して別のライブラリに対して nni_logon を呼び出し、その後 nni_logoff を呼び出すと、nni_initialize で渡されたライブラリおよびユーザー ID に対して Natural セッションがリセットされます。

パラメータ

パラメータ	意味
pnni_func	NNI インターフェイスインスタンスへのポインタ。
szCmdLine	Natural コマンド行。 NULL ポインタを使用できます。
void*	将来的に使用される予定。 NULL ポインタである必要があります。
void*	将来的に使用される予定。 NULL ポインタである必要があります。

リターンコード

リターンコードの意味については、「*リターンコード*」の説明を参照してください。

リターンコード	注釈
NNI_RC_OK	
NNI_RC_PARM_ERROR	
rc（*rc*< NNI_RC_SERR_OFFSET）	Natural スタートアップエラー。『*オペレーション*』ドキュメントの一部である「*Natural スタートアップエラー*」で説明されている実際の Natural スタートアップエラー番号は、次の計算で求めることができます。

リターンコード	注釈
	startup-error-nr = - (*rc* - NNI_RC_SERR_OFFSET) セッションの初期化中に発生する警告は無視されます。
> 0	Natural エラー番号。

nni_is_initialized

構文

```
int nni_is_initialized( void* pnni_func, int* piIsInit );
```

この関数は、インターフェイスインスタンスに初期化済みの Natural セッションが含まれているかどうかをチェックします。

パラメータ

パラメータ	意味
pnni_func	NNI インターフェイスインスタンスへのポインタ。
piIsInit	Natural セッションが初期化されていない場合には 0 が、初期化されている場合には 0 以外の値が返されます。

リターンコード

リターンコードの意味については、「*リターンコード*」の説明を参照してください。

リターンコード	注釈
NNI_RC_OK	
NNI_RC_PARM_ERROR	

nni_uninitialize

構文

```
int nni_uninitialize(void* pnni_func);
```

この関数は、指定したインターフェイスインスタンスによってホストされている Natural セッションを初期化解除します。

パラメータ

パラメータ	意味
pnni_func	NNI インターフェイスインスタンスへのポインタ。

リターンコード

リターンコードの意味については、「*リターンコード*」の説明を参照してください。

リターンコード	注釈
NNI_RC_OK	

nni_enter

構文

```
int nni_enter(void* pnni_func);
```

この関数は、現在のスレッドで、インターフェイスインスタンスおよびホストするNaturalセッションへの排他的アクセスを待機できるようにします。 スレッドは、他のスレッドに中断されない一連のインターフェイスコールを発行するときに、この関数を呼び出します。スレッドは、nni_leave を呼び出して、インターフェイスインスタンスへの排他的アクセスを解放します。

パラメータ

パラメータ	意味
pnni_func	NNI インターフェイスインスタンスへのポインタ。

リターンコード

リターンコードの意味については、「*リターンコード*」の説明を参照してください。

リターンコード	注釈
NNI_RC_OK	

nni_try_enter

構文

```
int nni_try_enter(void* pnni_func);
```

この関数は、nni_enter と同様に機能しますが、スレッドをブロックする代わりに、通常、即座に値を返します。 すでに別のスレッドがインターフェイスインスタンスに排他アクセスを行っている場合は、NNI_RC_LOCKED を返します。

パラメータ

パラメータ	意味
pnni_func	NNI インターフェイスインスタンスへのポインタ。

リターンコード

リターンコードの意味については、「*リターンコード*」の説明を参照してください。

リターンコード	注釈
NNI_RC_OK	
NNI_RC_LOCKED	

nni_leave

構文

```
int nni_leave(void* pnni_func);
```

この関数は、インターフェイスインスタンスへの排他的アクセスを解放し、そのインスタンスおよびホストする Natural セッションに他のスレッドがアクセスできるようにします。

パラメータ

パラメータ	意味
pnni_func	NNI インターフェイスインスタンスへのポインタ。

リターンコード

リターンコードの意味については、「*リターンコード*」の説明を参照してください。

リターンコード	注釈
NNI_RC_OK	

nni_logon

構文

```
int nni_logon(void* pnni_func, const char* szLibrary, const char* szUser, const
char* szPassword);
```

この関数は、指定された Natural ライブラリに対する LOGON を実行します。

パラメータ

パラメータ	意味
pnni_func	NNI インターフェイスインスタンスへのポインタ。
szLibrary	Natural ライブラリの名前。
szUser	Natural ユーザーの名前。 Natural セッションが Natural Security で実行中ではない場合、または初期化中に AUTO=ON が使用された場合は、NULL ポインタを使用できます。
szPassword	上記ユーザーのパスワード。 Natural セッションが Natural Security で実行中ではない場合、または初期化中に AUTO=ON が使用された場合は、NULL ポインタを使用できます。

リターンコード

リターンコードの意味については、「*リターンコード*」の説明を参照してください。

リターンコード	注釈
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
> 0	Natural エラー番号。

nni_logoff

構文

```
int nni_logoff(void* pnni_func);
```

この関数は、現在の Natural ライブラリからの LOGOFF を実行します。 以前のアクティブなライブラリおよびユーザー ID に LOGON することと同じです。

パラメータ

パラメータ	意味
pnni_func	NNI インターフェイスインスタンスへのポインタ。

リターンコード

リターンコードの意味については、「*リターンコード*」の説明を参照してください。

リターンコード	注釈
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
> 0	Natural エラー番号。

nni_callnat

構文

```
int nni_callnat(void* pnni_func, const char* szName, int iParm, struct
parameter_description* rgDesc, struct natural_exception* pExcep);
```

関数が Natural サブプログラムを呼び出します。

関数は、パラメータを parameter_description 構造の配列として受け取ります。 呼び出し元は、NNI 関数を使用して、次の方法でこれらの構造を作成します。

- 関数 create_parm または create_module_parm を使用すると、サブプログラムの適切なパラメータセットを作成できます。
- create_parm を使用した場合、関数 init_parm_* を使用して、各パラメータを適切な Natural データフォーマットに初期化します。 create_module_parm を使用した場合、パラメータはすでに適切な Natural データフォーマットに初期化されています。

- 関数 nni_put_parm または nni_put_parm_array を使用すると、値を各パラメータに割り当てることができます。
- セット内の各パラメータで nni_get_parm を呼び出します。 これにより、parameter_description 構造に値が指定されます。
- parameter_description 構造の配列を関数 nni_callnat に渡します。
- コールが実行されたら、関数 nni_get_parm または nni_get_parm_array を使用して、パラメータセットから修正されたパラメータ値を抽出します。

パラメータ

パラメータ	意味
pnni_func	NNI インターフェイスインスタンスへのポインタ。
szName	Natural サブプログラムの名前。
iParm	パラメータの数。 配列 rgDesc のオカレンス数を示します。
rgDesc	サブプログラムのパラメータを含む、parm_description 構造の配列。 サブプログラムでパラメータが必要ない場合、呼び出し元は NULL ポインタを渡します。
pExcep	natural_exception 構造へのポインタ。 サブプログラムの実行中に Natural エラーが発生すると、この構造に Natural エラー情報が記述されます。 呼び出し元は NULL ポインタを指定できます。 この場合、拡張されたエラー情報は返されません。

リターンコード

リターンコードの意味については、「*リターンコード*」の説明を参照してください。

リターンコード	注釈
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_NO_MEMORY	
> 0	Natural エラー番号。

nni_function

構文

```
int nni_callnat(void* pnni_func, const char* szName, int iParm, struct
parameter_description* rgDesc, struct natural_exception* pExcep);
```

関数が Natural サブプログラムを呼び出します。

関数は、パラメータを parameter_description 構造の配列として受け取ります。 呼び出し元は、NNI 関数を使用して、次の方法でこれらの構造を作成します。

- 関数 create_parm または create_module_parm を使用すると、サブプログラムの適切なパラメータセットを作成できます。
- create_parm を使用した場合、関数 init_parm_* を使用して、各パラメータを適切な Natural データフォーマットに初期化します。 create_module_parm を使用した場合、パラメータはすでに適切な Natural データフォーマットに初期化されています。
- 関数 nni_put_parm または nni_put_parm_array を使用すると、値を各パラメータに割り当てることができます。
- セット内の各パラメータで nni_get_parm を呼び出します。 これにより、parameter_description 構造に値が指定されます。
- parameter_description 構造の配列を関数 nni_callnat に渡します。
- コールが実行されたら、関数 nni_get_parm または nni_get_parm_array を使用して、パラメータセットから修正されたパラメータ値を抽出します。

パラメータ

パラメータ	意味
pnni_func	NNI インターフェイスインスタンスへのポインタ。
szName	Natural サブプログラムの名前。
iParm	パラメータの数。 配列 rgDesc のオカレンス数を示します。
rgDesc	サブプログラムのパラメータを含む、parm_description 構造の配列。 サブプログラムでパラメータが必要ない場合、呼び出し元は NULL ポインタを渡します。
pExcep	natural_exception 構造へのポインタ。 サブプログラムの実行中に Natural エラーが発生すると、この構造に Natural エラー情報が記述されます。 呼び出し元は NULL ポインタを指定できます。 この場合、拡張されたエラー情報は返されません。

リターンコード

リターンコードの意味については、「*リターンコード*」の説明を参照してください。

リターンコード	注釈
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_NO_MEMORY	
> 0	Natural エラー番号。

nni_create_object

構文

```
int nni_create_object(void* pnni_func, const char* szName, int iParm, struct
parameter_description* rgDesc, struct natural_exception* pExcep);
```

Natural オブジェクト（Natural クラスのインスタンス）を作成します。

関数は、パラメータを、parameter_description 構造の1つの要素の配列として受け取ります。呼び出し元は、NNI 関数を使用して、次の方法でこれらの構造を作成します。

- 関数 nni_create_parm を使用すると、1 つの要素を含むパラメータセットを作成できます。
- 関数 nni_init_parm_s を使用すると、タイプ HANDLE OF OBJECT のパラメータを初期化できます。
- このパラメータで nni_get_parm_info を呼び出します。 これにより、parameter_description 構造に値が指定されます。
- parameter_description 構造を関数 nni_create_object に渡します。
- コールが実行された後、関数 nni_get_parm の 1 つを使用して、パラメータセットから修正されたパラメータ値を抽出します。

rgDesc で渡されたパラメータの意味は、次のとおりです。

- 最初かつ唯一のパラメータは、データタイプ HANDLE OF OBJECT で初期化する必要があります。リターン時にはこのパラメータに、新しく作成されたオブジェクトの Natural オブジェクトハンドルが含まれます。

パラメータ

パラメータ	意味
pnni_func	NNI インターフェイスインスタンスへのポインタ。
szName	クラスの名前。
iParm	パラメータの数。 配列 rgDesc のオカレンス数を示します。
rgDesc	オブジェクト作成用のパラメータを含む、parm_description 構造の配列。呼び出し元は、常に 1 つのパラメータを渡します。リターン時にはこのパラメータにオブジェクトハンドルが含まれます。
pExcep	natural_exception 構造へのポインタ。 オブジェクトの作成中に Natural エラーが発生すると、この構造に Natural エラー情報が記述されます。 呼び出し元は NULL ポインタを指定できます。 この場合、拡張された例外情報は返されません。

リターンコード

リターンコードの意味については、「*リターンコード*」の説明を参照してください。

リターンコード	注釈
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_NO_MEMORY	
> 0	Natural エラー番号。

nni_send_method

構文

```
int nni_send_method(void* pnni_func, const char* szName, int iParm, struct
parameter_description* rgDesc, struct natural_exception* pExcep);
```

Natural オブジェクト（Natural クラスのインスタンス）にメソッドコールを送ります。

関数は、パラメータを parameter_description 構造の配列として受け取ります。 呼び出し元は、NNI 関数を使用して、次の方法でこれらの構造を作成します。

- 関数 nni_create_parm または nni_create_method_parm を使用すると、一致するパラメータセットを作成できます。
- create_parm を使用した場合、関数 init_parm_* を使用して、各パラメータを適切な Natural データフォーマットに初期化します。 nni_create_method_parm を使用した場合、パラメータはすでに適切な Natural データフォーマットに初期化されています。
- 関数 nni_put_parm または nni_put_parm_array を使用すると、値を各パラメータに割り当てることができます。
- セット内の各パラメータで nni_get_parm_info を呼び出します。 これにより、parameter_description 構造に値が指定されます。
- parameter_description 構造の配列を関数 nni_send_method に渡します。
- コールが実行されたら、関数 nni_get_parm の 1 つを使用して、パラメータセットから修正されたパラメータ値を抽出します。

rgDesc で渡されたパラメータの意味は、次のとおりです。

- 最初のパラメータには、オブジェクトハンドルが含まれています。
- 2 つ目のパラメータは、メソッドの戻り値のデータタイプに初期化する必要があります。 メソッドに戻り値がない場合、2つ目のパラメータは初期化されないままです。メソッドコールからのリターン時には、このパラメータにメソッドの戻り値が含まれます。
- 残りのパラメータはメソッドパラメータです。

パラメータ

パラメータ	意味
pnni_func	NNI インターフェイスインスタンスへのポインタ。
szName	メソッドの名前。
iParm	パラメータの数。 配列 rgDesc のオカレンス数を示します。 常に 2+ メソッドパラメータの数になります。
rgDesc	メソッドのパラメータを含む、parm_description 構造の配列。 メソッドでパラメータが要求されない場合でも、呼び出し元は 2 つのパラメータを渡します。1 つ目はオブジェクトハンドルに、2 つ目は戻り値に使用されます。
pExcep	natural_exception 構造へのポインタ。 メソッドの実行中に Natural エラーが発生すると、この構造に Natural エラー情報が記述されます。 呼び出し元は NULL ポインタを指定できます。 この場合、拡張された例外情報は返されません。

リターンコード

リターンコードの意味については、「*リターンコード*」の説明を参照してください。

リターンコード	注釈
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_NO_MEMORY	
> 0	Natural エラー番号。

nni_get_property

構文

```
int nni_get_property(void* pnni_func, const char* szName, int iParm, struct
parameter_description* rgDesc, struct natural_exception* pExcep);
```

Natural オブジェクト（Natural クラスのインスタンス）のプロパティ値を取得します。

関数は、パラメータを parameter_description 構造の配列として受け取ります。 呼び出し元は、NNI 関数を使用して、次の方法でこれらの構造を作成します。

- 関数 nni_create_parm または nni_create_method_parm を使用すると、一致するパラメータセットを作成できます。
- create_parm を使用した場合、関数 init_parm_* を使用して、各パラメータを適切な Natural データフォーマットに初期化します。 create_method_parm を使用した場合、パラメータはすでに適切な Natural データフォーマットに初期化されています。

- 関数 nni_put_parm または nni_put_parm_array を使用すると、値を各パラメータに割り当てることができます。
- セット内の各パラメータで nni_get_parm_info を呼び出します。 これにより、parameter_description 構造に値が指定されます。
- parameter_description 構造の配列を関数 nni_send_method に渡します。
- コールが実行されたら、関数 nni_get_parm の 1 つを使用して、パラメータセットから修正されたパラメータ値を抽出します。

rgDesc で渡されたパラメータの意味は、次のとおりです。

- 最初のパラメータには、オブジェクトハンドルが含まれています。
- 2つ目のパラメータは、プロパティのデータタイプに初期化されます。プロパティアクセスからのリターン時には、このパラメータにプロパティ値が含まれています。

パラメータ

パラメータ	意味
pnni_func	NNI インターフェイスインスタンスへのポインタ。
szName	プロパティの名前。
iParm	パラメータの数。 配列 rgDesc のオカレンス数を示します。 これは常に 2 です。
rgDesc	プロパティへのアクセス用のパラメータを含む、parm_description 構造の配列。 呼び出し元は常に 2 つのパラメータを渡します。1 つ目はオブジェクトハンドルに、2 つ目は返されるプロパティ値に使用されます。
pExcep	natural_exception 構造へのポインタ。 プロパティへのアクセス中に Natural エラーが発生すると、この構造に Natural エラー情報が記述されます。 呼び出し元は NULL ポインタを指定できます。 この場合、拡張された例外情報は返されません。

リターンコード

リターンコードの意味については、「*リターンコード*」の説明を参照してください。

リターンコード	注釈
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_NO_MEMORY	
> 0	Natural エラー番号。

nni_set_property

構文

```
int nni_set_property(void* pnni_func, const char* szName, int iParm, struct
parameter_description* rgDesc, struct natural_exception* pExcep);
```

Natural オブジェクト（Natural クラスのインスタンス）にプロパティ値を割り当てます。

関数は、パラメータを parameter_description 構造の配列として受け取ります。 呼び出し元は、NNI 関数を使用して、次の方法でこれらの構造を作成します。

- 関数 nni_create_parm または nni_create_prop_parm を使用して、一致するパラメータセットを作成します。
- create_parm を使用した場合、関数 init_parm_* を使用して、各パラメータを適切な Natural データフォーマットに初期化します。 create_prop_parm を使用した場合、パラメータはすでに適切な Natural データフォーマットに初期化されています。 関数 nni_put_parm の 1 つを使用して、各パラメータに値を割り当てます。
- 関数 nni_put_parm または nni_put_parm_array を使用すると、値を各パラメータに割り当てることができます。
- セット内の各パラメータで nni_get_parm_info を呼び出します。 これにより、parameter_description 構造に値が指定されます。
- parameter_description 構造の配列を関数 nni_set_property に渡します。

rgDesc で渡されたパラメータの意味は、次のとおりです。

- 最初のパラメータには、オブジェクトハンドルが含まれています。
- 2 つ目のパラメータには、プロパティ値が含まれています。

パラメータ

パラメータ	意味
pnni_func	NNI インターフェイスインスタンスへのポインタ。
szName	プロパティの名前。
iParm	パラメータの数。 配列 rgDesc のオカレンス数を示します。 これは常に 2 です。
rgDesc	プロパティへのアクセス用のパラメータを含む、parm_description 構造の配列。 呼び出し元は常に 2 つのパラメータを渡します。1 つ目はオブジェクトハンドルに、2 つ目はプロパティ値に使用されます。
pExcep	natural_exception 構造へのポインタ。 プロパティへのアクセス中に Natural エラーが発生すると、この構造に Natural エラー情報が記述されます。 呼び出し元は NULL ポインタを指定できます。 この場合、拡張された例外情報は返されません。

リターンコード

リターンコードの意味については、「*リターンコード*」の説明を参照してください。

リターンコード	注釈
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_NO_MEMORY	
> 0	Natural エラー番号。

nni_delete_object

構文

```
int nni_delete_object(void* pnni_func, int iParm, struct parameter_description*
rgDesc, struct natural_exception* pExcep);
```

nni_create_object で作成した Natural オブジェクト（Natural クラスのインスタンス）を削除します。

関数は、パラメータを、parameter_description 構造の1つの要素の配列として受け取ります。呼び出し元は、NNI 関数を使用して、次の方法でこれらの構造を作成します。

- 関数 nni_create_parm を使用すると、1 つの要素を含むパラメータセットを作成できます。
- 関数 nni_init_parm_s を使用すると、タイプ HANDLE OF OBJECT のパラメータを初期化できます。
- 関数 nni_put_parm の 1 つを使用して、パラメータに値を割り当てます。
- このパラメータで nni_get_parm_info を呼び出します。これにより、parameter_description 構造に値が指定されます。
- parameter_description 構造を関数 nni_delete_object に渡します。

rgDesc で渡されたパラメータの意味は、次のとおりです。

- 最初かつ唯一のパラメータは、データタイプ HANDLE OF OBJECT で初期化する必要があります。このパラメータには、削除される オブジェクトの Natural オブジェクトハンドルが含まれています。

パラメータ

パラメータ	意味
pnni_func	NNI インターフェイスインスタンスへのポインタ。
szName	クラスの名前。
iParm	パラメータの数。 配列 rgDesc のオカレンス数を示します。 これは常に 1 です。
rgDesc	オブジェクト作成用のパラメータを含む、parm_description 構造の配列。 呼び出し元は常に 1 つのパラメータを渡します。このパラメータにはオブジェクトハンドルが含まれています。
pExcep	natural_exception 構造へのポインタ。 オブジェクトの作成中に Natural エラーが発生すると、この構造に Natural エラー情報が記述されます。 呼び出し元は NULL ポインタを指定できます。 この場合、拡張された例外情報は返されません。

リターンコード

リターンコードの意味については、「*リターンコード*」の説明を参照してください。

リターンコード	注釈
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_NO_MEMORY	
> 0	Natural エラー番号。

nni_create_parm

構文

```
int nni_create_parm(void* pnni_func, int iParm, void** pparmhandle);
```

Natural モジュールに渡すことができるパラメータセットを作成します。

セットに含まれているパラメータは、まだNaturalの特定のデータタイプに初期化されていません。 このパラメータを nni_callnat、nni_create_object、nni_send_method、nni_set_property、または nni_get_property へのコールで使用する前に、次の手順が必要です。

- 関数 nni_init_parm_s、nni_init_parm_sa、nni_init_parm_d、nni_init_parm_da のうちの 1 つを使用して、各パラメータを必要な Natural データタイプに初期化します。
- 関数 nni_put_parm または nni_put_parm_array を使用すると、値を各パラメータに割り当てることができます。
- 関数 nni_get_parm_info を使用して、各パラメータを parm_description 構造に変更します。

パラメータ

パラメータ	意味
pnni_func	NNI インターフェイスインスタンスへのポインタ。
iParm	要求されたパラメータ数。 パラメータの最大数は 32767 です。
pparmhandle	リターン時のパラメータセットへのポインタ。

リターンコード

リターンコードの意味については、「*リターンコード*」の説明を参照してください。

リターンコード	注釈
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_ILL_PNUM	
> 0	Natural エラー番号。

nni_create_module_parm

構文

```
int nni_create_module_parm(void* pnni_func, char chType, const char* szName, void**
pparmhandle);
```

nni_callnat へのコールで使用できるパラメータセットを作成します。 この関数を使用すると、呼び出し可能な Natural モジュールの署名をアプリケーションでダイナミックに検索できます。

