HP OpenVMS

OpenVMS VAX から OpenVMS I64 へのアプリケーションの移行

2007 年 1 月

このマニュアルでは，OpenVMS VAX アプリケーションを HP OpenVMS Industry Standard 64 for Integrity Servers (I64) に移行する方法について説明します。

改訂/更新情報: 新規マニュアルです。

ソフトウェア・バージョン: OpenVMS I64 Version 8.2 以降
OpenVMS VAX Version 6.1 以降

日本ヒューレット・パッカード株式会社

原典: Migrating an Application from OpenVMS VAX to OpenVMS I64

本書は，日本語 VAX DOCUMENT V 2.1を用いて作成しています。

目次

例

図

表

まえがき

本書は，OpenVMS VAX アプリケーションを OpenVMS I64 システムまたは OpenVMS の混成アーキテクチャ・クラスタに移行しようとする開発者を支援することを目的としています。

対象読者

本書は，プログラミング言語で記述されたアプリケーション・コードを移行しようと考えている経験のあるソフトウェア・エンジニアを対象としています。

本書の構成

本書の構成は以下のとおりです。

- 第 1 章では，OpenVMS と VAX アーキテクチャおよび Itanium アーキテクチャの関係について概要を示し，アプリケーションを VAX システムから I64 システムに移行する手順について説明します。この章では，特に次のことについて説明します。
  - OpenVMS I64 と OpenVMS VAX で互換性が高い分野
  - Intel® Itanium®アーキテクチャと VAX アーキテクチャとの比較
  - 移行作業の各段階の概要
  - 2 種類の主な移行パス，つまりソース・コードの再コンパイルと VAX イメージのトランスレート
  - 弊社が提供するマイグレーション・サポート
- 第 2 章では，2 種類の移行パスの相違点について説明し，アプリケーションを移行するパスを選ぶ際に考慮しなければならない問題点について説明します。また，アプリケーションの個々の要素を分析して，移行に影響を与えるアーキテクチャ上の相違点を確認する方法と，これらの相違点を解決するための作業を評価する方法についても説明します。
- 第 3 章では，移行環境のセットアップから，移行したアプリケーションを新しい環境に統合する処理までを，実際の移行手順にもとづいて説明します。
- 第 4 章では，再コンパイルと再リンクによってアプリケーションを変換する方法の概要を説明します。

- 第 5 章では，アプリケーションが VAX のページ・サイズに依存している場合の対処方法について説明します。
- 第 6 章では，複数のプロセスによるデータ・アクセスに関して， VAX アーキテクチャが提供する同期化機能にアプリケーションが依存している場合に，それを取り扱う方法について説明します。
- 第 7 章では，アラインメントの検討事項も含めて， I64 システムでのデータ宣言について説明します。
- 第 8 章では，アプリケーションが VAX の条件処理機能に依存している場合に，それを取り扱う方法について説明します。
- 第 9 章では， OpenVMS I64 システム上のプログラミング言語 Ada， C， COBOL， Fortran，および Pascal でサポートされる新しい機能と変更された機能の概要を示します。
- 付録 A では， OpenVMS VAX から OpenVMS I64 に移行する際に，アプリケーションを評価するためのチェックリストを示します。

関連資料

アーカイブ扱いの多数の OpenVMS マニュアルでも，ポーティングに関するさまざまな情報が説明されています。これらのマニュアルへは，下記の URL の「Porting Documentation」ナビゲーション・バーからアクセスすることができます。

~~h71000.www7.hp.com/doc/~~

この中には以下のマニュアルがあります。

- 『DECmigrate for OpenVMS AXP Systems Translating Images』では， VAX Environment Software Translator (VEST) ユーティリティについて説明しています。このマニュアルはオプションのレイヤード・プロダクトである DECmigrate for OpenVMS Alpha に同梱されています。この製品は， OpenVMS VAX アプリケーションを OpenVMS Alpha システムに移行する作業を支援します。このマニュアルでは， VEST を使用して大部分のユーザ・モード VAX イメージを Alpha システムで実行できるトランスレートされたイメージに変換する方法，トランスレートされたイメージの実行時性能を向上する方法， VEST を使用して VAX イメージで Alpha と互換性のない部分を元のソース・ファイルまでさかのぼってトレースする方法，および VEST を使用してネイティブのランタイム・ライブラリとトランスレートされたランタイム・ライブラリとの間で互換性をサポートする方法についても説明します。また， VEST コマンドのすべてのリファレンス情報も示します。
- 『HP OpenVMS Migration Software for Alpha to Integrity Servers: Guide to Translating Images』では， HP OpenVMS Migration Software for Alpha to Integrity Servers (OMSAIS) を使用して OpenVMS Alpha アプリケーションを OpenVMS I64 に移行する方法について説明しています。

このドキュメントの翻訳版である『HP OpenVMS Migration Software for Alpha to Integrity Servers: **イメージ変換ガイド**』は，下記の URL から入手できます。

https://www.hpe.com/jp/openvms-migration

- 『Creating an OpenVMS AXP Step 2 Device Driver From an OpenVMS VAX Device Driver』では， OpenVMS VAX デバイス・ドライバを OpenVMS Alpha システムで実行するためのコンバートの方法について説明しています。OpenVMS VAX デバイス・ドライバをコンパイル，リンク，ロード，そして OpenVMS Alpha デバイス・ドライバとして実行するための準備に必要な変更点を示します。また， OpenVMS I64 Alpha ドライバで使用されているエントリ・ポイント，システム・ルーチン，データ構成，マクロについても説明しています。