返されるセットに含まれているパラメータは、指定されたモジュールのパラメータデータエリアに従って、すでに Natural データタイプに初期化されています。 このパラメータセットを nni_callnat へのコールで使用する前に、次の手順が必要です。

- 関数 nni_put_parm または nni_put_parm_array を使用すると、値を各パラメータに割り当てることができます。
- 関数 nni_get_parm_info を使用して、各パラメータを parm_description 構造に変更します。

パラメータ

パラメータ	意味
pnni_func	NNI インターフェイスインスタンスへのポインタ。
chType	Natural モジュールのタイプ。 常に "N" です（サブプログラムの場合）。
szName	Natural モジュールの名前。
pparmhandle	リターン時のパラメータセットへのポインタ。

リターンコード

リターンコードの意味については、「*リターンコード*」の説明を参照してください。

リターンコード	注釈
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
> 0	Natural エラー番号。

nni_create_method_parm

構文

```
int nni_create_method_parm( void* pnni_func, const char* szClass, const char*
szMethod, void** pparmhandle );
```

nni_send_method へのコールで使用できるパラメータセットを作成します。 この関数を使用すると、Natural クラスのメソッドの署名をアプリケーションでダイナミックに検索できます。

返されるパラメータセットには、メソッドパラメータだけでなく、nni_send_method で必要なその他のパラメータも含まれています。 つまり、メソッドに *n* 個のパラメータがある場合、パラメータセットには *n* + 2 個のパラメータが含まれることになります。

- セットの最初のパラメータは、データタイプ HANDLE OF OBJECT に初期化されます。
- セットの2つ目のパラメータは、メソッドの戻り値のデータタイプに初期化されます。 メソッドに戻り値がない場合、2 つ目のパラメータは初期化されません。
- セットの残りのパラメータは、メソッドパラメータのデータタイプに初期化されます。

このパラメータセットを nni_send_method へのコールで使用する前に、次の手順が必要です。

- 関数 nni_put_parm または nni_put_parm_array を使用すると、値を各パラメータに割り当てることができます。
- 関数 nni_get_parm_info を使用して、各パラメータを parm_description 構造に変更します。

パラメータ

パラメータ	意味
pnni_func	NNI インターフェイスインスタンスへのポインタ。
szClass	Natural クラスの名前。
szMethod	Natural メソッドの名前。
pparmhandle	リターン時のパラメータセットへのポインタ。

リターンコード

リターンコードの意味については、「*リターンコード*」の説明を参照してください。

リターンコード	注釈
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
> 0	Natural エラー番号。

nni_create_prop_parm

構文

```
int nni_create_prop_parm(void* pnni_func, const char* szClass, const char*
szProp,void** pparmhandle);
```

nni_get_property または nni_set_property へのコールで使用できるパラメータセットを作成します。 返されるパラメータセットには、nni_get_property または nni_set_property で必要となるすべてのパラメータが含まれます。この関数を使用すると、Natural クラスのプロパティのデータタイプをアプリケーションで決定できます。

- セットの最初のパラメータは、データタイプ HANDLE OF OBJECT に初期化されます。
- セットの 2 つ目のパラメータは、プロパティのデータタイプに初期化されます。

このパラメータセットを nni_get_property または nni_set_property へのコールで使用する前に、次の手順が必要です。

- 関数 nni_put_parm または nni_put_parm_array を使用すると、値を各パラメータに割り当てることができます。
- 関数 nni_get_parm_info を使用して、各パラメータを parm_description 構造に変更します。

パラメータ

パラメータ	意味
pnni_func	NNI インターフェイスインスタンスへのポインタ。
szClass	Natural クラスの名前。
szProp	Natural プロパティの名前。
pparmhandle	リターン時のパラメータセットへのポインタ。

リターンコード

リターンコードの意味については、「*リターンコード*」の説明を参照してください。

リターンコード	注釈
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
> 0	Natural エラー番号。

nni_parm_count

構文

```
int nni_parm_count( void* pnni_func, void* parmhandle, int* piParm )
```

この関数は、パラメータセット内のパラメータの数を取得します。

パラメータ

パラメータ	意味
pnni_func	NNI インターフェイスインスタンスへのポインタ。
parmhandle	パラメータセットへのポインタ。
piParm	パラメータセット内のパラメータの数を返します。

リターンコード

リターンコードの意味については、「*リターンコード*」の説明を参照してください。

リターンコード	注釈
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	

nni_init_parm_s

構文

```
int nni_init_parm_s(void* pnni_func, int iParm, void* parmhandle, char chFormat,
int iLength, int iPrecision, int iFlags);
```

パラメータセット内のパラメータをスタティックデータタイプに初期化します。

パラメータ

パラメータ	意味
pnni_func	NNI インターフェイスインスタンスへのポインタ。
iParm	パラメータのインデックス。 セットの最初のパラメータには、インデックス 0 が含まれます。
parmhandle	パラメータセットへのポインタ。
chFormat	パラメータの Natural データタイプ。
iLength	パラメータの Natural の長さ。
iPrecision	小数点以下の桁数（NNI_TYPE_NUM および NNI_TYPE_PACK のみ）。
iFlags	パラメータフラグ。 次のフラグを使用できます。 NNI_FLG_PROTECTED

リターンコード

リターンコードの意味については、「*リターンコード*」の説明を参照してください。

リターンコード	注釈
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_ILL_PNUM	
NNI_RC_NO_MEMORY	

リターンコード	注釈
NNI_RC_BAD_FORMAT	
NNI_RC_BAD_LENGTH	

nni_init_parm_sa

構文

```
int nni_init_parm_sa (void* pnni_func, int iParm, void* parmhandle, char chFormat,
int iLength, int iPrecision, int iDim, int* rgiOcc, int iFlags);
```

パラメータセット内のパラメータをスタティックデータタイプの配列に初期化します。

パラメータ

パラメータ	意味
pnni_func	NNI インターフェイスインスタンスへのポインタ。
iParm	パラメータのインデックス。 セットの最初のパラメータには、インデックス 0 が含まれます。
parmhandle	パラメータセットへのポインタ。
chFormat	パラメータの Natural データタイプ。
iLength	パラメータの Natural の長さ。
iPrecision	小数点以下の桁数（NNI_TYPE_NUM および NNI_TYPE_PACK のみ）。
iDim	パラメータの配列次元。
rgiOcc	各次元のオカレンス数を示す、int 値の 3 次元配列。 未使用の次元のオカレンス数は、0 として指定する必要があります。
iFlags	パラメータフラグ。 次のフラグを使用できます。 NNI_FLG_PROTECTED NNI_FLG_LBVAR_0 NNI_FLG_UBVAR_0 NNI_FLG_LBVAR_1 NNI_FLG_UBVAR_1 NNI_FLG_LBVAR_2 NNI_FLG_UBVAR_2 NNI_FLG_*VAR* フラグの 1 つが設定されている場合、配列は X-array です。 各次元で、下限または上限のいずれかだけを変数にすることができます（両方を変数にすることはできません）。 このため、例えば、フラグ IF4_FLG_LBVAR_0 を IF4_FLG_UBVAR_0 と組み合わせることはできません。

リターンコード

リターンコードの意味については、「*リターンコード*」の説明を参照してください。

リターンコード	注釈
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_ILL_PNUM	
NNI_RC_NO_MEMORY	
NNI_RC_BAD_FORMAT	
NNI_RC_BAD_LENGTH	
NNI_RC_BAD_DIM	
NNI_RC_BAD_BOUNDS	

nni_init_parm_d

構文

```
int nni_init_parm_d(void* pnni_func, int iParm, void* parmhandle, char chFormat,
int iFlags);
```

パラメータセット内のパラメータをダイナミックデータタイプに初期化します。

パラメータ

パラメータ	意味
pnni_func	NNI インターフェイスインスタンスへのポインタ。
iParm	パラメータのインデックス。 セットの最初のパラメータには、インデックス 0 が含まれます。
parmhandle	パラメータセットへのポインタ。
chFormat	パラメータの Natural データタイプ（NNI_TYPE_ALPHA または NNI_TYPE_BIN）。
iFlags	パラメータフラグ。 次のフラグを使用できます。 NNI_FLG_PROTECTED

リターンコード

リターンコードの意味については、「*リターンコード*」の説明を参照してください。

リターンコード	注釈
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_ILL_PNUM	
NNI_RC_NO_MEMORY	
NNI_RC_BAD_FORMAT	

nni_init_parm_da

構文

```
int nni_init_parm_da (void* pnni_func, int iParm, void* parmhandle, char chFormat,
int iDim, int* rgiOcc, int iFlags);
```

パラメータセット内のパラメータをダイナミックデータタイプの配列に初期化します。

パラメータ

パラメータ	意味
pnni_func	NNI インターフェイスインスタンスへのポインタ。
iParm	パラメータのインデックス。 セットの最初のパラメータには、インデックス 0 が含まれます。
parmhandle	パラメータセットへのポインタ。
chFormat	パラメータの Natural データタイプ（NNI_TYPE_ALPHA または NNI_TYPE_BIN）。
iDim	パラメータの配列次元。
rgiOcc	各次元のオカレンス数を示す、int 値の 3 次元配列。 未使用の次元のオカレンス数は、0 として指定する必要があります。
iFlags	パラメータフラグ。 次のフラグを使用できます。 NNI_FLG_PROTECTED NNI_FLG_LBVAR_0 NNI_FLG_UBVAR_0 NNI_FLG_LBVAR_1 NNI_FLG_UBVAR_1 NNI_FLG_LBVAR_2 NNI_FLG_UBVAR_2 NNI_FLG_*VAR* フラグの 1 つが設定されている場合、配列は X-array です。 各次元で、下限または上限のいずれかだけを変数にすることができます（両方を変数にすることはでき

パラメータ	意味
	ません）。 このため、例えば、フラグ IF4_FLG_LBVAR_0 を IF4_FLG_UBVAR_0 と組み合わせることはできません。

リターンコード

リターンコードの意味については、「*リターンコード*」の説明を参照してください。

リターンコード	注釈
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_ILL_PNUM	
NNI_RC_NO_MEMORY	
NNI_RC_BAD_FORMAT	
NNI_RC_BAD_DIM	
NNI_RC_BAD_BOUNDS	

nni_get_parm_info

構文

```
int nni_get_parm_info (void* pnni_func, int iParm, void* parmhandle, struct
parameter_description* pDesc);
```

パラメータセット内の特定のパラメータに関する詳細情報を返します。

パラメータ

パラメータ	意味
pnni_func	NNI インターフェイスインスタンスへのポインタ。
iParm	パラメータのインデックス。 セットの最初のパラメータには、インデックス 0 が含まれます。
parmhandle	パラメータセットへのポインタ。
pDesc	パラメータ記述構造

リターンコード

リターンコードの意味については、「*リターンコード*」の説明を参照してください。

リターンコード	注釈
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_ILL_PNUM	

nni_get_parm

構文

```
int nni_get_parm(void* pnni_func, int iParm, void* parmhandle, int iBufferLength,
void* pBuffer);
```

パラメータセット内の特定のパラメータの値を返します。pBuffer で指定されたアドレスのバッファに、iBufferLength に指定されたサイズで値が返されます。 値が正常に返されると、このバッファには Natural 内部フォーマットのデータが含まれます。 バッファの内容を解釈する方法については、「***Natural** データタイプ*」を参照してください。

Natural データタイプに基づくパラメータの長さが iBufferLength よりも長い場合は、Natural によってデータが指定された長さに切り捨てられ、コード NNI_RC_DATA_TRUNC が返されます。呼び出し元は、関数 nni_get_parm_info を使用して、パラメータ値の長さを事前に要求できます。

Natural データタイプに基づくパラメータの長さが iBufferLength よりも短い場合は、Natural によってパラメータの長さまでバッファが埋められ、リターンコードにコピーされたデータの長さが返されます。

パラメータが配列である場合は、バッファ内の配列全体が返されます。 このことは、固定サイズ要素の固定サイズ配列に対してのみ意味があります。これは、これ以外の場合には呼び出し元がバッファの内容を解釈できないためです。 任意の配列の個別オカレンスを取得するには、関数 nni_get_parm_array を使用します。

pBuffer に指定されたアドレスに割り当てられた iBufferLength で、メモリのサイズが指定されていない場合、操作の結果は予測できません。 Natural では、pBuffer が NULL ではないことのみがチェックされます。

パラメータ

パラメータ	意味
pnni_func	NNI インターフェイスインスタンスへのポインタ。
iParm	パラメータのインデックス。 セットの最初のパラメータには、インデックス 0 が含まれます。
parmhandle	パラメータセットへのポインタ。
iBufferLength	pBuffer で指定されたバッファの長さ。
pBuffer	値が返されるバッファ。

リターンコード

リターンコードの意味については、「*リターンコード*」の説明を参照してください。

リターンコード	注釈
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_ILL_PNUM	
NNI_RC_DATA_TRUNC	
= *n* (*n* > 0)	操作が成功しました。ただし、バッファに返されたのは *n* バイトのみでした。

nni_get_parm_array

構文

```
int nni_get_parm_array(void* pnni_func, int parmnum, void* parmhandle, int
iBufferLength, void* pBuffer, int* rgiInd);
```

パラメータセット内の特定の配列パラメータの特定のオカレンスの値を返します。nni_get_parm との唯一の違いは、配列インデックスを指定できる点です。未使用の次元のインデックス数は、0 として指定する必要があります。

パラメータ

パラメータ	意味
pnni_func	NNI インターフェイスインスタンスへのポインタ。
iParm	パラメータのインデックス。 セットの最初のパラメータには、インデックス 0 が含まれます。
parmhandle	パラメータセットへのポインタ。
iBufferLength	pBuffer で指定されたバッファの長さ。
pBuffer	値が返されるバッファ。

パラメータ	意味
rgiInd	特定の配列オカレンスを示す、int 値の 3 次元配列。 インデックスは 0 で始まります。

リターンコード

リターンコードの意味については、「*リターンコード*」の説明を参照してください。

リターンコード	注釈
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_ILL_PNUM	
NNI_RC_DATA_TRUNC	
NNI_RC_NOT_ARRAY	
NNI_RC_BAD_INDEX_0	
NNI_RC_BAD_INDEX_1	
NNI_RC_BAD_INDEX_2	
= *n* （*n* > 0）	操作が成功しました。ただし、返されたのは *n* バイトのみでした。

nni_get_parm_array_length

構文

```
int nni_get_parm_array_length(void* pnni_func, int iParm, void* parmhandle, int*
piLength, int* rgiInd);
```

パラメータセット内の特定の配列パラメータの特定のオカレンスの長さを返します。

パラメータ

パラメータ	意味
pnni_func	NNI インターフェイスインスタンスへのポインタ。
iParm	パラメータのインデックス。 セットの最初のパラメータには、インデックス 0 が含まれます。
parmhandle	パラメータセットへのポインタ。
piLength	返される値の長さの int 値へのポインタ。
rgiInd	特定の配列オカレンスを示す、int 値の 3 次元配列。 インデックスは 0 で始まります。

リターンコード

リターンコードの意味については、「*リターンコード*」の説明を参照してください。

リターンコード	注釈
NNI_RC_OK	
NNI_RC_ILL_PNUM	
NNI_RC_DATA_TRUNC	
NNI_RC_NOT_ARRAY	
NNI_RC_BAD_INDEX_0	
NNI_RC_BAD_INDEX_1	
NNI_RC_BAD_INDEX_2	

nni_put_parm

構文

```
int nni_put_parm(void* pnni_func, int iParm, void* parmhandle, int iBufferLength,
const void* pBuffer);
```

パラメータセット内の特定のパラメータに値を割り当てます。 pBuffer で指定されたアドレスのバッファ内の関数に、iBufferLength に指定されたサイズで値が渡されます。 バッファの内容を作成する方法については、「***Natural データタイプ***」を参照してください。

Natural データタイプに基づくパラメータの長さが、指定されたバッファの長さよりも短い場合、データは指定されたパラメータの長さに切り捨てられます。 バッファの残りは無視されます。 Naturalデータタイプに基づくパラメータの長さが、指定されたバッファの長さよりも長い場合、データは指定されたバッファの長さまでしかコピーされず、パラメータ値の残りはそのまま残ります。 Natural データタイプの内部的な長さについては、「***Natural データタイプ***」を参照してください。

パラメータがダイナミック変数である場合は、指定されたバッファの長さに従って、自動的にサイズ変更されます。

パラメータが配列である場合、関数ではバッファ内の配列全体が要求されます。 このことは、固定サイズ要素の固定サイズ配列に対してのみ意味があります。これは、これ以外の場合には呼び出し元がバッファの内容を解釈できないためです。 任意の配列の個々のオカレンスの値を割り当てるには、関数 nni_put_parm_array を使用します。

パラメータ

パラメータ	意味
pnni_func	NNI インターフェイスインスタンスへのポインタ。
iParm	パラメータのインデックス。 セットの最初のパラメータには、インデックス 0 が含まれます。
parmhandle	パラメータセットへのポインタ。
iBufferLength	pBuffer で指定されたバッファの長さ。
pBuffer	値が渡されるバッファ。

リターンコード

リターンコードの意味については、「*リターンコード*」の説明を参照してください。

リターンコード	注釈
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_ILL_PNUM	
NNI_RC_WRT_PROT	
NNI_RC_DATA_TRUNC	
NNI_RC_NO_MEMORY	
= *n* (*n* > 0)	操作が成功しました。ただし、使用されたバッファは *n* バイトののみでした。

nni_put_parm_array

構文

```
int nni_put_parm_array(void* pnni_func, int iParm, void* parmhandle, int
iBufferLength, const void* pBuffer, int* rgiInd);
```

パラメータセット内の特定の配列パラメータの特定のオカレンスに値を割り当てます。
nni_get_parm との唯一の違いは、配列インデックスを指定できる点です。 未使用の次元のインデックス数は、0 として指定する必要があります。

パラメータ

パラメータ	意味
pnni_func	NNI インターフェイスインスタンスへのポインタ。
iParm	パラメータのインデックス。 セットの最初のパラメータには、インデックス 0 が含まれます。
parmhandle	パラメータセットへのポインタ。
iBufferLength	pBuffer で指定されたバッファの長さ。
pBuffer	値が渡されるバッファ。
rgiInd	特定の配列オカレンスを示す、int 値の 3 次元配列。 インデックスは 0 で始まります。

リターンコード

リターンコードの意味については、「*リターンコード*」の説明を参照してください。

リターンコード	注釈
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_ILL_PNUM	
NNI_RC_WRT_PROT	
NNI_RC_DATA_TRUNC	
NNI_RC_NO_MEMORY	
NNI_RC_NOT_ARRAY	
NNI_RC_BAD_INDEX_0	
NNI_RC_BAD_INDEX_1	
NNI_RC_BAD_INDEX_2	
= *n* (*n* > 0)	操作が成功しました。ただし、使用されたバッファは *n* バイトののみでした。

nni_resize_parm_array

構文

```
int nni_resize_parm_array(void* pnni_func, int iParm, void* parmhandle, int* rgiOcc);
```

パラメータセット内の特定の X-array パラメータのオカレンス数を変更します。 *n* 次元配列では、すべての *n* 次元についてオカレンス数を指定する必要があります。 配列の次元が 3 次元未満の場合は、値 0 を使用しない次元に指定する必要があります。

この関数では、任意のX-arrayについて下限または上限の適切な方を変更することによって、各次元のオカレンス数のサイズ変更が試行されます。

パラメータ

パラメータ	意味
pnni_func	NNI インターフェイスインスタンスへのポインタ。
iParm	パラメータのインデックス。 セットの最初のパラメータには、インデックス 0 が含まれます。
parmhandle	パラメータセットへのポインタ。
rgiOcc	配列の新しいオカレンス数を示す int 値の 3 次元配列。

リターンコード

リターンコードの意味については、「*リターンコード*」の説明を参照してください。

リターンコード	注釈
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
NNI_RC_ILL_PNUM	
NNI_RC_WRT_PROT	
NNI_RC_DATA_TRUNC	
NNI_RC_NO_MEMORY	
NNI_RC_NOT_ARRAY	
NNI_RC_NOT_RESIZABLE	
> 0	Natural エラー番号。

nni_delete_parm

構文

```
int nni_delete_parm(void* pnni_func, void* parmhandle);
```

指定されたパラメータセットを削除します。

パラメータ

パラメータ	意味
pnni_func	NNI インターフェイスインスタンスへのポインタ。
parmhandle	パラメータセットへのポインタ。

リターンコード

リターンコードの意味については、「*リターンコード*」の説明を参照してください。

リターンコード	注釈
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	

nni_from_string

構文

```
int nni_from_string(void* pnni_func, const char* szString, char chFormat, int
iLength, int iPrecision, int iBufferLength, void* pBuffer);
```

Natural P 値、N 値、D 値、または T 値の文字列表現を、関数 nni_get_parm、nni_get_parm_array、nni_put_parm、および nni_put_parm_array で使用されるような値の内部表現に変換します。

これらの Natural データタイプの文字列表現は、次のようになります。

フォーマット	文字列表現
P、N	"-3.141592" など。ここでは、DC パラメータで定義した小数点文字が使用されます。
D	DTFORM パラメータで定義した日付フォーマット（例：DTFORM=I の場合は「2004-07-06」）。
T	DTFORM パラメータで定義した日付フォーマットを hh:ii:ss:t 形式の時刻の値と組み合わせたもの（例：DTFORM=I であれば "2004-07-06 11:30:42:7"）、または hh:ii:ss:t 形式の時刻の値（例："11:30:42:7"）。

パラメータ

パラメータ	意味
pnni_func	NNI インターフェイスインスタンスへのポインタ。
szString	値の文字列表現。
chFormat	値の Natural データタイプ。
iLength	値の Natural の長さ。 NNI_TYPE_NUM および NNI_TYPE_PACK の場合の有効桁の総数。これ以外の場合は、0。
iPrecision	NNI_TYPE_NUM および NNI_TYPE_PACK の場合の小数点以下の桁数。これ以外の場合は、0。
iBufferLength	pBuffer で指定されたバッファの長さ。
pBuffer	リターン時に値の内部表現が含まれるバッファ。 このバッファには、値の Natural 内部表現を十分に格納できる大きさが必要です。 必要なサイズについては、「*ユーザー定義変数のフォーマットおよび長さ*」に記載されています。

リターンコード

リターンコードの意味については、「*リターンコード*」の説明を参照してください。

リターンコード	注釈
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
> 0	Natural エラー番号

nni_to_string

構文

```
int nni_to_string(void* pnni_func, int iBufferLength, const void* pBuffer, char
chFormat, int iLength, int iPrecision, int iStringLength, char* szString);
```

Natural P 値、N 値、D 値、または T 値の内部表現を、関数 nni_get_parm、nni_get_parm_array、nni_put_parm、および nni_put_parm_array で使用されるような文字列表現に変換します。

これらの Natural データタイプの文字列表現は、関数 nni_from_string で説明したものと同じです。

パラメータ

パラメータ	意味
pnni_func	NNI インターフェイスインスタンスへのポインタ。
iBufferLength	pBuffer で指定されたバッファの長さ。
pBuffer	値の内部表現が含まれるバッファ。必要なサイズについては、「*ユーザー定義変数のフォーマットおよび長さ*」に記載されています。
chFormat	値の Natural データタイプ。
iLength	値の Natural の長さ。NNI_TYPE_NUM および NNI_TYPE_PACK の場合の有効桁の総数。これ以外の場合は、0。
iPrecision	NNI_TYPE_NUM および NNI_TYPE_PACK の場合の小数点以下の桁数。これ以外の場合は、0。
iStringLength	szString で指定される文字列バッファに最後のゼロを加えた長さ。
szString	リターン時に値の文字列表現が含まれている文字列バッファ。この文字列バッファには、最後のゼロを含む外部表現を十分に格納できる大きさが必要です。

リターンコード

リターンコードの意味については、「*リターンコード*」の説明を参照してください。

リターンコード	注釈
NNI_RC_OK	
NNI_RC_NOT_INIT	
NNI_RC_PARM_ERROR	
> 0	Natural エラー番号

126 パラメータ記述構造

インターフェイスでは、parameter_description という名前の構造に Natural サブプログラムまたはメソッドのパラメータに関する情報が提供されています。 この構造は、ヘッダーファイル *natuser.h* で定義されています。 このファイルは、ディレクトリ *%NATDIR%\%NATVERS%\samples\sysexnni* にあります。

parameter_description 構造の配列は、nni_callnat および同様の関数へのコールごとにインターフェイスに渡されます。 parameter_description 構造は、関数 nni_get_parm_info を使用して、パラメータセット内のパラメータから作成されます。

この構造の関連要素には、次の情報が含まれています。 この表にリストされていない要素は、すべて内部でのみ使用されます。

フォーマット	要素名	内容
void*	address	パラメータ値のアドレス。再割り当てや解放はできません。address 要素は、ダイナミック変数の配列および X-array に対しては NULL ポインタになります。 この場合は、配列データ全体にアクセスすることはできず、パラメータアクセス関数 nni_get_parm を使用して要素ごとにアクセスすることのみが可能です。
int	format	パラメータの Natural データタイプ。 詳細については、「*Natural データタイプ*」を参照してください。
int	length	パラメータ値の Natural の長さ。 データタイプ NNI_TYPE_ALPHA および NNI_TYPE_UNICODE の場合は文字数。 データタイプ NNI_TYPE_PACK および NNI_TYPE_NUM の場合は小数点文字の前にある桁数。 配列の場合は 1 つのオカレンスの長さ。 ダイナミック変数の配列の場合は長さが 0 で示されます。次に、関数 nni_get_parm_array_length を使用して、個別のオカレンスの長さを決定する必要があります。
int	precision	データタイプ NNI_TYPE_PACK および NNI_TYPE_NUM の場合は、小数点文字の後にある桁数。これ以外では 0。

フォーマット	要素名	内容
int	byte_length	パラメータ値の長さ（バイト単位）。 配列の場合は 1 つのオカレンスの長さ（バイト単位）。 ダイナミック変数の配列の場合は長さが 0 バイトで示されます。 次に、関数 nni_get_parm_array_length を使用して、個別のオカレンスの長さを決定する必要があります。
int	dimensions	次元の数。 スカラーの場合は 0。 次元の最大数は 3 です。
int	length_all	パラメータ値の合計長（バイト単位）。 配列の場合は配列全体の長さ（バイト単位）。 ダイナミック変数の配列の場合は合計長が 0 で示されます。 次に、関数nni_get_parm_array_lengthを使用して、個別のオカレンスの長さを決定する必要があります。
int	flags	パラメータフラグ。「フラグ」を参照してください。
int	occurrences[10]	各次元におけるオカレンス数。最初の3つのオカレンスだけが使用されます。
int	indexfactors[10]	各次元の配列インデックスファクタ。 最初の 3 つのオカレンスだけが使用されます。

変数の範囲が固定された固定長の配列の場合、配列の内容には構造要素 address を使ってダイレクトにアクセスできます。 この場合は、次のことが適用されます。

- 3 次元配列の要素（i、j、k）のアドレスは次のように計算されます。

  elementaddress = address + i * indexfactors[0] + j * indexfactors[1] + k * indexfactors[2]
- 2 次元配列の要素（i、j）のアドレスは、次のように計算されます。

  elementaddress = address + i * indexfactors[0] + j * indexfactors[1]
- 1 次元配列の要素（i）のアドレスは次のように計算されます。

  elementaddress = address + i * indexfactors[0]

127 Natural データタイプ

パラメータアクセス関数の一部（nni_get_parm や nni_put_parm など）では、正しい表現のパラメータ値が含まれているバッファが使用されます。 バッファの長さは Natural データタイプによって異なります。 バッファのデータフォーマットは、次の表に従って定義されます。

Natural データタイプ	バッファフォーマット
A	char[ ]
B	byte[ ]
C	short
F4	float
F8	double
I1	signed char
I2	short
I4	int
L	NNI_L_TRUE または NNI_L_FALSE。*natni.h* を参照。
HANDLE OF OBJECT	byte[8]
P、N、D、T	バッファの内容は、関数 nni_from_string を使用して文字列表現から作成する必要があります。 この内容は、関数 nni_to_string を使用して文字列表現に変換できます。
U	UTF-16 文字の配列。 Windows および wchar が UTF-16 文字に相当する UNIX プラットフォームでは wchar[] になります。

パラメータアクセス関数の一部（nni_get_parmおよびnni_put_parmなど）では、Naturalデータタイプを指定する必要があります。 この場合は、次に示す定数を使用する必要があります。 定数はヘッダーファイル *natni.h* で定義されます。 このファイルは、ディレクトリ *%NATDIR%\%NATVERS%\samples\sysexnni* にあります。

Natural データタイプ	定数
A	NNI_TYPE_ALPHA
B	NNI_TYPE_BIN
C	NNI_TYPE_CV
D	NNI_TYPE_DATE
F	NNI_TYPE_FLOAT
I	NNI_TYPE_INT
L	NNI_TYPE_LOG
N	NNI_TYPE_NUM
HANDLE OF OBJECT	NNI_TYPE_OBJECT
P	NNI_TYPE_PACK
T	NNI_TYPE_TIME
U	NNI_TYPE_UNICODE

128 フラグ

構造 parameter_description には、パラメータの状態に関する情報が含まれている要素 flags があります。 関数 nni_init_parm* を使用して、パラメータの初期化中にこれらのフラグの一部を指定することもできます。 個々のフラグは、要素 flags 内で論理 OR と組み合わせることができます。 ヘッダーファイル *natni.h* には、次のフラグが定義されています。 このファイルは、ディレクトリ *%NATDIR%\%NATVERS%\samples\sysexnni* にあります。

リターンコード	意味
NNI_FLG_PROTECTED	パラメータは書き込み保護されています。
NNI_FLG_DYNAMIC (*)	パラメータはダイナミック（可変長または X-array）です。
NNI_FLG_NOT_CONTIG (*)	配列は連続していません。
NNI_FLG_AIV (*)	パラメータは AIV 変数または INDEPENDENT 変数です。
NNI_FLG_DYNVAR (*)	パラメータの長さは可変です。
NNI_FLG_XARRAY (*)	パラメータは X-array です。
NNI_FLG_LBVAR_0	次元 0 の下限は可変です。
NNI_FLG_UBVAR_0	次元 0 の上限は可変です。
NNI_FLG_LBVAR_1	次元 1 の下限は可変です。
NNI_FLG_UBVAR_1	次元 1 の上限は可変です。
NNI_FLG_LBVAR_2	次元 2 の下限は可変です。
NNI_FLG_UBVAR_2	次元 2 の上限は可変です。

関数 nni_init_parm* に明示的に設定できるのは、"（*）" でマークされたフラグのみです。 その他のフラグは、パラメータのタイプに従って、インターフェイスによって自動的に設定されます。

NNI_FLG_*VAR* フラグの 1 つが設定されている場合、配列は X-array です。 X-array の各次元では、下限または上限の両方ではなくどちらかのみを可変にすることができます。 このため、例えば、フラグ NNI_FLG_LBVAR_0 は NNI_FLG_UBVAR_0 と組み合わせることはできません。

NNI_FLG_DYNAMIC がオンになっているときは、NNI_FLG_DYNVAR または NNI_FLG_XARRAY、あるいはこの両方もオンになります。 両方がオンの場合、パラメータは可変長の要素が含まれる X-array になります。

129 リターンコード

インターフェイス関数では、次のリターンコードが返されます。定数はヘッダーファイル *natni.h* で定義されます。このファイルは、ディレクトリ *%NATDIR%\%NATVERS%\samples\sysexnni* にあります。

リターンコード	意味
NNI_RC_OK	実行が成功しました。
NNI_RC_ILL_PNUM	無効なパラメータ番号。
NNI_RC_INT_ERROR	内部エラーが発生しました。
NNI_RC_DATA_TRUNC	パラメータ値のアクセス中にデータが切り捨てられました。
NNI_RC_NOT_ARRAY	パラメータが配列ではありません。
NNI_RC_WRT_PROT	パラメータは書き込み保護されています。
NNI_RC_NO_MEMORY	メモリの割り当てに失敗しました。
NNI_RC_BAD_FORMAT	無効な Natural データタイプ。
NNI_RC_BAD_LENGTH	無効な長さまたは精度。
NNI_RC_BAD_DIM	無効な次元数。
NNI_RC_BAD_BOUNDS	無効な X-array 範囲定義。
NNI_RC_NOT_RESIZABLE	要求された方法では配列のサイズを変更できません。
NNI_RC_BAD_INDEX_0	配列次元 0 のインデックスが範囲外です。
NNI_RC_BAD_INDEX_1	配列次元 1 のインデックスが範囲外です。
NNI_RC_BAD_INDEX_2	配列次元 2 のインデックスが範囲外です。
NNI_RC_VERSION_ERROR	要求されたインターフェイスバージョンはサポートされていません。
NNI_RC_NOT_INIT	このインターフェイスインスタンスで初期化された Natural セッションはありません。
NNI_RC_NOT_IMPL	このインターフェイスバージョンでは実装されていない関数です。
NNI_RC_PARM_ERROR	必須パラメータが指定されていません。

リターンコード	意味
NNI_RC_LOCKED	インターフェイスインスタンスが他のスレッドによってロックされています。
rc（*rc* < NNI_RC_SERR_OFFSET）	Naturalスタートアップエラーが発生しました。『*オペレーション*』ドキュメントの一部である「*Natural* スタートアップエラー」で説明されているNaturalスタートアップエラー番号は、次の計算でリターンコードから求めることができます。 *startup-error-nr* = - (*rc* - NNI_RC_SERR_OFFSET)
> 0	Natural エラー番号。

130 Natural 例外構造

Naturalコードを実行する`nni_callnat`などのインターフェイス関数は、発生した可能性があるNatural エラーに関する詳細情報が含まれている`natural_exception`と呼ばれる構造を返します。この構造は、ヘッダーファイル*natni.h*で定義されています。このファイルは、ディレクトリ*%NATDIR%\%NATVERS%\samples\sysexnni*にあります。

構造の要素には、次の情報が含まれています。

フォーマット	要素名	内容
int	natMessageNumber	Natural のメッセージ番号。
char	natMessageText[NNI_LEN_TEXT+1]	すべての置換が含まれる Natural メッセージテキスト。
char	natLibrary[NNI_LEN_LIBRARY+1]	Natural ライブラリ名。
char	natMember[NNI_LEN_MEMBER+1]	Natural メンバ名。
char	natName[NNI_LEN_NAME+1];	Natural関数名、サブルーチン名、またはクラス名。
char	natMethod[NNI_LEN_NAME+1];	Natural メソッド名またはプロパティ名。
int	int natLine;	エラーが発生した Natural コード行。

131 インターフェイスの使用

インターフェイスの一般的な使用法は、次のとおりです。ここではNaturalサブプログラムの呼び出しを例に説明します。

1. Natural バイナリの場所を決定します。
2. Natural Native Interface ライブラリをロードします。
3. `nni_get_interface` を呼び出して、インターフェイスインスタンスを取得します。
4. `nni_initialize` を呼び出して、Natural セッションを初期化します。
5. `nni_logon` を呼び出して、特定の Natural ライブラリにログオンします。
6. `nni_create_parm` または関連する関数を呼び出して、パラメータセットを作成します。
7. 各パラメータについて、次の手順を実行します。
   - `nni_init_parm` 関数の 1 つを呼び出して、パラメータを正しいタイプに初期化します。
   - `nni_put_parm` 関数の 1 つを呼び出して、パラメータに値を割り当てます。
   - `nni_get_parm_info` を呼び出して、`parameter_description` 構造を作成します。
8. `nni_callnat` を呼び出して、サブプログラムを呼び出します。
9. 変更可能な各パラメータについて、次の手順を実行します。
   - `nni_get_parm` 関数の 1 つを呼び出して、パラメータ値を取得します。
10. `nni_delete_parm` を呼び出して、パラメータ構造を解放します。
11. `nni_uninitialize` を呼び出して、Natural セッションを初期化解除します。
12. `nni_logoff` を呼び出して、前のライブラリに戻ります。
13. `nni_free_interface` を呼び出して、インターフェイスインスタンスを解放します。

インターフェイスの使用法を示した C プログラムの例 *nnisample.c* は、*%NATDIR%\%NATVERS%\samples\sysexnni* にあります。

132 スレッドの問題

Windows および UNIX 上での Natural プロセスには、Natural コードを実行するスレッドが、常に 1 つのみ含まれています。 したがって、対話形式で開始された Natural セッションで、Natural コードを複数のスレッドが並行して実行しようとすることはありません。 複数のスレッドを並行して実行するクライアントプログラムが Natural Native Interface を使用する場合は、状況が異なります。

Natural Native Interface は、マルチスレッドアプリケーションで使用可能です。 インターフェイス関数はスレッドセーフです。 任意のスレッド T がインターフェイス関数の 1 つで実行中である限り、インターフェイス関数の 1 つを呼び出す同じプロセスの別のスレッドは、T がインターフェイス関数を解放するまでブロックされます。 実際には、プロセスで並行して実行されるスレッドは、インターフェイス関数の使用に関する限り、シリアライズされます。 異なるプロセスのスレッド間でインターフェイスアクセスをシリアライズする必要はありません。これは、NNI を使用する各プロセスごとに独自の Natural セッションが実行されるためです。

呼び出し元のアプリケーションは、NNIへのマルチスレッドアクセスも明示的に制御できます。このことは、スレッドが別のスレッドに中断されずに一連の NNI コールを実行する場合に意味があります。 この処理を実現するため、スレッドは `nni_enter` を呼び出します。この関数を使用すると、スレッドは他のすべてのスレッドが NNI を解放するまで待機できます。 その後スレッドは作業を開始し、NNI 関数を自由に呼び出します。 作業が終了すると、スレッドは `nni_leave` を呼び出して、他のスレッドが NNI にアクセスできるようにします。

NNI を使用するマルチスレッドアプリケーションは、次のルールに従う必要があります。

- 関数 `nni_initialize` および `nni_uninitialize` は、プロセスごとに少なくとも 1 回呼び出す必要があります。
- 関数 `nni_uninitialize` は、`nni_initialize` への対応するコールと同じスレッドで呼び出す必要があります。
- 関数`nni_uninitialize`は、NNIを使用する最後のスレッドが終了する前に呼び出さないでください。

133 NaturalX

ここでは、オブジェクトベースのアプリケーションの開発および配布方法について説明します。

次のトピックについて説明します。

134 NaturalX について

このchapterでは、NaturalX インターフェイスおよび専用の Natural ステートメントセットを使用するコンポーネントベースのプログラミングについて簡単に紹介します。

次のトピックについて説明します。

なぜ NaturalX か

コンポーネントアーキテクチャに基づくソフトウェアアプリケーションには、従来の設計よりも多くの利点があります。 例えば、次のような利点があります。

- 迅速な開発。 プログラマは、事前に構築されたコンポーネントからソフトウェアをアセンブルすることによって、アプリケーションをより速く構築できます。
- 開発費の削減。 プログラムのインターフェイスの共通のセットを用意することによって、コンポーネントを完全なソリューションに統合するための労力が少なくなります。
- 柔軟性の向上。 アプリケーションを構成しているコンポーネントの一部を交換するだけで、社内の各部門に合わせたソフトウェアのカスタマイズが容易になります。
- メンテナンスコストの削減。 アップグレードの際は、多くの場合、アプリケーション全体を修正するのではなく、コンポーネントの一部を交換するだけで済みます。
- 容易な配布。 各コンポーネントには、データ構造と機能が配布可能なユニットにカプセル化されています。

NaturalX を使用すると、コンポーネントベースのアプリケーションを作成できます。

NaturalX は DCOM と連携して使用できます。 これにより、次のことが可能になります。

- 使用するコンポーネントに他のコンポーネントからアクセスします。
- ローカルまたはリモートのサーバーでこれらのコンポーネントを実行します。
- Natural プログラム内からプロセスおよびマシン境界を経由して、さまざまなプログラミング言語で書かれたコンポーネントにアクセスします。
- 既存の Natural アプリケーションに（準）標準化インターフェイスを提供します。

企業がこれらの利点をどのように利用できるかをシナリオで説明すると、次のようになります。ある企業が、コンポーネントを使用するアプリケーション設計に基づく新しい販売管理システムを導入します。 アプリケーションには多数のデータエントリコンポーネントがあり、各セールスポイントに1つずつ割り当てられています。ただし、これらすべてのセールスポイントでは、サーバー上で実行する共通の税金計算コンポーネントが使用されています。 税法が変更された場合は、各サイトでデータエントリコンポーネントを変更するのではなく、税金コンポーネントを更新するだけで済みます。 また、ネットワークプログラミングや、異なる言語で記述されているコンポーネントの統合について悩む必要がないため、プログラマの作業は容易になります。

プログラミング手法

このセクションでは、次のトピックについて説明します。

オブジェクトベースのプログラミング

NaturalX はオブジェクトベースのプログラミングのアプローチに従っています。 このアプローチの特徴は、対応する機能を持つデータ構造を複数のクラスにカプセル化することです。 カプセル化は、配布を容易にするための基礎となります。 オブジェクトモデルに基づくソフトウェアコンポーネントの相互運用に関する（準）標準があることから、オブジェクトベースのアプローチは、プログラム、マシン、およびプログラミング言語の境界を越えてソフトウェアコンポーネントを相互運用可能にするための優れた方法でもあります。

クラスの定義

オブジェクトベースのアプリケーションでは、各機能はオブジェクトから提供されるサービスであるとみなされます。 各オブジェクトは特定のクラスに属します。 クライアントは、サービスを使用して、ビジネスタスクを実行したり、さらに複雑なサービスを構築して他のクライアントに提供したりすることができます。 したがって、NaturalX でアプリケーションを作成する基本手順は、アプリケーションを形成するクラスを定義することです。 多くの場合、各クラスは例えば、銀行口座、航空機、出荷など、当該アプリケーションが処理する実際の物事に単純に一致します。オブジェクト指向の設計については広範な優れた文献があります。また、十分に実証された多くのメソッドを使用して、特定のビジネスのクラスを識別することができます。

クラスを定義するプロセスは、大まかに次の手順に分けることができます。

- クラスタイプの Natural モジュールを作成します。
- `DEFINE CLASS` ステートメントを使用して、クラスの名前を指定します。 この名前は、このクラスのオブジェクトを作成するためにクライアントが使用します。
- `DEFINE DATA` ステートメントの `OBJECT` 節を使用して、クラスのオブジェクトが内部的にどのように表示されるかを定義します。 データエリアエディタを使用して、オブジェクトのレイアウトを記述するローカルデータエリアを作成し、このデータエリアを `OBJECT` 節に割り当てます。

これらの手順の詳細は、「*オブジェクトベースの **Natural** アプリケーションの開発*」に記載されています。

インターフェイスの定義

クラスは、クライアントに役立つサービスを提供する必要があり、このサービスはインターフェイスを使用して行われます。インターフェイスは、メソッドとプロパティのコレクションです。メソッドは、クライアントによって要求されたときにクラスのオブジェクトが実行できる機能です。 プロパティは、クライアントが検索したり変更したりできるオブジェクトの属性です。 クライアントは、クラスのオブジェクトを作成し、そのインターフェイスのメソッドとプロパティを使用することによってサービスにアクセスします。

インターフェイスを定義するプロセスは、大まかに次の手順に分けることができます。

- INTERFACE 節を使用して、インターフェイス名を指定します。
- インターフェイスのプロパティを PROPERTY 定義で定義します。
- インターフェイスのメソッドを METHOD 定義で定義します。

これらの手順の詳細は、「*オブジェクトベースの Natural アプリケーションの開発*」に記載されています。

単純なクラスにはインターフェイスが 1 つしかありませんが、1 つのクラスに複数のインターフェイスがある場合もあります。 この機能によって、クラスの同じ機能面に属しているメソッドとプロパティを 1 つのインターフェイスにまとめ、別の機能面を処理するために異なるインターフェイスを定義できます。 例えば、Employee（従業員）クラスには、従業員の管理面に関するすべてのメソッドとプロパティが含まれる Administration（管理）インターフェイスを定義することができます。 このインターフェイスには、Salary（給与）、Department（部門）の各プロパティと、TransferToDepartment メソッドを含めることができます。 もう 1 つのインターフェイスである Qualifications（資格）には、従業員の資格に関する面を含めることができます。

インターフェイスの継承

1つのクラスに複数のインターフェイスを定義することは、クラスとインターフェイスのより高度な設計方法であるインターフェイス継承を使用するための最初の手順です。 これにより、異なるクラスで同じインターフェイス定義を再利用できるようになります。Manager（経営者）クラスがあると仮定してください。このクラスは、資格については Employee クラスと同様に処理されますが、管理に関しては違う方法で処理される必要があります。 このことは、両方のクラス内に Qualification インターフェイスを定義することによって実現できます。 このような処理には、特定のオブジェクト上で Qualification インターフェイスを使用するクライアントが、オブジェクトが Employee と Manager のどちらを表しているのかを明示的にチェックする必要がないという利点があります。オブジェクトのクラスを知らなくても、同じメソッドとプロパティを単純に使用することができます。 また、プロパティまたはメソッドが同じインターフェイス定義で提示される場合は、2 つのクラス内で異なる方法で実装することもできます。

インターフェイス継承を使用するプロセスは、大まかに次の手順に分けることができます。

- INTERFACE ステートメントを使用して、1 つ以上のインターフェイスをクラスに直接定義するのではなく、コピーコードに定義します。

- INTERFACE ステートメントの METHOD と PROPERTY の定義に IS 節を含める必要はありません。この時点では、メソッドとプロパティに実装を割り当てずに、インターフェイスの外部的な外観だけを定義します。
- INTERFACE 節を使用して、インターフェイスを定義したコピーコードを、インターフェイスを実装する各クラスにインクルードします。
- METHOD ステートメントと PROPERTY ステートメントを使用して、インターフェイスを実装する各クラス内のインターフェイスのメソッドとプロパティに実装を割り当てます。

135 NaturalX アプリケーションの開発

このchapterでは、クラスを定義および使用してアプリケーションを開発する方法について説明します。

次のトピックについて説明します。

開発環境

- **Windows** プラットフォームでのクラスの開発
  Windows プラットフォームでは、Natural クラスを開発するためのツールとして、Natural からクラスビルダが提供されます。 クラスビルダは、構造化された階層順に Natural クラスを表示し、ユーザーがクラスとそのコンポーネントを効率的に管理できるようにします。 クラスビルダを使用すると、この後で説明する構文要素に関する知識はまったく必要ないか、または基礎知識だけで済みます。
- **SPoD** を使用したクラスの作成
  メインフレーム、UNIX、または OpenVMS の一部またはすべてのリモート開発サーバーが含まれる Natural Single Point of Development（SPoD）環境では、Natural スタジオのフロントエンドで使用可能なクラスビルダを使用して、メインフレーム、UNIX、または OpenVMS プラットフォーム上でクラスを開発できます。 この場合、この後で説明する構文要素に関する知識はまったく必要ないか、または基礎知識だけで済みます。
- メインフレーム、**UNIX**、または **OpenVMS** プラットフォームでのクラスの開発
  SPoD を使用しない場合は、Natural プログラムエディタを使ってこれらのプラットフォームのクラスを開発することができます。 この場合は、この後で説明するクラス定義の構文の知識が必要です。

クラスの定義

クラスを定義するときは、Natural クラスモジュールを作成する必要があります。この中で `DEFINE CLASS` ステートメントを作成します。 `DEFINE CLASS` ステートメントを使用して、外部的に使用可能な名前をクラスに割り当て、そのインターフェイス、メソッドおよびプロパティを定義します。 クラスのインスタンスのレイアウトを記述するオブジェクトデータエリアをクラスに割り当てることもできます。 `DEFINE CLASS` ステートメントは、COM クラスとして登録されるクラスにグローバルユニーク ID を割り当てるためにも使用されます。

このセクションでは、次のトピックについて説明します。

- Natural クラスモジュールの作成
- クラスの指定
- インターフェイスの定義
- オブジェクトデータ変数のプロパティへの割り当て
- サブプログラムのメソッドへの割り当て

- メソッドの実装

Natural クラスモジュールの作成

手順 135.1. Natural クラスモジュールを作成するには

- CREATE OBJECT ステートメントを使用して、クラスタイプの Natural オブジェクトを作成します。

クラスの指定

DEFINE CLASS ステートメントは、クラスの名前、クラスがサポートするインターフェイス、およびそのオブジェクトの構造を定義します。COM クラスとして登録されるクラスについては、クラスのグローバルユニーク ID およびそのアクティベーションポリシーも指定されます。

手順 135.2. クラスを指定するには

- DEFINE CLASS ステートメントを使用します。詳細については『ステートメント』ドキュメントを参照してください。

インターフェイスの定義

クラスの各インターフェイスは、クラス定義内部の INTERFACE ステートメントで指定されます。INTERFACE ステートメントは、インターフェイスの名前と多数のプロパティおよびメソッドを指定します。COM クラスとして登録されるクラスについては、インターフェイスのグローバルユニーク ID も指定されます。

クラスには 1 つ以上のインターフェイスを定義できます。 インターフェイスごとに 1 つの INTERFACE ステートメントがクラス定義に指定されます。 各 INTERFACE ステートメントには 1 つ以上の PROPERTY 節および METHOD 節が含まれます。 通常、1 つのインターフェイスに含まれるプロパティとメソッドは、技術またはビジネスのいずれかの観点から関連付けられます。

PROPERTY 節はプロパティの名前を定義し、オブジェクトデータエリアからプロパティに変数を割り当てます。 この変数を使用して、プロパティの値を保存します。

METHOD 節はメソッドの名前を定義し、メソッドにサブプログラムを割り当てます。 このサブプログラムを使用して、メソッドを実装します。

手順 135.3. インターフェイスを定義するには

- INTERFACE ステートメントを使用します。詳細については『ステートメント』ドキュメントを参照してください。

オブジェクトデータ変数のプロパティへの割り当て

PROPERTY ステートメントは、複数のクラスが同じインターフェイスを異なる方法で実装するときにのみ使用されます。 この場合、クラスは同じインターフェイス定義を共有し、Natural コピーコードからインクルードします。 次に PROPERTY ステートメントを使用して、インターフェイス定義の外でオブジェクトデータエリアからプロパティに変数を割り当てます。 INTERFACE ステートメントの PROPERTY 節と同様に、PROPERTY ステートメントはプロパティの名前を定義し、オブジェクトデータエリアからプロパティに変数を割り当てます。 この変数を使用して、プロパティの値を保存します。

手順 135.4. オブジェクトデータ変数をプロパティに割り当てるには

- PROPERTY ステートメントを使用します。詳細については『ステートメント』ドキュメントを参照してください。

サブプログラムのメソッドへの割り当て

METHOD ステートメントは、複数のクラスが同じインターフェイスを異なる方法で実装するときにのみ使用されます。 この場合、クラスは同じインターフェイス定義を共有し、Natural コピーコードからインクルードします。 METHOD ステートメントは、インターフェイス定義の外でサブプログラムをメソッドに割り当てるために使用されます。 INTERFACE ステートメントの METHOD 節と同様に、METHOD ステートメントはメソッドの名前を定義し、サブプログラムをメソッドに割り当てます。 このサブプログラムを使用して、メソッドを実装します。

手順 135.5. サブプログラムをメソッドに割り当てるには

- METHOD ステートメントを使用します。詳細については『ステートメント』ドキュメントを参照してください。

メソッドの実装

メソッドは次のような一般形式の Natural サブプログラムとして実装されます。

```
DEFINE DATA statement
*
* Implementation code of the method
*
END
```

DEFINE DATA ステートメントの詳細については、『ステートメント』ドキュメントを参照してください。

DEFINE DATA ステートメントのすべての節はオプションです。

データの整合性を確保するために、インラインデータ定義ではなくデータエリアを使うことをお勧めします。

PARAMETER 節を指定すると、メソッドにパラメータや戻り値を指定することができます。

パラメータデータエリアの BY VALUE とマークされたパラメータは、メソッドの入力パラメータです。

BY VALUE とマークされていないパラメータは、「BY REFERENCE」によって渡される入力／出力パラメータです。 これがデフォルトです。

BY VALUE RESULT とマークされている最初のパラメータが、メソッドの戻り値として返されます。 複数のパラメータがこのようにマークされている場合は、その他のパラメータは入力／出力パラメータとして扱われます。

OPTIONAL としてマークされているパラメータは、メソッドが呼び出される場合は指定する必要はありません。 SEND METHOD ステートメントで *n*X 表記を使用することにより、これらを未指定にしておくことができます。

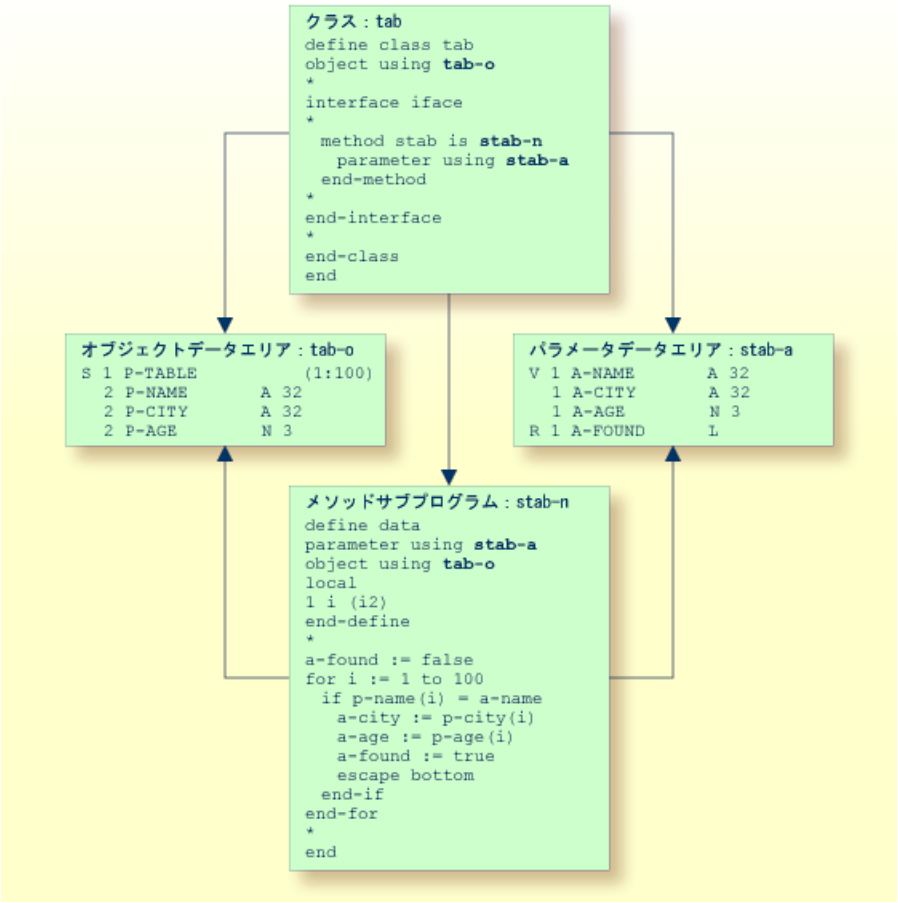
メソッドサブプログラムが、クラス定義内の対応する METHOD ステートメントに指定されたものとまったく同じパラメータを受け入れるようにするには、インラインデータ定義ではなくパラメータデータエリアを使用します。 対応する METHOD ステートメント内のものと同じパラメータデータエリアを使用します。

メソッドサブプログラムでオブジェクトデータ構造にアクセスするために、OBJECT 節を指定できます。 メソッドサブプログラムがオブジェクトデータに正しくアクセスできるようにするには、インラインデータ定義ではなくローカルデータエリアを使用します。 DEFINE CLASS ステートメントの OBJECT 節に指定されているものと同じローカルデータエリアを使用してください。

GLOBAL 節、LOCAL 節、および INDEPENDENT 節は、他の任意の Natural プログラムで使用できます。

技術的には可能ですが、通常はメソッドサブプログラムで CONTEXT 節を使用することは意味がありません。

次の例は、特定の人物に関するデータをテーブルから検索します。 検索キーは BY VALUE パラメータとして渡されます。 結果のデータは、「BY REFERENCE」 パラメータで返されます。 「BY REFERENCE」 はデフォルト定義です。 メソッドの戻り値は BY VALUE RESULT 指定によって定義されます。



クラスとオブジェクトの使用

ローカルの Natural セッションで作成されたオブジェクトは、同じ Natural セッションの他のモジュールからアクセスできます。

他のプロセスまたはリモートマシンで作成されたオブジェクトは、DCOM 経由でアクセスできます。

いずれの場合も、クラスとそのオブジェクトへのアクセスおよび使用に関する規則は同じです。

`CREATE OBJECT` ステートメントは、特定のクラスのオブジェクト（インスタンスとも呼ばれます）を作成するために使用します。

Natural プログラムのオブジェクトを参照するには、オブジェクトハンドルを `DEFINE DATA` ステートメントで定義する必要があります。 オブジェクトのメソッドは `SEND METHOD` ステートメントで呼び出されます。 オブジェクトにはプロパティを指定できます。プロパティには通常の割り当て構文を使用してアクセスできます。

Note: DCOM 経由で NaturalX クラスを使用するには、最初にクラスを登録する必要があります。

次の手順について説明します。

- オブジェクトハンドルの定義
- クラスのインスタンスの作成
- オブジェクトの特定メソッドの呼び出し
- プロパティへのアクセス
- サンプルアプリケーション

オブジェクトハンドルの定義

Natural プログラムのオブジェクトを参照するには、オブジェクトハンドルを DEFINE DATA ステートメントで次のように定義する必要があります。

```
DEFINE DATA
  level-handle-name [(array-definition)] HANDLE OF OBJECT
  ...
END-DEFINE
```

例：

```
DEFINE DATA LOCAL
1 #MYOBJ1 HANDLE OF OBJECT
1 #MYOBJ2 (1:5) HANDLE OF OBJECT
END-DEFINE
```

クラスのインスタンスの作成

手順 135.6. クラスのインスタンスを作成するには

- CREATE OBJECT ステートメントを使用します。詳細については『ステートメント』ドキュメントを参照してください。

オブジェクトの特定メソッドの呼び出し

手順 135.7. オブジェクトの特定メソッドを呼び出すには

- SEND METHOD ステートメントを使用します。詳細については『ステートメント』ドキュメントを参照してください。

プロパティへのアクセス

プロパティには、次のように ASSIGN（または COMPUTE）ステートメントを使用してアクセスできます。

```
ASSIGN operand1.property-name = operand2
ASSIGN operand2 = operand1.property-name
```

オブジェクトハンドル - *operand1*

operand1 はオブジェクトハンドルとして定義する必要があり、プロパティがアクセスされるオブジェクトを識別します。 オブジェクトがすでに存在している必要があります。

operand2

operand2 として、プロパティのフォーマットに対してデータ転送互換を必要とするフォーマットのオペランドを指定します。 詳細については、データ転送の互換性規則の説明を参照してください。

オブジェクトがDCOM経由でアクセスされる場合、データタイプの変換規則も考慮する必要があります。この規則の概要については『*オペレーション*』ドキュメントの「*タイプ情報の使用*」を参照してください。

property-name

オブジェクトのプロパティの名前です。

プロパティ名が Natural 識別子構文に対応する場合は、次のように指定できます。

```
create object #o1 of class "Employee"
  #age := #o1.Age
```

プロパティ名がNatural識別子構文に対応しない場合は、次のように山カッコで囲む必要があります。

```
create object #o1 of class "Employee"
  #salary := #o1.<<%Salary>>
```

プロパティ名は、インターフェイス名で修飾することもできます。 同じ名前のプロパティが含まれる複数のインターフェイスがオブジェクトにある場合は、修飾が必要になります。 この場合、修飾したプロパティ名は山カッコで囲む必要があります。

```
create object #o1 of class "Employee"
  #age := #o1.<<PersonalData.Age>>
```

例：

```
define data
  local
  1 #i         (i2)
  1 #o         handle of object
  1 #p         (5) handle of object
  1 #q         (5) handle of object
  1 #salary    (p7.2)
  1 #history   (p7.2/1:10)
  end-define
  * ...
  * Code omitted for brevity.
  * ...
  * Set/Read the Salary property of the object #o.
  #o.Salary := #salary
  #salary := #o.Salary
  * Set/Read the Salary property of
  * the second object of the array #p.
  #p.Salary(2) := #salary
  #salary := #p.Salary(2)
  *
  * Set/Read the SalaryHistory property of the object #o.
  #o.SalaryHistory := #history(1:10)
  #history(1:10) := #o.SalaryHistory
  * Set/Read the SalaryHistory property of
  * the second object of the array #p.
  #p.SalaryHistory(2) := #history(1:10)
  #history(1:10) := #p.SalaryHistory(2)
```

```
*
* Set the Salary property of each object in #p to the same value.
#p.Salary(*) := #salary
* Set the SalaryHistory property of each object in #p
* to the same value.
#p.SalaryHistory(*) := #history(1:10)
*
* Set the Salary property of each object in #p to the value
* of the Salary property of the corresponding object in #q.
#p.Salary(*) := #q.Salary(*)
* Set the SalaryHistory property of each object in #p to the value
* of the SalaryHistory property of the corresponding object in #q.
#p.SalaryHistory(*) := #q.SalaryHistory(*)
*
end
```

配列が値として正しく含まれるオブジェクトハンドルとプロパティの配列を使用するためには、次の知識が重要です。

オブジェクトハンドルの配列のオカレンスのプロパティは、次のインデックス表記で指定します。

```
#p.Salary(2) := #salary
```

配列が値として含まれるプロパティは、常に全体としてアクセスされます。 したがって、プロパティ名でのインデックス表記は不要です。

```
#o.SalaryHistory := #history(1:10)
```

結果的に、配列が値として含まれるオブジェクトハンドルの配列のオカレンスのプロパティは、次のように指定します。

```
#p.SalaryHistory(2) := #history(1:10)
```

サンプルアプリケーション

アプリケーション例がライブラリ SYSEXCOM および SYSEXCOC で提供されています。 サンプルを実行する方法については、これらのライブラリの A-README メンバを参照してください。

136 NaturalX アプリケーションの配布

NaturalXクラスで構成されるアプリケーションは、DCOMを使用して複数のプロセスおよびマシンに配布することができます。

このchapterでは、次のトピックについて説明します。

『オペレーション』ドキュメントの「*NaturalX サーバー環境でのステートメントおよびコマンドの使用*」も参照してください。

概要

NaturalX を使用すると、Natural クラスとそのサービスをローカルおよびリモートのクライアントで使用可能にすることによって、分散アプリケーションを作成できます。 ローカルクライアントとは、特定の NaturalX サーバーと同じマシンで実行されるプロセスであり、リモートクライアントとは異なるマシンで実行されるプロセスです。

アプリケーションの配布には、広く使われている分散オブジェクトテクノロジである Microsoft 分散コンポーネントオブジェクトモデル（DCOM）を使用します。 Natural クラスを DCOM に登録すると、そのインターフェイスが準標準化された形でダイナミック COM インターフェイスとしてクライアントに提供されます。ダイナミック COM インターフェイスはディスパッチインターフェイスとも呼ばれます。 これらのインターフェイスは、Visual Basic、Java、C++、および Natural などの多くのプログラミング言語によって簡単に指定できます。

NaturalX アプリケーションの配布を編成するときには、いくつかの点を考慮する必要があります。 それぞれの点の詳細については、このセクションおよび『オペレーション』ドキュメントで説明します。

- 各クラスを内部、外部、またはローカルのいずれにするかを決定します。「*内部、外部、およびローカルクラス*」を参照してください。
- グローバルユニーク ID（GUID）を内部クラスと外部クラス、およびそれらのインターフェイスに割り当てて、ネットワーク内で一意に指定できるようにする必要があります。「*グローバルユニーク ID - GUID*」を参照してください。
- 各クラスにアクティベーションポリシーを定義して、どの DCOM の下でアクティブにするかの条件を制御することができます。『オペレーション』ドキュメントの「アクティベーションポリシー」を参照してください。
- クラスをアプリケーションに編成するために、NaturalX サーバーを定義し、クラスを各サーバーに割り当てることができます。『オペレーション』ドキュメントの「*NaturalX サーバー*」を参照してください。
- DCOM に認識させるためにクラスを登録する必要があります。『オペレーション』ドキュメントの「登録」を参照してください。
- アプリケーションの動作をさらに制御するために、アプリケーションを設定できます。『オペレーション』ドキュメントの「*コンフィグレーション概要*」および「*Windows 上での DCOM コンフィグレーション*」を参照してください。

内部、外部、およびローカルクラス

内部使用、外部使用、およびローカル使用専用にクラスを区別することは重要です。

内部クラス

内部クラスのオブジェクト（インスタンス）はクライアントプロセスでのみ作成できます。

内部クラスには次の特徴があります。

- ファイルやシステム変数などクライアントセッション依存のリソースにアクセスします。
- クライアントトランザクション内で実行できます。
- Natural デバッガを使用してデバッグできます（ローカルデバッグ）。

外部クラス

外部クラスのオブジェクト（インスタンス）は、異なるプロセスまたは異なるマシンで作成できます。 クライアントプロセスがこのクラスのサーバーでもある場合は、このクライアントプロセスでも作成できます。

外部クラスには次の特徴があります。

- クライアントセッション依存のリソース（スタック、ファイル、システム変数など）にアクセスしません。
- クライアントトランザクション内で実行できません。
- リモートノードから使用できます。
- Natural、Java、Visual Basic、C/C++ などのさまざまな言語を使用して、各種クライアントから使用できます。
- Natural デバッガを使用してデバッグできます（ローカルデバッグ）。

ローカルクラス

ローカルクラスは、ローカルの実行モードで実行されるクラスです。 クラスが登録されていない場合、またはDCOMを使用できない場合、Naturalはクラスをローカル、つまりNaturalセッション内で実行します。

ローカルクラスには次の特徴があります。

- DCOM を使用できなくても使用できます。
- DCOM に登録する必要はありません。
- クライアントプロセスの外部からは使用できません。

グローバルユニーク ID - GUID

DCOM ではグローバルユニーク ID（GUID）を使用して、あらゆるインターフェイスとクラスを識別します。GUID は 128 ビットの整数で、世界中で一意であることが事実上保証されています。 したがって、サーバーコンポーネントを確実に特定し、クライアントが誤ってオブジェクトに接続することを防止するのに役立ちます。

クラスを DCOM に登録する場合は、Natural クラス内に定義されているすべてのインターフェイスおよび登録するクラス自体を、このグローバルユニーク ID に関連付ける必要があります。

いったんグローバルユニーク ID をインターフェイスまたはクラスに割り当てた後は、この ID を変更することはできません。

クラスビルダの使用

Natural クラスを開発するためのツールとして、Natural からクラスビルダが提供されます。 クラスビルダは、すべてのクラスおよびインターフェイスに GUID を自動的に割り当てます。

137 ActiveX コンポーネント SoftwareAG.NaturalX.Utilities

このchapterでは、次のトピックについて説明します。

目的

ActiveX コンポーネントの SoftwareAG.NaturalX.Utilities は、NaturalX および Natural スタジオプラグインのコンテキストで役立つ数多くの方法を提供します。

例えば、Naturalアプリケーションでのこのコンポーネントの一般的な使用方法は、次のようになります。

```
define data
local
1 #util handle of object
1 #studio handle of object
end-define
*
* First create an instance of the class SoftwareAG.NaturalX.Utilities.
create object #util of 'SoftwareAG.NaturalX.Utilities.4'
if #util eq null-handle
  escape routine
end-if
*
* Now call the individual methods of the component, for instance
* to get access to the Natural Studio Automation Interface.
*
send 'GetThisNaturalStudio' to #util return #studio
if #studio eq null-handle
  escape routine
end-if
*
end
```

この例が正常に実行されるのは、Natural スタジオセッションで実行された場合のみです。Natural ランタイムセッションや Natural デバッガの下で実行された場合は、GetThisNaturalStudio に対する呼び出しで NULL-HANDLE が返されます。 これは、Natural スタジオセッションのみに INatAutoStudio インターフェイスがあるためです。 プログラムが Natural デバッガの下で実行中である場合は、Natural スタジオ外部の Natural ランタイムセッションで効率的に実行されるため、GetThisNaturalStudio に対する呼び出しではやはり NULL-HANDLE が返されます。

この例を Natural デバッガの下でも実行されるようにするには、次のように修正して、Natural スタジオセッションの INatAutoStudio インターフェイスを取得する必要があります。

```
define data
local
1 #util handle of object
1 #studio handle of object
1 #rot handle of object
1 #ro (a) dynamic
end-define
*
* First create an instance of the class SoftwareAG.NaturalX.Utilities.
create object #util of 'SoftwareAG.NaturalX.Utilities.4'
if #util eq null-handle
  escape routine
end-if
*
* Now call the individual methods of the component, for instance
* to get access to the Natural Studio Automation Interface.
*
send 'GetThisNaturalStudio' to #util return #studio
if #studio eq null-handle
* We might be in a debugging session.
* Try to locate the Natural Studio session
* from which the debugger has been started.
* Retrieve the running objects table.
  send 'GetRunningObjects' to #util return #rot
  if #rot eq null-handle
    escape routine
  end-if
* Iterate across the running objects table.
  repeat
    send 'Next' to #rot return #ro
    if #ro eq ' '
      escape bottom
    end-if
*   If we hit a running Natural Studio session, we access it.
    if substring(#ro,1,13) eq 'NaturalStudio'
      send 'BindToObject' to #util
      with #ro (ad=o) return #studio
      escape bottom
    end-if
  end-repeat
end-if
```

```
*
end
```

インターフェイス

個々のインターフェイス、そのメソッドおよび使用方法については、別のドキュメントで詳しく説明しています。

このコンポーネントでは、次のインターフェイスが提供されます。

- インターフェイス INaturalXUtilities
- インターフェイス IRunningObjects

138 インターフェイス INaturalXUtilities

このchapterでは、次のトピックについて説明します。

目的

コンポーネント `SoftwareAG.NaturalX.Utilities` の主要インターフェイスです。 これは、コンポーネントの新しいインスタンスが作成されるときに返されます。

例：

```
define data
local
1 #util handle of object
end-define
*
* Create an instance of the class SoftwareAG.NaturalX.Utilities.
create object #util of 'SoftwareAG.NaturalX.Utilities.4'
if #util eq null-handle
  escape routine
end-if
*
end
```

`CREATE OBJECT` ステートメントが正常に実行されると、変数 `#util` にタイプ `INaturalXUtilities` のインターフェイスが入ります。

メソッド

次のメソッドを使用できます。

- GetThisNaturalStudio
- GetRunningObjects

- BindToObject

GetThisNaturalStudio

現在の Natural スタジオセッションのルートインターフェイス `INatAutoStudio` を取得します。クライアントはこのインターフェイスを取得すると、Natural スタジオオートメーションインターフェイスで提供されたときと同様に、Natural スタジオの機能にアクセスできます。

パラメータ

名前	Natural タイプ	変形タイプ	注釈
戻り値	HANDLE OF OBJECT	VT_DISPATCH（INatAutoStudio）	

戻り値
現在の Natural スタジオセッションのルートインターフェイス `INatAutoStudio`。このメソッドが Natural スタジオセッションではない Natural セッション内で呼び出された場合は `NULL-HANDLE`。

GetRunningObjects

実行オブジェクトテーブル（ROT）に含まれるオブジェクトの名前の中で反復させるために使用するインターフェイス `IRunningObjects` が返されます。

パラメータ

名前	Natural タイプ	変形タイプ	注釈
戻り値	HANDLE OF OBJECT	VT_DISPATCH（**IRunningObjects**）	

戻り値
実行オブジェクトテーブル（ROT）に含まれるオブジェクトの名前の中で反復させるために使用するインターフェイス `IRunningObjects`。

BindToObject

「モニカ」（Windows 用語）と呼ばれる特定の種類の名前で識別されるオブジェクトへのインターフェイスが返されます。モニカの詳細については、「*インターフェイス **IRunningObjects***」を参照してください。

パラメータ

名前	Natural タイプ	変形タイプ	注釈
戻り値	HANDLE OF OBJECT	VT_DISPATCH	
名前	A	VT_BSTR	値

戻り値
名前で指定された名前で識別されるオブジェクトへのインターフェイス。

名前
特定のオブジェクトを名前で識別するために使用します。この名前は、次のいずれかのカテゴリに属する必要があります。

- ファイルモニカ。例えば *c:\MyDoc.doc* など。
- URL モニカ。例えば *http://www.myorg.org/MyDoc.doc* または *ftp://ftp.myorg.org/MyDoc.doc* など。
- ROT に含まれるオブジェクトの名前。 ROT に含まれるオブジェクトの名前は、インターフェイス `IRunningObjects` を使用して取得できます。

ファイルモニカまたは URL モニカを指定した場合は、対応するファイル拡張子に登録されているアプリケーションに、対応するオブジェクトがロードされ、そのオブジェクトへのインターフェイスポインタ（オブジェクトハンドル）が返されます。このオブジェクトがすでにアプリケーションにロードされている場合は、すでに実行中のインスタンスへのインターフェイスポインタ（オブジェクトハンドル）が返されます。

例：

```
define data
local
1 #util handle of object
1 #obj handle of object
1 #content handle of object
1 #word handle of object
1 #doc (a) dynamic
1 #text (a) dynamic
end-define
*
* Create an instance of the utilities class.
create object #util of 'SoftwareAG.NaturalX.Utilities.4'
if #util eq null-handle
  escape routine
end-if
*
* Load a document into Microsoft Word.
* The option (ad=o) is essential, because the
* method expects a by value parameter.
#doc := 'c:\word.doc'
```

```
send 'BindToObject' to #util with #doc (ad=o) return #obj
if #obj eq null-handle
  escape routine
end-if
*
* Access the content of the document.
#content := #obj.Content
#text := #content.Text
write 'Content:' #text (al=60)
*
* Close Microsoft Word.
#word := #obj.Application
send 'Quit' to #word
*
end
```

139 インターフェイス IRunningObjects

このchapterでは、次のトピックについて説明します。

目的

このインターフェイスは、実行オブジェクトテーブル（ROT）に含まれるオブジェクトの名前の中で使用する反復子です。 ROT は Windows によって提供され、管理されるシステムテーブルです。 ROT を使用すると、ユーザーが現在アプリケーションで作業中のオブジェクトやドキュメントを、他のアプリケーションで使用可能にすることができます。

このテーブルに含まれている各オブジェクトは、「モニカ」 という名前で識別されます。 モニカとは、特定の構文に従う名前です。 例えば、ファイルモニカは、ファイルシステム内でファイルを識別します。 読み取り可能な形のファイルモニカとは、*c:\MyDoc.doc* のように、完全パス名を使用した単なるファイル名です。 別の例として、インターネット内のリソースおよびそれにアクセスするために使用するプロトコルを識別する URL モニカがあります。 読み取り可能な形の URL モニカとは、単なる通常の URL です。例えば *http://www.myorg.org/MyDoc.doc* のようになります。

ROT 内のオブジェクトへのアクセスを試みるアプリケーションは、オブジェクトを識別するモニカを指定し、そのオブジェクトへのインターフェイスポインタ（オブジェクトハンドラ）を受け取ります。

ユーザーが現在 ROT で作業中のオブジェクトやドキュメントが、アプリケーションによって入力されることもよくあります。

例：

ドキュメント *c:\MyDoc.doc* を Microsoft Word で開くと、このドキュメントの名前である *c:\MyDoc.doc* とこのドキュメントへのインターフェイスポインタが Microsoft Word によって ROT に入力されます。

Natural スタジオセッションが開始すると、このセッションのオートメーションルートインターフェイスである `INatAutoStudio` が Natural スタジオによって ROT に入力されます。 インターフェイスは次のような構造の名前で識別されます。

```
NaturalStudio/<version>/<userid>/>processid>
```

例：

```
NaturalStudio/n.n/SCULLY/42
```

n.n は製品バージョンです。

他のアプリケーションはこの名前を指定することによって ROT 内のオートメーションルートインターフェイスを取得し、これを使用して Natural スタジオセッションにアクセスできます。

インターフェイス IRunningObjects を使用すると、ROT に現在含まれているすべてのオブジェクトの名前の中で反復することができます。 見つかった名前は、その後メソッド INaturalXUtilities::BindToObject で使用して、対応するオブジェクトへのインターフェイスポインタ（オブジェクトハンドル）を取得できます。

例：

次のサンプルプログラムでは、文字 *n.n* が Natural 製品バージョンを表します。 このサンプルプログラムを実行する場合は、この文字を現在の Natural 製品バージョンで置き換えてください。

```
define data
local
1 #util handle of object
1 #studio handle of object
1 #objects handle of object
1 #progs handle of object
1 #prog handle of object
1 #rot handle of object
1 #ro (a) dynamic
end-define
*
* Create an instance of the Utilities class.
create object #util of 'SoftwareAG.NaturalX.Utilities.4'
if #util eq null-handle
  escape routine
end-if
*
* Retrieve the running objects table.
send 'GetRunningObjects' to #util return #rot
if #rot eq null-handle
  escape routine
end-if
*
* Iterate across the running objects table.
repeat
  send 'Next' to #rot return #ro
  if #ro eq ' '
    escape bottom
  end-if
* If we hit a running Natural Studio session,...
  if substring(#ro,1,17) eq 'NaturalStudio/n.n'
*   ...we access it,...
    send 'BindToObject' to #util
    with #ro (ad=o) return #studio
*   ...open a Program Editor in that session...
    #objects := #studio.objects
    #progs := #objects.programs
    send 'Add' to #progs
```

```
    with 1009 return #prog
*   ...and display the identifier of this session in the editor.
    compress 'This is' #ro to #ro
    #prog.source := #ro
  end-if
end-repeat
*
end
```

メソッド

次のメソッドを使用できます。

- Next
- Reset

Next

ROT 内の次のオブジェクトの名前を返します。 この名前は、その後メソッド `INaturalXUtilities::BindToObject` で使用して、対応するオブジェクトへのインターフェイスポインタ（オブジェクトハンドル）を取得できます。

パラメータ

名前	Natural タイプ	変形タイプ	注釈
戻り値	A	VT_BSTR	

戻り値
ROT 内の次のオブジェクトの名前。

Reset

反復子を初期状態にリセットします。 `Reset` を呼び出した後は、その後の `Next` の呼び出しによって、ROT 内の最初のオブジェクトの名前が返されます。

140 ActiveX コンポーネント

SoftwareAG.NaturalX.Enumerator

このchapterでは、次のトピックについて説明します。

目的

ActiveXコンポーネントの SoftwareAG.NaturalX.Enumerator は、オートメーションオブジェクトのコレクション内を反復するために使用できる一般的な列挙子クラスを提供します。

例えば、Naturalアプリケーションでのこのコンポーネントの一般的な使用方法は、次のようになります。 完全に機能する例は、ライブラリ SYSEXPG のプログラム UTIL04 です。

```
define data
local
1 #enum handle of object
1 #files handle of object
1 #file handle of object
end-define
*
* First create an instance of the class SoftwareAG.NaturalX.Enumerator.
create object #enum of 'SoftwareAG.NaturalX.Enumerator.4'
if #enum eq null-handle
  escape routine
end-if
*
* Have a collection of Automation objects
* in the variable #files.
* Code omitted.
* ...
*
* Attach the collection to the enumerator.
send 'Attach' to #enum with #files (ad=o)
*
* Now iterate across the collection.
send 'Next' to #enum return #file
repeat while #file ne null-handle
* Process the item.
* Code omitted.
* ...
* Get the next item.
  send 'Next' to #enum return #file
end-repeat
```

```
*
end
```

インターフェイス

このコンポーネントのインターフェイス、そのメソッドおよび使用法については、別のドキュメントで詳しく説明しています。

このコンポーネントでは、次のインターフェイスが提供されます。

- インターフェイス IEnumerator

141 インターフェイス IEnumerator

このchapterでは、次のトピックについて説明します。

目的

コンポーネント SoftwareAG.NaturalX.Enumerator の主要インターフェイスです。 これは、コンポーネントの新しいインスタンスが作成されるときに返されます。

```
define data
local
1 #enum handle of object
end-define
*
* Create an instance of the class SoftwareAG.NaturalX.Enumerator.
create object #enum of 'SoftwareAG.NaturalX.Enumerator.4'
if #enum eq null-handle
  escape routine
end-if
*
end
```

CREATE OBJECT ステートメントが正常に実行されると、変数 #enum にタイプ IEnumerator のインターフェイスが入ります。

メソッド

次のメソッドを使用できます。

- Attach
- Reset
- Next

Attach

列挙子にコレクションをアタッチします。 すでにアタッチされていたコレクションは、自動的にデタッチされます。 コレクションをアタッチした後で、列挙子を使用して、このコレクションに含まれている項目を列挙できます。

パラメータ

名前	Natural タイプ	変形タイプ	注釈
コレクション	HANDLE OF OBJECT	VT_DISPATCH	値

コレクション
　オートメーションオブジェクトのコレクション。

Reset

列挙子を初期状態にリセットします。その後のメソッド `Next` への呼び出しでは、コレクションの最初の項目が返されます。

Next

コレクションの次の項目を返します。 次の項目がない場合は、`NULL-HANDLE` が返されます。 列挙をやり直す場合は、メソッド `Reset` を呼び出すことができます。

パラメータ

名前	Natural タイプ	変形タイプ	注釈
戻り値	HANDLE OF OBJECT	VT_DISPATCH	

戻り値
　コレクションの次の項目へのインターフェイス。

142 Natural 予約キーワード

このchapterでは、Naturalプログラミング言語で予約されているすべてのキーワードのリストを提供します。

Important: 命名の競合を避けるため、Natural 予約キーワードを変数の名前として使用しないことを強くお勧めします。

次のトピックについて説明します。

Natural 予約キーワードのアルファベット順リスト

次のリストは、Natural予約キーワードの概要および全般的な情報のみを示しています。疑問がある場合は、コンパイラの キーワードチェック機能を使用してください。

[A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z]

- A -

ABS
ABSOLUTE
ACCEPT
ACTION
ACTIVATION
AD
ADD
AFTER
AL
ALARM
ALL
ALPHA
ALPHABETICALLY
AND
ANY
APPL
APPLICATION
ARRAY
AS
ASC
ASCENDING
ASSIGN
ASSIGNING
ASYNC
AT
ATN

ATT
ATTRIBUTES
AUTH
AUTHORIZATION
AUTO
AVER
AVG

- B -

BACKOUT
BACKWARD
BASE
BEFORE
BETWEEN
BLOCK
BOT
BOTTOM
BREAK
BROWSE
BUT
BX
BY

- C -

CABINET
CALL
CALLDBPROC
CALLING
CALLNAT
CAP
CAPTIONED
CASE
CC
CD
CDID
CF
CHAR
CHARLENGTH
CHARPOSITION
CHILD
CIPH
CIPHER
CLASS
CLOSE

CLR
COALESCE
CODEPAGE
COMMAND
COMMIT
COMPOSE
COMPRESS
COMPUTE
CONCAT
CONDITION
CONST
CONSTANT
CONTEXT
CONTROL
CONVERSATION
COPIES
COPY
COS
COUNT
COUPLED
CS
CURRENT
CURSOR
CV

- D -

DATA
DATAAREA
DATE
DAY
DAYS
DC
DECIDE
DECIMAL
DEFINE
DEFINITION
DELETE
DELIMITED
DELIMITER
DELIMITERS
DESC
DESCENDING
DIALOG
DIALOG-ID

DIGITS
DIRECTION
DISABLED
DISP
DISPLAY
DISTINCT
DIVIDE
DL
DLOGOFF
DLOGON
DNATIVE
DNRET
DO
DOCUMENT
DOEND
DOWNLOAD
DU
DY
DYNAMIC

- E -

EDITED
EJ
EJECT
ELSE
EM
ENCODED
END
END-ALL
END-BEFORE
END-BREAK
END-BROWSE
END-CLASS
END-DECIDE
END-DEFINE
END-ENDDATA
END-ENDFILE
END-ENDPAGE
END-ERROR
END-FILE
END-FIND
END-FOR
END-FUNCTION
END-HISTOGRAM

ENDHOC
END-IF
END-INTERFACE
END-LOOP
END-METHOD
END-NOREC
END-PARAMETERS
END-PARSE
END-PROCESS
END-PROPERTY
END-PROTOTYPE
END-READ
END-REPEAT
END-RESULT
END-SELECT
END-SORT
END-START
END-SUBROUTINE
END-TOPPAGE
END-WORK
ENDING
ENTER
ENTIRE
ENTR
EQ
EQUAL
ERASE
ERROR
ERRORS
ES
ESCAPE
EVEN
EVENT
EVERY
EXAMINE
EXCEPT
EXISTS
EXIT
EXP
EXPAND
EXPORT
EXTERNAL
EXTRACTING

- F -

FALSE
FC
FETCH
FIELD
FIELDS
FILE
FILL
FILLER
FINAL
FIND
FIRST
FL
FLOAT
FOR
FORM
FORMAT
FORMATTED
FORMATTING
FORMS
FORWARD
FOUND
FRAC
FRAMED
FROM
FS
FULL
FUNCTION
FUNCTIONS

- G -

GC
GE
GEN
GENERATED
GET
GFID
GIVE
GIVING
GLOBAL
GLOBALS
GREATER
GT
GUI

- H -

HANDLE
HAVING
HC
HD
HE
HEADER
HEX
HISTOGRAM
HOLD
HORIZ
HORIZONTALLY
HOUR
HOURS
HW

- I -

IA
IC
ID
IDENTICAL
IF
IGNORE
IM
IMMEDIATE
IMPORT
IN
INC
INCCONT
INCDIC
INCDIR
INCLUDE
INCLUDED
IA
IC
ID
IDENTICAL
IF
IGNORE
IM
IMMEDIATE
IMPORT
IN
INC
INCCONT

INCDIC
INCDIR
INCLUDE
INCLUDED
INCLUDING
INCMAC
INDEPENDENT
INDEX
INDEXED
INDICATOR
INIT
INITIAL
INNER
INPUT
INSENSITIVE
INSERT
INTO
INT
INTEGER
INTERCEPTED
INTERFACE
INTERFACE4
INTERMEDIATE
INTERSECT
INTO
INVERTED
INVESTIGATE
IP
IS
ISN

- J -

JOIN
JUST
JUSTIFIED

- K -

KD
KEEP
KEY
KEYS

- L -

LANGUAGE
LAST
LC
LE
LEAVE
LEAVING
LEFT
LENGTH
LESS
LEVEL
LIB
LIBPW
LIBRARY
LIBRARY-PASSWORD
LIKE
LIMIT
LINDICATOR
LINES
LISTED
LOCAL
LOCKS
LOG
LOG-LS
LOG-PS
LOGICAL
LOOP
LOWER
CASE
LS
LT

- M -

MACROAREA
MAP
MARK
MASK
MAX
MC
MCG
MESSAGES
METHOD
MGID
MICROSECOND
MIN

MINUTE
MODAL
MODIFIED
MODULE
MONTH
MORE
MOVE
MOVING
MP
MS
MT
MULTI-FETCH
MULTIPLY

- N -

NAME
NAMED
NAMESPACE
NATIVE
NAVER
NC
NCOUNT
NE
NEWPAGE
NL
NMIN
NO
NODE
NOHDR
NONE
NORMALIZE
NORMALIZED
NOT
NOTIT
NOTITLE
NULL
NULL-HANDLE
NUMBER
NUMERIC

- O -

OBJECT
OBTAIN
OCCURRENCES

OF
OFF
OFFSET
OLD
ON
ONCE
ONLY
OPEN
OPTIMIZE
OPTIONAL
OPTIONS
OR
ORDER
OUTER
OUTPUT

- P -

PACKAGESET
PAGE
PARAMETER
PARAMETERS
PARENT
PARSE
PASS
PASSW
PASSWORD
PATH
PATTERN
PA1
PA2
PA3
PC
PD
PEN
PERFORM
PF*n*（*n* = 1～9）
PF*nn*（*nn* = 10～99）
PGDN
PGUP
PGM
PHYSICAL
PM
POLICY
POS

POSITION
PREFIX
PRINT
PRINTER
PROCESS
PROCESSING
PROFILE
PROGRAM
PROPERTY
PROTOTYPE
PRTY
PS
PT
PW

- Q -

QUARTER
QUERYNO

- R -

RD
READ
READONLY
REC
RECORD
RECORDS
RECURSIVELY
REDEFINE
REDUCE
REFERENCED
REFERENCING
REINPUT
REJECT
REL
RELATION
RELATIONSHIP
RELEASE
REMAINDER
REPEAT
REPLACE
REPORT
REPORTER
REPOSITION
REQUEST

REQUIRED
RESET
RESETTING
RESIZE
RESPONSE
RESTORE
RESULT
RET
RETAIN
RETAINED
RETRY
RETURN
RETURNS
REVERSED
RG
RIGHT
ROLLBACK
ROUNDED
ROUTINE
ROW
ROWS
RR
RS
RULEVAR
RUN

- S -

SA
SAME
SCAN
SCREEN
SCROLL
SECOND
SELECT
SELECTION
SEND
SENSITIVE
SEPARATE
SEQUENCE
SERVER
SET
SETS
SETTIME
SF

SG
SGN
SHORT
SHOW
SIN
SINGLE
SIZE
SKIP
SL
SM
SOME
SORT
SORTED
SORTKEY
SOUND
SPACE
SPECIFIED
SQL
SQLID
SQRT
STACK
START
STARTING
STATEMENT
STATIC
STATUS
STEP
STOP
STORE
SUBPROGRAM
SUBPROGRAMS
SUBROUTINE
SUBSTR
SUBSTRING
SUBTRACT
SUM
SUPPRESS
SUPPRESSED
SUSPEND
SYMBOL
SYNC
SYSTEM

- T -

TAN
TC
TERMINATE
TEXT
TEXTAREA
TEXTVARIABLE
THAN
THEM
THEN
THRU
TIME
TIMESTAMP
TIMEZONE
TITLE
TO
TOP
TOTAL
TP
TR
TRAILER
TRANSACTION
TRANSFER
TRANSLATE
TREQ
TRUE
TS
TYPE
TYPES

- U -

UC
UNDERLINED
UNION
UNIQUE
UNKNOWN
UNTIL
UPDATE
UPLOAD
UPPER
UR
USED
USER
USING

- V -

VAL
VALUE
VALUES
VARGRAPHIC
VARIABLE
VARIABLES
VERT
VERTICALLY
VIA
VIEW

- W -

WH
WHEN
WHERE
WHILE
WINDOW
WITH
WORK
WRITE
WITH_CTE

- X -

XML

- Y -

YEAR

- Z -

ZD
ZP

Natural 予約キーワードのチェックの実行

一部の Natural キーワードは、変数の名前として使用すると不明瞭になります。 特に、Natural ステートメント（ADD、FIND など）やシステム関数（ABS、SUM、など）を特定するキーワードがこれに該当します。 このようなキーワードを変数の名前として使用した場合、CALLNAT や WRITE のようなオプションのオペランドのコンテキストでその変数を使うことはできません。

例：

```
DEFINE DATA LOCAL
1 ADD (A10)
END-DEFINE
CALLNAT 'MYSUB' ADD 4     /* ADD is regarded as ADD statement
END
```

プログラミングオブジェクト内の変数名をこのようなNatural予約キーワードと照らし合わせてチェックするために、Natural プロファイルパラメータ KCHECK、または COMPOPT システムコマンドの KCHECK オプションを使用できます。

次のテーブルに、KC または KCHECK でチェックされる Natural 予約キーワードのリストを示します。

A-AVER	BREAK	DIVIDE	END-ENDPAGE	END-WORK	INCCONT	NCOUNT	REDU
ABS	BROWSE	DLOGOFF	END-ERROR	ENTIRE	INCDIC	NEWPAGE	REINI
ACCEPT	CALL	DLOGON	END-FILE	ESCAPE	INCDIR	NMIN	REJEC
ADD	CALLDBPROC	DNATIVE	END-FIND	EXAMINE	INCLUDE	NONE	RELEA
ALL	CALLNAT	DO	END-FOR	EXP	INCMAC	NULL-HANDLE	REPEA
A-MAX	CLOSE	DOEND	END-HISTOGRAM	EXPAND	INPUT	OBTAIN	REQU
A-MIN	COMMIT	DOWNLOAD	ENDHOC	EXPORT	INSERT	OLD	RESET
A-NAVER	COMPOSE	EJECT	END-IF	FALSE	INT	ON	RESIZ
A-NCOUNT	COMPRESS	ELSE	END-LOOP	FETCH	INVESTIGATE	OPEN	RESTO
A-NMIN	COMPUTE	END	END-NOREC	FIND	LIMIT	OPTIONS	RET
ANY	COPY	END-ALL	END-PARSE	FOR	LOG	PARSE	RETRY
ASSIGN	COS	END-BEFORE	END-PROCESS	FORMAT	LOOP	PASSW	RETU
A-SUM	COUNT	END-BREAK	END-READ	FRAC	MAP	PERFORM	ROLL
AT	CREATE	END-BROWSE	END-REPEAT	GET	MAX	POS	RULE
ATN	DECIDE	END-DECIDE	END-RESULT	HISTOGRAM	MIN	PRINT	RUN
AVER	DEFINE	END-ENDDATA	END-SELECT	IF	MOVE	PROCESS	SELEC
BACKOUT	DELETE	END-ENDFILE	END-SORT	IGNORE	MULTIPLY	READ	SEND
BEFORE	DISPLAY		END-START	IMPORT	NAVER	REDEFINE	SEPAR
			END-SUBROUTINE				SET
			END-TOPPAGE				

デフォルトでは、キーワードチェックは実行されません。

143 参照プログラム例

このchapterでは、『*プログラミングガイド*』で参照されている追加のプログラム例を示します。

次のトピックについて説明します。

READ ステートメント

次の例は、「*データベースアクセスのためのステートメント*」セクションで参照されています。

READX03 - READ ステートメント（**LOGICAL** 節を含む）

```
** Example 'READX03': READ (with LOGICAL clause)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 PERSONNEL-ID
  2 JOB-TITLE
END-DEFINE
*
LIMIT 8
READ EMPLOY-VIEW LOGICAL BY PERSONNEL-ID
  DISPLAY NOTITLE  *ISN      NAME
                  'PERS-NO'  PERSONNEL-ID
                  'POSITION' JOB-TITLE
END-READ
END
```

プログラム READX03 の出力：

```
   ISN              NAME         PERS-NO          POSITION
----------- -------------------- -------- -------------------------

        204 SCHINDLER            11100102 PROGRAMMIERER
        205 SCHIRM               11100105 SYSTEMPROGRAMMIERER
        206 SCHMITT              11100106 OPERATOR
        207 SCHMIDT              11100107 SEKRETAERIN
        208 SCHNEIDER            11100108 SACHBEARBEITER
        209 SCHNEIDER            11100109 SEKRETAERIN
```

```
        210 BUNGERT              11100110 SYSTEMPROGRAMMIERER
        211 THIELE               11100111 SEKRETAERIN
```

FIND ステートメント

次の例は、「*データベースアクセスのためのステートメント*」セクションで参照されています。

FINDX07 - FIND ステートメント（複数節を含む）

```
** Example 'FINDX07': FIND  (with several clauses)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
  2 SALARY    (1)
  2 CURR-CODE (1)
END-DEFINE
*
FIND EMPLOY-VIEW WITH  PHONETIC-NAME = 'JONES' OR = 'BECKR'
                 AND   CITY          = 'BOSTON' THRU 'NEW YORK'
                 BUT NOT               'CHAPEL HILL'
                 SORTED BY NAME
                 WHERE SALARY (1)    < 28000
  DISPLAY NOTITLE NAME FIRST-NAME CITY SALARY (1)
END-FIND
END
```

プログラム FINDX07 の出力：

```
        NAME              FIRST-NAME              CITY           ANNUAL
                                                                 SALARY
-------------------- -------------------- -------------------- ----------

BAKER                PAULINE              DERBY                      4450
JONES                MARTHA               KALAMAZOO                 21000
JONES                KEVIN                DERBY                      7000
```

FINDX08 - FIND ステートメント（**LIMIT** を含む）

```
** Example 'FINDX08': FIND (with LIMIT)
**                     Demonstrates FIND statement with LIMIT option to
**                     terminate program with an error message if the
**                     number of records selected exceeds a specified
**                     limit (no output).
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 JOB-TITLE
END-DEFINE
*
FIND EMPLOY-VIEW WITH LIMIT (5) JOB-TITLE = 'SALES PERSON'
  DISPLAY NAME JOB-TITLE
END-FIND
END
```

プログラム FINDX08 によって発生するランタイムエラー：

```
NAT1005 検索制限の指定より多くのレコードが見つかりました。
```

FINDX09 - FIND ステートメント（***NUMBER**、***COUNTER**、***ISN** を使用）

```
** Example 'FINDX09': FIND (using *NUMBER, *COUNTER, *ISN)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 DEPT
  2 NAME
END-DEFINE
*
FIND EMPLOY-VIEW WITH CITY  = 'BOSTON'
                 WHERE DEPT = 'TECH00' THRU 'TECH10'
  DISPLAY NOTITLE
          'COUNTER' *COUNTER NAME DEPT 'ISN' *ISN
  AT START OF DATA
    WRITE '(TOTAL NUMBER IN BOSTON:' *NUMBER ')' /
  END-START
END-FIND
END
```

プログラム FINDX09 の出力：

```
  COUNTER           NAME         DEPARTMENT     ISN
                                    CODE
---------- -------------------- ---------- ----------

(TOTAL NUMBER IN BOSTON:           7 )

         1 STANWOOD             TECH10             782
         2 PERREAULT            TECH10             842
```

FINDX10 - FIND ステートメント（**READ** との組み合わせ）

```
** Example 'FINDX10': FIND (combined with READ)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 FIRST-NAME
1 VEHIC-VIEW VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
END-DEFINE
*
LIMIT 15
*
EMP. READ EMPLOY-VIEW BY NAME STARTING FROM 'JONES'
  VEH. FIND VEHIC-VIEW WITH PERSONNEL-ID = PERSONNEL-ID (EMP.)
    IF NO RECORDS FOUND
      MOVE '*** NO CAR ***' TO MAKE
    END-NOREC
    DISPLAY NOTITLE
            NAME (EMP.) (IS=ON)
            FIRST-NAME (EMP.) (IS=ON)
            MAKE (VEH.)
  END-FIND
END-READ
END
```

プログラム FINDX10 の出力：

```
        NAME             FIRST-NAME             MAKE
-------------------- -------------------- --------------------

JONES                VIRGINIA             CHRYSLER
                     MARSHA               CHRYSLER
                                          CHRYSLER
                     ROBERT               GENERAL MOTORS
                     LILLY                FORD
```

```
                                    MG
                    EDWARD          GENERAL MOTORS
                    MARTHA          GENERAL MOTORS
                    LAUREL          GENERAL MOTORS
                    KEVIN           DATSUN
                    GREGORY         FORD
JOPER               MANFRED         *** NO CAR ***
JOUSSELIN           DANIEL          RENAULT
JUBE                GABRIEL         *** NO CAR ***
JUNG                ERNST           *** NO CAR ***
JUNKIN              JEREMY          *** NO CAR ***
KAISER              REINER          *** NO CAR ***
```

FINDX11 - FIND NUMBER ステートメント（***NUMBER** を含む）

```
** Example 'FINDX11': FIND NUMBER (with *NUMBER)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 FIRST-NAME
  2 NAME
  2 CITY
  2 JOB-TITLE
  2 SALARY        (1)
*
1 #PERCENT        (N.2)
1 REDEFINE #PERCENT
  2 #WHOLE-NBR    (N2)
1 #ALL-BOST       (N3.2)
1 #SECR-ONLY      (N3.2)
1 #BOST-NBR       (N3)
1 #SECR-NBR       (N3)
END-DEFINE
*
F1. FIND NUMBER EMPLOY-VIEW WITH CITY = 'BOSTON'
F2. FIND NUMBER EMPLOY-VIEW WITH CITY = 'BOSTON'
                            AND JOB-TITLE = 'SECRETARY'
*
MOVE *NUMBER(F1.) TO #ALL-BOST  #BOST-NBR
MOVE *NUMBER(F2.) TO #SECR-ONLY #SECR-NBR
DIVIDE #ALL-BOST INTO #SECR-ONLY GIVING #PERCENT
*
WRITE TITLE LEFT JUSTIFIED UNDERLINED
  'There are' #BOST-NBR 'employees in the Boston offices.' /
  #SECR-NBR '(=' #WHOLE-NBR (EM=99%')') 'are secretaries.'
*
SKIP 1
FIND EMPLOY-VIEW WITH CITY      = 'BOSTON'
                 AND  JOB-TITLE = 'SECRETARY'
  DISPLAY NAME FIRST-NAME JOB-TITLE SALARY (1)
```

```
END-FIND
END
```

プログラム FINDX11 の出力：

```
There are    7 employees in the Boston offices.
   3 (= 42%) are secretaries.
------------------------------------------------------------------------------

        NAME               FIRST-NAME               CURRENT             ANNUAL
                                                   POSITION             SALARY
-------------------- -------------------- ------------------------- ----------

SHAW                 LESLIE               SECRETARY                      18000
CREMER               WALT                 SECRETARY                      20000
COHEN                JOHN                 SECRETARY                      16000
```

READ および FIND ステートメントのネスト

次の例は、「データベース処理ループ」セクションで参照されています。

READX04 - READ ステートメント（**FIND** およびシステム変数 ***NUMBER** と ***COUNTER** との組み合わせ）

```
** Example 'READX04': READ (in combination with FIND and the system
**                     variables *NUMBER and *COUNTER)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 FIRST-NAME
1 VEHIC-VIEW VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
END-DEFINE
*
LIMIT 10
RD. READ EMPLOY-VIEW BY NAME STARTING FROM 'JONES'
  FD. FIND VEHIC-VIEW WITH PERSONNEL-ID = PERSONNEL-ID (RD.)
    IF NO RECORDS FOUND
      ENTER
    END-NOREC
    /*
    DISPLAY NOTITLE
            *COUNTER (RD.)(NL=8) NAME         (AL=15) FIRST-NAME (AL=10)
```

```
            *NUMBER  (FD.)(NL=8) *COUNTER (FD.)(NL=8) MAKE
  END-FIND
END-READ
END
```

プログラム READX04 の出力：

```
  CNT          NAME       FIRST-NAME   NMBR      CNT             MAKE
--------- --------------- ---------- --------- --------- --------------------

       1 JONES           VIRGINIA           1         1 CHRYSLER
       2 JONES           MARSHA             2         1 CHRYSLER
       2 JONES           MARSHA             2         2 CHRYSLER
       3 JONES           ROBERT             1         1 GENERAL MOTORS
       4 JONES           LILLY              2         1 FORD
       4 JONES           LILLY              2         2 MG
       5 JONES           EDWARD             1         1 GENERAL MOTORS
       6 JONES           MARTHA             1         1 GENERAL MOTORS
       7 JONES           LAUREL             1         1 GENERAL MOTORS
       8 JONES           KEVIN              1         1 DATSUN
       9 JONES           GREGORY            1         1 FORD
      10 JOPER           MANFRED            0         0
```

LIMITX01 - LIMIT ステートメント（**READ**、**FIND** ループ処理）

```
** Example 'LIMITX01': LIMIT  (for READ, FIND loop processing)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 FIRST-NAME
  2 NAME
1 VEH-VIEW VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
END-DEFINE
*
LIMIT 4
*
READ EMPLOY-VIEW BY NAME STARTING FROM 'A'
  FIND VEH-VIEW WITH PERSONNEL-ID = EMPLOY-VIEW.PERSONNEL-ID
    IF NO RECORDS FOUND
      MOVE 'NO CAR' TO MAKE
    END-NOREC
    DISPLAY PERSONNEL-ID NAME FIRST-NAME MAKE
  END-FIND
```

```
END-READ
END
```

プログラム LIMITX01 の出力：

```
Page      1                                                  04-12-13  14:01:57

PERSONNEL-ID         NAME              FIRST-NAME              MAKE
------------ -------------------- -------------------- --------------------

             ABELLAN              KEPA                 NO CAR
30000231     ACHIESON             ROBERT               FORD
             ADAM                 SIMONE               NO CAR
20008800     ADKINSON             JEFF                 GENERAL MOTORS
```

ACCEPT および REJECT ステートメント

次の例は、「*ACCEPT/REJECT を使用したレコードの選択*」セクションで参照されています。

ACCEPX04 - ACCEPT IF ... LESS THAN ... ステートメント

```
** Example 'ACCEPX04': ACCEPT IF ... LES THAN ...
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 JOB-TITLE
  2 SALARY (1)
END-DEFINE
*
LIMIT 20
READ EMPLOY-VIEW BY PERSONNEL-ID = '20017000'
  ACCEPT IF SALARY (1) LESS THAN 38000
  DISPLAY NOTITLE PERSONNEL-ID NAME JOB-TITLE SALARY (1)
END-READ
END
```

プログラム ACCEPX04 の出力：

```
PERSONNEL         NAME                 CURRENT            ANNUAL
   ID                                  POSITION           SALARY
--------- -------------------- ------------------------- ----------

20017000  CREMER               ANALYST                        34000
20017100  MARKUSH              TRAINEE                        22000
20017400  NEEDHAM              PROGRAMMER                     32500
20017500  JACKSON              PROGRAMMER                     33000
20017600  PIETSCH              SECRETARY                      22000
20017700  PAUL                 SECRETARY                      23000
20018000  FARRIS               PROGRAMMER                     30500
20018100  EVANS                PROGRAMMER                     31000
20018200  HERZOG               PROGRAMMER                     31500
20018300  LORIE                SALES PERSON                   28000
20018400  HALL                 SALES PERSON                   30000
20018500  JACKSON              MANAGER                        36000
20018800  SMITH                SECRETARY                      24000
20018900  LOWRY                SECRETARY                      25000
```

ACCEPX05 - ACCEPT IF ... AND ... ステートメント

```
** Example 'ACCEPX05': ACCEPT IF ... AND ...
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 JOB-TITLE
  2 SALARY (1:2)
END-DEFINE
*
LIMIT 6
READ EMPLOY-VIEW PHYSICAL WHERE SALARY(2) > 0
  ACCEPT IF  SALARY(1) > 10000
         AND SALARY(1) < 50000
  DISPLAY (AL=15) 'SALARY I' SALARY (1) 'SALARY II'  SALARY (2)
                   NAME JOB-TITLE  CITY
END-READ
END
```

プログラム ACCEPX05 の出力：

```
Page      1                                                  04-12-13  14:05:28

 SALARY I  SALARY II       NAME           CURRENT          CITY
                                         POSITION
---------- ---------- --------------- --------------- --------------

     48000      46000 SPENGLER        SACHBEARBEITER  DARMSTADT
     45000      40000 SPECK           SACHBEARBEITER  DARMSTADT
     48000      46000 SCHINDLER       PROGRAMMIERER   HEPPENHEIM
     36000      32000 SCHMIDT         SEKRETAERIN     HEPPENHEIM
```

ACCEPX06 - REJECT IF ... OR ... ステートメント

```
** Example 'ACCEPX06': REJECT IF ... OR ...
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 SALARY (1)
  2 JOB-TITLE
  2 CITY
  2 NAME
END-DEFINE
*
LIMIT 20
READ EMPLOY-VIEW LOGICAL BY PERSONNEL-ID = '20017000'
  REJECT IF SALARY (1) < 20000
         OR SALARY (1) > 26000
  DISPLAY NOTITLE SALARY (1) NAME JOB-TITLE CITY
END-READ
END
```

プログラム ACCEPX06 の出力：

```
  ANNUAL           NAME                 CURRENT                  CITY
  SALARY                                POSITION
---------- -------------------- ----------------------- --------------------

     22000 MARKUSH              TRAINEE                 LOS ANGELES
     22000 PIETSCH              SECRETARY               VISTA
     23000 PAUL                 SECRETARY               NORFOLK
```

```
     24000 SMITH                SECRETARY                SILVER SPRING
     25000 LOWRY                SECRETARY                LEXINGTON
```

AT START OF DATA および AT END OF DATA ステートメント

次の例は、「*AT START/END OF DATA ステートメント*」セクションで参照されています。

ATENDX01 - AT END OF DATA ステートメント

```
** Example 'ATENDX01': AT END OF DATA
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 JOB-TITLE
END-DEFINE
*
READ (6) EMPLOY-VIEW BY PERSONNEL-ID FROM '20017000'
  DISPLAY NOTITLE NAME JOB-TITLE
  AT END OF DATA
    WRITE / 'LAST PERSON SELECTED:' OLD(NAME)
  END-ENDDATA
END-READ
END
```

プログラム ATENDX01 の出力：

```
        NAME                  CURRENT
                             POSITION
-------------------- -------------------------

CREMER               ANALYST
MARKUSH              TRAINEE
GEE                  MANAGER
KUNEY                DBA
NEEDHAM              PROGRAMMER
JACKSON              PROGRAMMER

LAST PERSON SELECTED: JACKSON
```

ATSTAX02 - AT START OF DATA ステートメント

```
** Example 'ATSTAX02': AT START OF DATA
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 FIRST-NAME
  2 NAME
  2 SALARY    (1)
  2 CURR-CODE (1)
  2 BONUS     (1,1)
END-DEFINE
*
LIMIT 3
FIND EMPLOY-VIEW WITH CITY = 'MADRID'
  DISPLAY NAME FIRST-NAME SALARY(1) BONUS(1,1) CURR-CODE (1)
  /*
  AT START OF DATA
    WRITE NOTITLE *DAT4E /
  END-START
END-FIND
END
```

プログラム ATSTAX02 の出力：

```
        NAME              FIRST-NAME        ANNUAL     BONUS    CURRENCY
                                            SALARY               CODE
-------------------- -------------------- ---------- ---------- --------

13/12/2004

DE JUAN              JAVIER                1988000          0 PTA
DE LA MADRID         ANSELMO               3120000          0 PTA
PINERO               PAULA                 1756000          0 PTA
```

WRITEX09 - WRITE ステートメント（**AT END OF DATA** との組み合わせ）

```
** Example 'WRITEX09': WRITE (in combination with AT END OF DATA )
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 BIRTH
  2 JOB-TITLE
  2 DEPT
END-DEFINE
*
READ (3) EMPLOY-VIEW BY CITY
```

```
  DISPLAY NOTITLE NAME BIRTH (EM=YYYY-MM-DD) JOB-TITLE
  WRITE 38T 'DEPT CODE:' DEPT
  /*
  AT END OF DATA
    WRITE / 'LAST PERSON SELECTED:' OLD(NAME)
  END-ENDDATA
  SKIP 1
END-READ
END
```

プログラム WRITEX09 の出力：

```
        NAME            DATE             CURRENT
                         OF             POSITION
                       BIRTH
-------------------- ---------- -------------------------

SENKO                1971-09-11 PROGRAMMER
                                      DEPT CODE: TECH10

GODEFROY             1949-01-09 COMPTABLE
                                      DEPT CODE: COMP02

CANALE               1942-01-01 CONSULTANT
                                      DEPT CODE: TECH03


LAST PERSON SELECTED: CANALE
```

DISPLAY および WRITE ステートメント

次の例は、「*DISPLAY および WRITE ステートメント*」セクションで参照されています。

DISPLX13 - DISPLAY ステートメント（**WRITE** を使用する **WRITEX08** と比較）

```
** Example 'DISPLX13': DISPLAY (compare with WRITEX08 using WRITE)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 FIRST-NAME
  2 NAME
  2 SALARY (2)
  2 BONUS  (1,1)
  2 CITY
END-DEFINE
```

```
*
LIMIT 2
READ EMPLOY-VIEW WITH CITY = 'CHAPEL HILL' WHERE BONUS(1,1) NE 0
  /*
  DISPLAY 'PERS/ID' PERSONNEL-ID  NAME / FIRST-NAME
          '**' '=' SALARY(1:2) 'BONUS' BONUS(1,1) CITY (AL=15)
  /*
  SKIP 1
END-READ
END
```

プログラム DISPLX13 の出力：

```
Page      1                                                 04-12-13  14:11:28

  PERS           NAME            ANNUAL       BONUS         CITY
   ID         FIRST-NAME          SALARY
-------- -------------------- ------------ ---------- ---------------

20027000 CUMMINGS             **      41000       1500 CHAPEL HILL
         PUALA                        38900

20000200 WOOLSEY              **      26000       3000 CHAPEL HILL
         LOUISE                       24700
```

WRITEX08 - WRITE ステートメント（**DISPLAY** を使用する **DISPLX13** と比較）

```
** Example 'WRITEX08': WRITE  (compare with DISPLX13 using DISPLAY)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 FIRST-NAME
  2 NAME
  2 SALARY (2)
  2 BONUS  (1,1)
  2 CITY
END-DEFINE
*
LIMIT 2
READ EMPLOY-VIEW WITH CITY = 'CHAPEL HILL' WHERE BONUS(1,1) NE 0
  /*
  WRITE 'PERS/ID' PERSONNEL-ID  NAME / FIRST-NAME
        '**' '=' SALARY(1:2) 'BONUS' BONUS(1,1) CITY (AL=15)
  /*
  SKIP 1
```

```
END-READ
END
```

プログラム WRITEX08 の出力：

```
Page      1                                                  04-12-13  14:12:43

PERS/ID 20027000 CUMMINGS
PUALA                ** ANNUAL SALARY:      41000      38900 BONUS       1500
CHAPEL HILL

PERS/ID 20000200 WOOLSEY
LOUISE               ** ANNUAL SALARY:      26000      24700 BONUS       3000
CHAPEL HILL
```

DISPLX14 - DISPLAY ステートメント（**AL**、**SF**、*n***X** を含む）

```
** Example 'DISPLX14': DISPLAY  (with AL, SF and nX)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 FIRST-NAME
  2 NAME
  2 ADDRESS-LINE (1)
  2 TELEPHONE
    3 AREA-CODE
    3 PHONE
  2 CITY
END-DEFINE
*
READ (3) EMPLOY-VIEW BY NAME STARTING FROM 'W'
  DISPLAY (AL=15 SF=5) NAME CITY / ADDRESS-LINE(1) 2X TELEPHONE
  SKIP 1
END-READ
END
```

プログラム DISPLX14 の出力：

```
Page      1                                                  04-12-13  14:14:00

     NAME               CITY                     TELEPHONE
                        ADDRESS
                                              AREA              TELEPHONE
                                              CODE
---------------     ---------------  ---------------     ---------------

WABER               HEIDELBERG       06221               456452
                    ERBACHERSTR. 78
```

```
WADSWORTH           DERBY            0332                515365
                    56 PINECROFT CO

WAGENBACH           FRANKFURT        069                 983218
                    BECKERSTR. 4
```

WRITEX09 - WRITE ステートメント（**AT END OF DATA** との組み合わせ）

```
** Example 'WRITEX09': WRITE (in combination with AT END OF DATA )
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 BIRTH
  2 JOB-TITLE
  2 DEPT
END-DEFINE
*
READ (3) EMPLOY-VIEW BY CITY
  DISPLAY NOTITLE NAME BIRTH (EM=YYYY-MM-DD) JOB-TITLE
  WRITE 38T 'DEPT CODE:' DEPT
  /*
  AT END OF DATA
    WRITE / 'LAST PERSON SELECTED:' OLD(NAME)
  END-ENDDATA
  SKIP 1
END-READ
END
```

プログラム WRITEX09 の出力：

```
        NAME            DATE             CURRENT
                         OF             POSITION
                       BIRTH
-------------------- ---------- -------------------------

SENKO                1971-09-11 PROGRAMMER
                                DEPT CODE: TECH10

GODEFROY             1949-01-09 COMPTABLE
                                DEPT CODE: COMP02

CANALE               1942-01-01 CONSULTANT
                                DEPT CODE: TECH03
```

```
LAST PERSON SELECTED: CANALE
```

DISPLAY ステートメント

次の例は、「*ページタイトル、改ページ、空行*」セクションで参照されています。

DISPLX21 DISPLAY ステートメント（スラッシュ '/' を含む。**WRITE** と比較）

```
** Example 'DISPLX21': DISPLAY (usage of slash '/' in DISPLAY and WRITE)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 NAME
  2 FIRST-NAME
  2 ADDRESS-LINE (1)
END-DEFINE
*
WRITE TITLE LEFT JUSTIFIED UNDERLINED
       *TIME
   5X  'PEOPLE LIVING IN SALT LAKE CITY'
   21X 'PAGE:' *PAGE-NUMBER /
   15X 'AS OF' *DAT4E //
*
WRITE TRAILER UNDERLINED 'REGISTER OF' / 'SALT LAKE CITY'
*
READ (2) EMPLOY-VIEW WITH CITY = 'SALT LAKE CITY'
  DISPLAY  NAME /
           FIRST-NAME
           'HOME/CITY' CITY
           'STREET/OR BOX NO.' ADDRESS-LINE (1)
  SKIP 1
END-READ
END
```

プログラム DISPLX21 の出力：

```
14:15:50.1     PEOPLE LIVING IN SALT LAKE CITY                    PAGE:      1
               AS OF 13/12/2004


------------------------------------------------------------------------------
        NAME                HOME                STREET
     FIRST-NAME             CITY              OR BOX NO.
-------------------- -------------------- --------------------
```

```
ANDERSON             SALT LAKE CITY       3701 S. GEORGE MASON
JENNY

SAMUELSON            SALT LAKE CITY       7610 W. 86TH STREET
MARTIN

                                    REGISTER OF
                                  SALT LAKE CITY
------------------------------------------------------------------------------
```

列ヘッダー

次の例は、「*列ヘッダー*」セクションで参照されています。

DISPLX15 - DISPLAY ステートメント（**FC**、**UC** を含む）

```
** Example 'DISPLX15': DISPLAY  (with FC, UC)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 FIRST-NAME
  2 NAME
  2 ADDRESS-LINE (1)
  2 CITY
  2 TELEPHONE
    3 AREA-CODE
    3 PHONE
END-DEFINE
*
FORMAT AL=12 GC== UC=%
*
READ (3) EMPLOY-VIEW BY NAME STARTING FROM 'R'
  DISPLAY NOTITLE (FC=*)
          NAME FIRST-NAME CITY  (FC=- UC=-) /
          ADDRESS-LINE(1) TELEPHONE
  SKIP 1
END-READ
END
```

プログラム DISPLX15 の出力：

```
****NAME**** *FIRST-NAME* ----CITY---- ========TELEPHONE========
                          **ADDRESS***
                                        ****AREA**** *TELEPHONE**
                                        ****CODE****
%%%%%%%%%%% %%%%%%%%%%% ------------ %%%%%%%%%%% %%%%%%%%%%%

RACKMANN     MARIAN       FRANKFURT    069          375849
                          FINKENSTR. 1

RAMAMOORTHY  TY           SEPULVEDA    209          175-1885
                          12018 BROOKS

RAMAMOORTHY  TIMMIE       SEATTLE      206          151-4673
                          921-178TH PL
```

DISPLX16 - DISPLAY ステートメント（'/'、'*text*'、'*text*/*text*' を含む）

```
** Example 'DISPLX16': DISPLAY  (with '/', 'text', 'text/text')
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 FIRST-NAME
  2 NAME
  2 ADDRESS-LINE (1)
  2 CITY
  2 TELEPHONE
    3 AREA-CODE
    3 PHONE
END-DEFINE
*
READ (5) EMPLOY-VIEW BY NAME STARTING FROM 'E'
  DISPLAY NOTITLE
    '/'          NAME       (AL=12) /* suppressed header
    'FIRST/NAME' FIRST-NAME (AL=10) /* two-line user-defined header
    'ADDRESS'    CITY /             /* user-defined header
    ' '          ADDRESS-LINE(1)    /* 'blank' header
                 TELEPHONE (HC=L)   /* default header
  SKIP 1
END-READ
END
```

プログラム DISPLX16 の出力：

```
               FIRST          ADDRESS        TELEPHONE
                NAME
                                              AREA   TELEPHONE
                                              CODE
            ---------- -------------------- ------ ---------------

EAVES       TREVOR     DERBY                0332   657623
                       17 HARTON ROAD

ECKERT      KARL       OBERRAMSTADT         06154  99722
                       FORSTWEG 22

ECKHARDT    RICHARD    DARMSTADT
                       BRESLAUERPL. 4

EDMUNDSON   LES        TULSA                918    945-4916
                       2415 ALSOP CT.

EGGERT      HERMANN    STUTTGART            0711   981237
                       RABENGASSE 8
```

フィールド出力関連パラメータ

次の例は、「*フィールドの出力に影響を与えるパラメータ*」セクションで参照されています。

これらの例は、LC、IC、TC、AL、NL、IS、ZP、ES の各パラメータ、および SUSPEND IDENTICAL SUPPRESS ステートメントの使用法を示すために提供されています。

DISPLX17 - DISPLAY ステートメント（**NL**、**AL**、**IC**、**LC**、**TC** を含む）

```
** Example 'DISPLX17': DISPLAY (with NL, AL, IC, LC, TC)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 FIRST-NAME
  2 NAME
  2 SALARY (1)
  2 BONUS  (1,1)
END-DEFINE
*
READ (3) EMPLOY-VIEW BY NAME STARTING FROM 'JONES'
  DISPLAY NOTITLE (IS=ON NL=15)
                      NAME
    '-' '='           FIRST-NAME (AL=12)
    'ANNUAL SALARY' SALARY(1)  (LC=USD   TC=.00)   /
    '+ BONUSES'     BONUS(1,1) (IC='+  ' TC=.00)
  SKIP 1
```

```
END-READ
END
```

プログラム DISPLX17 の出力：

```
        NAME           FIRST-NAME       ANNUAL SALARY
                                          + BONUSES
-------------------- -------------- ---------------------

JONES                - VIRGINIA     USD           46000.00
                                                +  9000.00

                     - MARSHA       USD           50000.00
                                                   +  0.00

                     - ROBERT       USD           31000.00
                                                   +  0.00
```

DISPLX18 - DISPLAY ステートメント（**SF**、**AL**、**UC**、**LC**、**IC**、**TC** のデフォルト設定を使用。**DISPLX19** と比較）

```
** Example 'DISPLX18': DISPLAY  (using default settings for SF, AL, UC,
**                      LC, IC, TC and compare with DISPLX19)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
  2 SALARY    (1)
  2 BONUS     (1,1)
END-DEFINE
*
FIND (6) EMPLOY-VIEW WITH CITY = 'CHAPEL HILL'
  DISPLAY NAME FIRST-NAME SALARY(1) BONUS(1,1)
END-FIND
END
```

プログラム DISPLX18 の出力：

```
Page      1                                                 04-12-13  14:20:48

        NAME              FIRST-NAME        ANNUAL     BONUS
                                            SALARY
-------------------- -------------------- ---------- ----------

KESSLER              CLARE                     41000          0
ADKINSON             DAVID                     24000          0
GEE                  TOMMIE                    39500          0
```

```
HERZOG               JOHN                        31500          0
QUILLION             TIMOTHY                     30500          0
CUMMINGS             PUALA                       41000       1500
```

DISPLX19 - DISPLAY ステートメント（**SF**、**AL**、**LC**、**IC**、**TC** を含む。**DISPLX18** と比較）

```
** Example 'DISPLX19': DISPLAY  (with SF, AL, LC, IC, TC and compare
**                      with DISPLX19)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 FIRST-NAME
  2 CITY
  2 SALARY (1)
  2 BONUS  (1,1)
END-DEFINE
*
FORMAT SF=3 AL=15 UC==
*
FIND (6) EMPLOY-VIEW WITH CITY = 'CHAPEL HILL'
  DISPLAY (NL=10)
    NAME
    FIRST-NAME  (LC='- ' UC=-)
    SALARY (1)  (LC=USD)
    BONUS (1,1) (IC='*** ' TC=' ***')
END-FIND
END
```

プログラム DISPLX19 の出力：

```
Page      1                                                 04-12-13  14:21:57

     NAME            FIRST-NAME           ANNUAL              BONUS
                                          SALARY
===============   -----------------   ==============   ====================

KESSLER           - CLARE             USD      41000             *** 0 ***
ADKINSON          - DAVID             USD      24000             *** 0 ***
GEE               - TOMMIE            USD      39500             *** 0 ***
HERZOG            - JOHN              USD      31500             *** 0 ***
QUILLION          - TIMOTHY           USD      30500             *** 0 ***
CUMMINGS          - PUALA             USD      41000          *** 1500 ***
```

SUSPEX01 - SUSPEND IDENTICAL SUPPRESS ステートメント（**DISPLAY** のパラメータ **IS**、**ES**、**ZP** との組み合わせ）

```
** Example 'SUSPEX01': SUSPEND IDENTICAL SUPPRESS (in conjunction with
**                      parameters IS, ES, ZP in DISPLAY)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 FIRST-NAME
  2 NAME
  2 CITY
1 VEH-VIEW VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
END-DEFINE
*
LIMIT 15
RD. READ EMPLOY-VIEW BY NAME STARTING FROM 'JONES'
  SUSPEND IDENTICAL SUPPRESS
  FD. FIND VEH-VIEW WITH PERSONNEL-ID = PERSONNEL-ID (RD.)
    IF NO RECORDS FOUND
      MOVE '******' TO MAKE
    END-NOREC
    DISPLAY NOTITLE (ES=OFF IS=ON ZP=ON AL=15)
            NAME       (RD.)
            FIRST-NAME (RD.)
            MAKE       (FD.) (IS=OFF)
  END-FIND
END-READ
END
```

プログラム SUSPEX01 の出力：

```
     NAME          FIRST-NAME          MAKE
--------------- --------------- ---------------

JONES           VIRGINIA        CHRYSLER
JONES           MARSHA          CHRYSLER
                                CHRYSLER
JONES           ROBERT          GENERAL MOTORS
JONES           LILLY           FORD
                                MG
JONES           EDWARD          GENERAL MOTORS
JONES           MARTHA          GENERAL MOTORS
JONES           LAUREL          GENERAL MOTORS
JONES           KEVIN           DATSUN
JONES           GREGORY         FORD
JOPER           MANFRED         ******
JOUSSELIN       DANIEL          RENAULT
JUBE            GABRIEL         ******
JUNG            ERNST           ******
```

```
JUNKIN          JEREMY          ******
KAISER          REINER          ******
```

SUSPEX02 - SUSPEND IDENTICAL SUPPRESS ステートメント（**DISPLAY** のパラメータ **IS**、**ES**、**ZP** との組み合わせ）**IS=OFF** である以外は **SUSPEX01** と同じ

```
** Example 'SUSPEX02': SUSPEND IDENTICAL SUPPRESS (in conjunction with
**                     parameters IS, ES, ZP in DISPLAY)
**                     Identical to SUSPEX01, but with IS=OFF.
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 FIRST-NAME
  2 NAME
  2 CITY
1 VEH-VIEW VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
END-DEFINE
*
LIMIT 15
RD. READ EMPLOY-VIEW BY NAME STARTING FROM 'JONES'
  SUSPEND IDENTICAL SUPPRESS
  FD. FIND VEH-VIEW WITH PERSONNEL-ID = PERSONNEL-ID (RD.)
    IF NO RECORDS FOUND
      MOVE '******' TO MAKE
    END-NOREC
    DISPLAY NOTITLE (ES=OFF IS=OFF ZP=ON AL=15)
            NAME       (RD.)
            FIRST-NAME (RD.)
            MAKE       (FD.) (IS=OFF)
  END-FIND
END-READ
END
```

プログラム SUSPEX02 の出力：

```
     NAME         FIRST-NAME         MAKE
--------------- --------------- ---------------

JONES           VIRGINIA        CHRYSLER
JONES           MARSHA          CHRYSLER
JONES           MARSHA          CHRYSLER
JONES           ROBERT          GENERAL MOTORS
JONES           LILLY           FORD
JONES           LILLY           MG
JONES           EDWARD          GENERAL MOTORS
JONES           MARTHA          GENERAL MOTORS
```

```
JONES           LAUREL          GENERAL MOTORS
JONES           KEVIN           DATSUN
JONES           GREGORY         FORD
JOPER           MANFRED         ******
JOUSSELIN       DANIEL          RENAULT
JUBE            GABRIEL         ******
JUNG            ERNST           ******
JUNKIN          JEREMY          ******
KAISER          REINER          ******
```

COMPRX03 - COMPRESS ステートメント

```
** Example 'COMPRX03': COMPRESS (using parameters LC and TC)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 SALARY     (1)
  2 CURR-CODE  (1)
  2 LEAVE-DUE
  2 NAME
  2 FIRST-NAME
  2 JOB-TITLE
*
1 #SALARY      (N9)
1 #FULL-SALARY (A25)
1 #VACATION    (A11)
END-DEFINE
*
READ (3) EMPLOY-VIEW WITH CITY = 'BOSTON'
  MOVE SALARY(1) TO #SALARY
  COMPRESS 'SALARY  :' CURR-CODE(1) #SALARY INTO #FULL-SALARY
  COMPRESS 'VACATION:' LEAVE-DUE            INTO #VACATION
  /*
  DISPLAY NOTITLE NAME FIRST-NAME
            'JOB DESCRIPTION' JOB-TITLE  (LC='JOB     : ')  /
            '/'               #FULL-SALARY                   /
            '/'               #VACATION  (TC='DAYS')
  SKIP 1
END-READ
END
```

プログラム COMPRX03 の出力：

```
        NAME              FIRST-NAME               JOB DESCRIPTION
-------------------- -------------------- ----------------------------------

SHAW                 LESLIE               JOB     : SECRETARY
                                          SALARY  : USD 18000
                                          VACATION: 2DAYS

STANWOOD             VERNON               JOB     : PROGRAMMER
                                          SALARY  : USD 31000
                                          VACATION: 1DAYS

CREMER               WALT                 JOB     : SECRETARY
                                          SALARY  : USD 20000
                                          VACATION: 3DAYS
```

編集マスク

次の例は、「*編集マスク - EM パラメータ*」セクションで参照されています。

EDITMX03 - 編集マスク（英数字フィールドの異なる **EM**）

```
** Example 'EDITMX03': Edit mask (different EM for alpha-numeric fields)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 FIRST-NAME
  2 NAME
  2 CITY
  2 SALARY(1)
END-DEFINE
*
LIMIT 3
READ EMPLOY-VIEW BY PERSONNEL-ID FROM '20018000'
                 WHERE SALARY(1) = 28000 THRU 30000
  DISPLAY 'N A M E'  NAME       (EM=X^^X^^X^^X^^X^^X^^X^^X^^X^^X) /
          'NAME HEX' NAME       (EM=H^H^H^H^H^H^H^H^H^H^H)
                     FIRST-NAME (EM=' - 'X(15)*)
                     CITY       (EM=X..X(10))
  SKIP 1
END-READ
END
```

プログラム EDITMX03 の出力：

```
Page      1                                                  04-12-13  14:26:57

            N A M E                  FIRST-NAME         CITY
            NAME HEX
------------------------------ ------------------ -------------

L  O  R  I  E                    - JEAN-PAUL      * C..LEVELAND
D3 D6 D9 C9 C5 40 40 40 40 40 40

H  A  L  L                       - ARTHUR         * A..NN ARBER
C8 C1 D3 D3 40 40 40 40 40 40 40

V  A  S  W  A  N  I              - TOMMIE         * M..ONTERREY
E5 C1 E2 E6 C1 D5 C9 40 40 40 40
```

EDITMX04 - 編集マスク（数値フィールドの異なる **EM**）

```
** Example 'EDITMX04': Edit mask (different EM for numeric fields)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 FIRST-NAME
  2 NAME
  2 SALARY (1)
  2 BONUS  (1,1)
  2 LEAVE-DUE
END-DEFINE
*
LIMIT 2
READ EMPLOY-VIEW BY PERSONNEL-ID = '20018000'
                 WHERE SALARY(1) = 28000 THRU 30000
  DISPLAY (SF=4)
          'N A M E'    NAME
          'SALARY'     SALARY(1)  (EM=*USD^ZZZ,999)
          'BONUS (ZZ)' BONUS(1,1) (EM=S*ZZZ,999) /
          'BONUS (Z9)' BONUS(1,1) (EM=SZ99,999+) /
          '->' '='     BONUS(1,1) (EM=-999,999)
          'VAC/DUE'    LEAVE-DUE  (EM=+999)
  SKIP 1
END-READ
END
```

プログラム EDITMX04 の出力：

```
Page      1                                                 04-12-13  14:27:43

      N A M E             SALARY       BONUS (ZZ)     VAC
                                       BONUS (Z9)     DUE
                                          BONUS
--------------------    -----------    -----------    ---

LORIE                   USD *28,000    +**4,000       +13
                                       + 04,000+
                                       ->  004,000

HALL                    USD *30,000    +**5,000       +14
                                       + 05,000+
                                       ->  005,000
```

EDITMX05 - 編集マスク（日付および時刻システム変数の **EM**）

```
** Example 'EDITMX05': Edit mask (EM for date and time system variables)
************************************************************************
WRITE NOTITLE //
  'DATE INTERNAL :' *DATX (DF=L) /
  '              :' *DATX (EM=N(9)' 'ZW.'WEEK 'YYYY)  /
  '              :' *DATX (EM=ZZJ'.DAY 'YYYY)        /
  '     ROMAN    :' *DATX (EM=R)  /
  '     AMERICAN :' *DATX (EM=MM/DD/YYYY)      12X 'OR  ' *DAT4U /
  '     JULIAN   :' *DATX (EM=YYYYJJJ)         15X 'OR  ' *DAT4J /
  '     GREGORIAN:' *DATX (EM=ZD.''L(10)''YYYY) 5X 'OR  ' *DATG ///
  'TIME INTERNAL :' *TIMX                      14X 'OR  ' *TIME /
  '              :' *TIMX (EM=HH.II.SS.T) /
  '              :' *TIMX (EM=HH.II.SS' 'AP) /
  '              :' *TIMX (EM=HH)
END
```

プログラム EDITMX05 の出力：

```


DATE INTERNAL : 2004-12-13
              : Monday 51.WEEK 2004
              : 348.DAY 2004
     ROMAN    : MMIV
     AMERICAN : 12/13/2004           OR   12/13/2004
     JULIAN   : 2004348              OR   2004348
     GREGORIAN: 13.December2004      OR   13December 2004


TIME INTERNAL : 14:28:49             OR   14:28:49.1
              : 14.28.49.1
```

```
              : 02.28.49 PM
              : 14
```

WRITE ステートメントを含む DISPLAY VERT

WRITEX10 - WRITE ステートメント（*n***T**、**T*****field*、**P*****field* を含む）

```
** Example 'WRITEX10': WRITE (with nT, T*field and P*field)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 JOB-TITLE
  2 NAME
  2 SALARY (1)
  2 BONUS  (1,1)
END-DEFINE
*
READ (3) EMPLOY-VIEW WITH JOB-TITLE FROM 'SALES PERSON'
  DISPLAY NOTITLE NAME 30T JOB-TITLE
          VERT AS 'SALARY/BONUS' SALARY(1) BONUS(1,1)
  AT BREAK OF JOB-TITLE
    WRITE 20T 'AVERAGE' T*JOB-TITLE OLD(JOB-TITLE) (AL=15)
              '(SAL)'   P*SALARY AVER(SALARY(1))  /
          46T '(BON)'   P*BONUS  AVER(BONUS(1,1)) /
  END-BREAK
  SKIP 1
END-READ
END
```

プログラム WRITEX10 の出力：

```
        NAME                          CURRENT            SALARY
                                     POSITION            BONUS
--------------------          ------------------------ ----------

SAMUELSON                     SALES PERSON                  32000
                                                             6000

PAPAYANOPOULOS                SALES PERSON                  34000
                                                             7000

HELL                          SALES PERSON                  38000
                                                             9000
```

```
                  AVERAGE   SALES PERSON    (SAL)          34666
                                            (BON)           7333
```

AT BREAK ステートメント

次の例は、「*コントロールブレイク*」セクションで参照されています。

ATBREX06 - AT BREAK OF ステートメント（**NMIN**、**NAVER**、**NCOUNT** を **MIN**、**AVER**、**COUNT** と比較）

```
** Example 'ATBREX06': AT BREAK OF (comparing NMIN, NAVER, NCOUNT with
**                      MIN, AVER, COUNT)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 SALARY (1:2)
END-DEFINE
*
WRITE TITLE '-- SALARY STATISTICS BY CITY --' /
*
READ (2) EMPLOY-VIEW WITH CITY = 'NEW YORK'
  DISPLAY CITY  'SALARY (1)' SALARY(1) 15X 'SALARY (2)' SALARY(2)
  AT BREAK OF CITY
    WRITE /
      14T 'S A L A R Y   (1)'         39T 'S A L A R Y  (2)'          /
      13T '-   MIN:' MIN(SALARY(1))   38T '-   MIN:' MIN(SALARY(2))   /
      13T '-  AVER:' AVER(SALARY(1))  38T '-  AVER:' AVER(SALARY(2))  /
      16T COUNT(SALARY(1)) 'RECORDS'  41T COUNT(SALARY(2)) 'RECORDS'  //
      13T '-  NMIN:' NMIN(SALARY(1))  38T '-  NMIN:' NMIN(SALARY(2))  /
      13T '- NAVER:' NAVER(SALARY(1)) 38T '- NAVER:' NAVER(SALARY(2)) /
      16T NCOUNT(SALARY(1)) 'RECORDS' 41T NCOUNT(SALARY(2)) 'RECORDS'
  END-BREAK
END-READ
END
```

プログラム ATBREX06 の出力：

```
                        -- SALARY STATISTICS BY CITY --

        CITY          SALARY (1)               SALARY (2)
-------------------- ----------               ----------

NEW YORK                   17000                    16100
NEW YORK                   38000                    34900
```

```
             S A L A R Y   (1)        S A L A R Y  (2)
            -   MIN:      17000      -   MIN:      16100
            -  AVER:      27500      -  AVER:      25500
                     2 RECORDS                2 RECORDS

            -  NMIN:      17000      -  NMIN:      16100
            - NAVER:      27500      - NAVER:      25500
                     2 RECORDS                2 RECORDS
```

COMPUTE、MOVE、および COMPRESS ステートメント

次の例は、「*データ計算*」セクションで参照されています。

WRITEX11 - WRITE ステートメント（*n***X**、*n*/*n*、および **COMPRESS** を含む）

```
** Example 'WRITEX11': WRITE (with nX, n/n and COMPRESS)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 SALARY       (1)
  2 FIRST-NAME
  2 NAME
  2 CITY
  2 ZIP
  2 CURR-CODE    (1)
  2 JOB-TITLE
  2 LEAVE-DUE
  2 ADDRESS-LINE (1)
*
1 #SALARY        (A8)
1 #FULL-NAME     (A25)
1 #FULL-CITY     (A25)
1 #FULL-SALARY   (A25)
1 #VACATION      (A16)
END-DEFINE
*
READ (3) EMPLOY-VIEW LOGICAL BY PERSONNEL-ID = '2001800'
  MOVE SALARY(1) TO  #SALARY
  COMPRESS FIRST-NAME NAME                    INTO #FULL-NAME
  COMPRESS ZIP        CITY                    INTO #FULL-CITY
  COMPRESS 'SALARY  :'  CURR-CODE(1)  #SALARY INTO #FULL-SALARY
  COMPRESS 'VACATION:'  LEAVE-DUE    'DAYS'   INTO #VACATION
  /*
  DISPLAY NOTITLE 'NAME AND ADDRESS' NAME
               5X 'PERS-NO.'         PERSONNEL-ID
               3X 'JOB TITLE'        JOB-TITLE (LC='JOB     : ')
  WRITE   1/5 #FULL-NAME        1/37  #FULL-SALARY
```

```
          2/5 ADDRESS-LINE(1)   2/37  #VACATION
          3/5 #FULL-CITY
  SKIP 1
END-READ
END
```

プログラム WRITEX11 の出力：

```
  NAME AND ADDRESS       PERS-NO.               JOB TITLE
--------------------     --------   ----------------------------------

FARRIS                   20018000   JOB     : PROGRAMMER
    JACKIE FARRIS                   SALARY  : USD 30500
    918 ELM STREET                  VACATION: 10 DAY
    32306 TALLAHASSEE

EVANS                    20018100   JOB     : PROGRAMMER
    JO EVANS                        SALARY  : USD 31000
    1058 REDSTONE LANE              VACATION: 11 DAY
    68508 LINCOLN

HERZOG                   20018200   JOB     : PROGRAMMER
    JOHN HERZOG                     SALARY  : USD 31500
    255 ZANG STREET #253            VACATION: 12 DAY
    27514 CHAPEL HILL
```

IFX03 - IF ステートメント

```
** Example 'IFX03': IF
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 CITY
  2 BONUS  (1,1)
  2 SALARY (1)
*
1 #INCOME  (N9)
1 #TEXT    (A26)
END-DEFINE
*
WRITE TITLE '-- DISTRIBUTION OF CATALOGS I AND II --' /
*
READ (3) EMPLOY-VIEW BY CITY = 'SAN FRANSISCO'
  COMPUTE #INCOME = BONUS(1,1) + SALARY(1)
  /*
  IF #INCOME > 40000
    MOVE 'CATALOGS I AND II' TO #TEXT
  ELSE
```

```
    MOVE 'CATALOG I'          TO #TEXT
  END-IF
  /*
  DISPLAY NAME 5X 'SALARY' SALARY(1) / BONUS(1,1)
  WRITE T*SALARY '-'(10) /
        16X 'INCOME:' T*SALARY #INCOME 3X #TEXT /
        16X '='(19)
  SKIP 1
END-READ
END
```

プログラム IFX03 の出力：

```
                    -- DISTRIBUTION OF CATALOGS I AND II --

        NAME               SALARY
                           BONUS
--------------------     ----------

COLVILLE JR                   56000
                                  0
                           ----------
                INCOME:       56000   CATALOGS I AND II
                ====================

RICHMOND                       9150
                                  0
                           ----------
                INCOME:        9150   CATALOG I
                ====================

MONKTON                       13500
                                600
                           ----------
                INCOME:       14100   CATALOG I
                ====================
```

COMPRX03 - COMPRESS ステートメント（パラメータ **LC** および **TC** を使用）

```
** Example 'COMPRX03': COMPRESS (using parameters LC and TC)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 SALARY     (1)
  2 CURR-CODE  (1)
  2 LEAVE-DUE
  2 NAME
  2 FIRST-NAME
```

```
  2 JOB-TITLE
*
1 #SALARY      (N9)
1 #FULL-SALARY (A25)
1 #VACATION    (A11)
END-DEFINE
*
READ (3) EMPLOY-VIEW WITH CITY = 'BOSTON'
  MOVE SALARY(1) TO #SALARY
  COMPRESS 'SALARY  :' CURR-CODE(1) #SALARY INTO #FULL-SALARY
  COMPRESS 'VACATION:' LEAVE-DUE            INTO #VACATION
  /*
  DISPLAY NOTITLE NAME FIRST-NAME
          'JOB DESCRIPTION' JOB-TITLE  (LC='JOB     : ')  /
          '/'               #FULL-SALARY                   /
          '/'               #VACATION  (TC='DAYS')
  SKIP 1
END-READ
END
```

プログラム COMPRX03 の出力：

```
        NAME              FIRST-NAME               JOB DESCRIPTION
-------------------- -------------------- -----------------------------------

SHAW                 LESLIE               JOB     : SECRETARY
                                          SALARY  : USD 18000
                                          VACATION: 2DAYS

STANWOOD             VERNON               JOB     : PROGRAMMER
                                          SALARY  : USD 31000
                                          VACATION: 1DAYS

CREMER               WALT                 JOB     : SECRETARY
                                          SALARY  : USD 20000
                                          VACATION: 3DAYS
```

システム変数

次の例は、「*システム変数とシステム関数*」で参照されているものです。

EDITMX05 - 編集マスク（日付および時刻システム変数の **EM**）

```
** Example 'EDITMX05': Edit mask (EM for date and time system variables)
************************************************************************
WRITE NOTITLE //
  'DATE INTERNAL :' *DATX (DF=L) /
  '              :' *DATX (EM=N(9)' 'ZW.'WEEK 'YYYY)  /
  '              :' *DATX (EM=ZZJ'.DAY 'YYYY)        /
  '     ROMAN    :' *DATX (EM=R)  /
  '     AMERICAN :' *DATX (EM=MM/DD/YYYY)      12X 'OR  ' *DAT4U /
  '     JULIAN   :' *DATX (EM=YYYYJJJ)         15X 'OR  ' *DAT4J /
  '     GREGORIAN:' *DATX (EM=ZD.''L(10)''YYYY) 5X 'OR  ' *DATG ///
  'TIME INTERNAL :' *TIMX                      14X 'OR  ' *TIME /
  '              :' *TIMX (EM=HH.II.SS.T) /
  '              :' *TIMX (EM=HH.II.SS' 'AP) /
  '              :' *TIMX (EM=HH)
END
```

プログラム EDITMX05 の出力：

```
DATE INTERNAL : 2004-12-13
              : Monday 51.WEEK 2004
              : 348.DAY 2004
     ROMAN    : MMIV
     AMERICAN : 12/13/2004           OR   12/13/2004
     JULIAN   : 2004348              OR   2004348
     GREGORIAN: 13.December2004      OR   13December 2004


TIME INTERNAL : 14:36:58             OR   14:36:58.8
              : 14.36.58.8
              : 02.36.58 PM
              : 14
```

READX04 - READ ステートメント（**FIND** およびシステム変数 ***NUMBER** と ***COUNTER** との組み合わせ）

```
** Example 'READX04': READ (in combination with FIND and the system
**                     variables *NUMBER and *COUNTER)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 FIRST-NAME
1 VEHIC-VIEW VIEW OF VEHICLES
  2 PERSONNEL-ID
  2 MAKE
```

```
END-DEFINE
*
LIMIT 10
RD. READ EMPLOY-VIEW BY NAME STARTING FROM 'JONES'
  FD. FIND VEHIC-VIEW WITH PERSONNEL-ID = PERSONNEL-ID (RD.)
    IF NO RECORDS FOUND
      ENTER
    END-NOREC
    /*
    DISPLAY NOTITLE
            *COUNTER (RD.)(NL=8) NAME         (AL=15) FIRST-NAME (AL=10)
            *NUMBER  (FD.)(NL=8) *COUNTER (FD.)(NL=8) MAKE
  END-FIND
END-READ
END
```

プログラム READX04 の出力：

```
  CNT          NAME       FIRST-NAME   NMBR      CNT             MAKE
--------- --------------- ---------- --------- --------- --------------------

        1 JONES           VIRGINIA           1         1 CHRYSLER
        2 JONES           MARSHA             2         1 CHRYSLER
        2 JONES           MARSHA             2         2 CHRYSLER
        3 JONES           ROBERT             1         1 GENERAL MOTORS
        4 JONES           LILLY              2         1 FORD
        4 JONES           LILLY              2         2 MG
        5 JONES           EDWARD             1         1 GENERAL MOTORS
        6 JONES           MARTHA             1         1 GENERAL MOTORS
        7 JONES           LAUREL             1         1 GENERAL MOTORS
        8 JONES           KEVIN              1         1 DATSUN
        9 JONES           GREGORY            1         1 FORD
       10 JOPER           MANFRED            0         0
```

WTITLX01 - WRITE TITLE ステートメント（***PAGE-NUMBER** を含む）

```
** Example 'WTITLX01': WRITE TITLE (with *PAGE-NUMBER)
************************************************************************
DEFINE DATA LOCAL
1 VEHIC-VIEW VIEW OF VEHICLES
  2 MAKE
  2 YEAR
  2 MAINT-COST (1)
END-DEFINE
*
LIMIT 5
*
READ VEHIC-VIEW
END-ALL
```

```
SORT BY YEAR USING MAKE MAINT-COST (1)
  DISPLAY NOTITLE YEAR MAKE MAINT-COST (1)
  AT BREAK OF YEAR
    MOVE 1 TO *PAGE-NUMBER
    NEWPAGE
  END-BREAK
  /*
  WRITE TITLE LEFT JUSTIFIED
        'YEAR:' YEAR 15X 'PAGE' *PAGE-NUMBER
END-SORT
END
```

プログラム WTITLX01 の出力：

```
YEAR:  1980               PAGE      1
YEAR          MAKE         MAINT-COST
----- -------------------- ----------

 1980 RENAULT                  20000
 1980 RENAULT                  20000
 1980 PEUGEOT                  20000
```

システム関数

次の例は、「*システム変数とシステム関数*」で参照されているものです。

ATBREX06 - AT BREAK OF ステートメント（**NMIN**、**NAVER**、**NCOUNT** を **MIN**、**AVER**、**COUNT** と比較）

```
** Example 'ATBREX06': AT BREAK OF (comparing NMIN, NAVER, NCOUNT with
**                      MIN, AVER, COUNT)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 CITY
  2 SALARY (1:2)
END-DEFINE
*
WRITE TITLE '-- SALARY STATISTICS BY CITY --' /
*
READ (2) EMPLOY-VIEW WITH CITY = 'NEW YORK'
  DISPLAY CITY  'SALARY (1)' SALARY(1) 15X 'SALARY (2)' SALARY(2)
  AT BREAK OF CITY
    WRITE /
      14T 'S A L A R Y   (1)'         39T 'S A L A R Y  (2)'          /
      13T '-   MIN:' MIN(SALARY(1))   38T '-   MIN:' MIN(SALARY(2))   /
```

```
      13T '-  AVER:' AVER(SALARY(1))  38T '-  AVER:' AVER(SALARY(2))  /
      16T COUNT(SALARY(1)) 'RECORDS'  41T COUNT(SALARY(2)) 'RECORDS'  //
      13T '-  NMIN:' NMIN(SALARY(1))  38T '-  NMIN:' NMIN(SALARY(2))  /
      13T '- NAVER:' NAVER(SALARY(1)) 38T '- NAVER:' NAVER(SALARY(2)) /
      16T NCOUNT(SALARY(1)) 'RECORDS' 41T NCOUNT(SALARY(2)) 'RECORDS'
  END-BREAK
END-READ
END
```

プログラム ATBREX06 の出力：

```
                        -- SALARY STATISTICS BY CITY --

        CITY          SALARY (1)                SALARY (2)
-------------------- ----------                ----------

NEW YORK                   17000                     16100
NEW YORK                   38000                     34900

             S A L A R Y   (1)         S A L A R Y  (2)
            -   MIN:      17000       -   MIN:      16100
            -  AVER:      27500       -  AVER:      25500
                      2 RECORDS                 2 RECORDS

            -  NMIN:      17000       -  NMIN:      16100
            - NAVER:      27500       - NAVER:      25500
                      2 RECORDS                 2 RECORDS
```

ATENPX01 - AT END OF PAGE ステートメント（**DISPLAY** の **GIVE SYSTEM FUNCTIONS** 経由で使用可能なシステム関数を含む）

```
** Example 'ATENPX01': AT END OF PAGE (with system function available
**                     via GIVE SYSTEM FUNCTIONS in DISPLAY)
************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 PERSONNEL-ID
  2 NAME
  2 JOB-TITLE
  2 SALARY (1)
END-DEFINE
*
READ (10) EMPLOY-VIEW BY PERSONNEL-ID = '20017000'
  DISPLAY NOTITLE GIVE SYSTEM FUNCTIONS
          NAME JOB-TITLE 'SALARY' SALARY(1)
  /*
  AT END OF PAGE
    WRITE / 24T 'AVERAGE SALARY: ...' AVER(SALARY(1))
  END-ENDPAGE
```

```
END-READ
END
```

プログラム ATENPX01 の出力：

```
        NAME                   CURRENT             SALARY
                              POSITION
-------------------- ------------------------- ----------

CREMER               ANALYST                         34000
MARKUSH              TRAINEE                         22000
GEE                  MANAGER                         39500
KUNEY                DBA                             40200
NEEDHAM              PROGRAMMER                      32500
JACKSON              PROGRAMMER                      33000
PIETSCH              SECRETARY                       22000
PAUL                 SECRETARY                       23000
HERZOG               MANAGER                         48500
DEKKER               DBA                             48000

                       AVERAGE SALARY: ...       34270
```

索引

シンボル

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

W

X

Y

Z

あ

い

う

え

お

か

き

く

け

こ

さ

し

す

せ

そ

た

ち

つ

て

と

な

に

ね

は

ひ

ふ

へ

ほ

ま

め

も

ゆ

よ

ら

り

る

れ

ろ

わ