ホワイト・ペーパー『Intel® Itanium®アーキテクチャにおける OpenVMS 浮動小数点について』では，浮動小数点データ型の OpenVMS I64 とその他のプラットフォームでの相違点について説明しています。このホワイト・ペーパーは，次の場所にあります。

https://www.hpe.com/jp/openvms-migration

HP OpenVMS Migration Software for Alpha to Integrity Servers は， OpenVMS Alpha アプリケーションを OpenVMS I64 システム用にトランスレートおよびポーティングするための一連のツールで構成されています。この移行ツールのマニュアルへのリンクなど，詳細については次の URL を参照してください。

https://www.hpe.com/jp/openvms-migration

また，本書で説明する問題の最新情報については，以下の一般的なプログラミング・マニュアルを参照してください。

- 『VAX Architecture Reference Manual』
- 『OpenVMS Guide to Upgrading Privileged-Code Applications』
- 『VAX/VMS Internals and Data Structures』
- 『OpenVMS Programming Concepts Manual』
- 『OpenVMS Programming Interfaces: Calling a System Routine』
- 『Guide to the POSIX Threads Library』
- 『OpenVMS Calling Standard』
- 『OpenVMS デバッガ説明書』
- 『OpenVMS Linker Utility Manual』
- 『OpenVMS System Analysis Tools Manual』

個々のコンパイラのマニュアルもポーティング作業で役立ちます。

HP OpenVMS 製品とサービスの詳細については，以下の Web サイトにアクセスしてください。

https://www.hpe.com/jp/openvms

本書の表記法

本書の表記法は以下のとおりです。

Ctrl/x	Ctrl/xという表記は， Ctrl キーを押しながら別のキーまたはポインティング・デバイス・ボタンを押すことを示します。
PF1 x	PF1 xという表記は， PF1 に定義されたキーを押してから，別のキーまたはポインティング・デバイス・ボタンを押すことを示します。
Return	例の中で，キー名が四角で囲まれている場合には，キーボード上でそのキーを押すことを示します。テキストの中では，キー名は四角で囲まれていません。 HTML 形式のドキュメントでは，キー名は四角ではなく，括弧で囲まれています。
. . .	例の中の水平方向の反復記号は，次のいずれかを示します。 • 文中のオプションの引数が省略されている。 • 前出の 1 つまたは複数の項目を繰り返すことができる。 • パラメータや値などの情報をさらに入力できる。
. . .	垂直方向の反復記号は，コードの例やコマンド形式の中の項目が省略されていることを示します。このように項目が省略されるのは，その項目が説明している内容にとって重要ではないからです。
( )	コマンドの形式の説明において，括弧は，複数のオプションを選択した場合に，選択したオプションを括弧で囲まなければならないことを示しています。
[ ]	コマンドの形式の説明において，大括弧で囲まれた要素は任意のオプションです。オプションをすべて選択しても，いずれか 1 つを選択しても，あるいは 1 つも選択しなくても構いません。ただし， OpenVMS ファイル指定のディレクトリ名の構文や，割り当て文の部分文字列指定の構文の中では，大括弧に囲まれた要素は省略できません。
[|]	コマンド形式の説明では，括弧内の要素を分けている垂直棒線はオプションを 1 つまたは複数選択するか，または何も選択しないことを意味します。
{ }	コマンドの形式の説明において，中括弧で囲まれた要素は必須オプションです。いずれか 1 のオプションを指定しなければなりません。
太字	太字のテキストは，新しい用語，引数，属性，条件を示しています。
italic text	イタリック体のテキストは，重要な情報を示します。また，システム・メッセージ (たとえば内部エラー*number*)，コマンド・ライン (たとえば/PRODUCER=*name*)，コマンド・パラメータ (たとえば*device-name*) などの変数を示す場合にも使用されます。

UPPERCASE TEXT	英大文字のテキストは，コマンド，ルーチン名，ファイル名，ファイル保護コード名，システム特権の短縮形を示します。
Monospace type	モノスペース・タイプの文字は，コード例および会話型の画面表示を示します。 C プログラミング言語では，テキスト中のモノスペース・タイプの文字は，キーワード，別々にコンパイルされた外部関数およびファイルの名前，構文の要約，または例に示される変数または識別子への参照などを示します。
–	コマンド形式の記述の最後，コマンド・ライン，コード・ラインにおいて，ハイフンは，要求に対する引数がその後の行に続くことを示します。
数字	特に明記しない限り，本文中の数字はすべて 10 進数です。 10 進数以外 (2 進数， 8 進数， 16 進数) は，その旨を明記してあります。

1
移行プロセスの概要

多くの場合， OpenVMS VAX から OpenVMS I64 への移行 (マイグレーション) は簡単です。アプリケーションがユーザ・モードでのみ実行され，標準的な高級言語で作成されている場合には，ほとんどの場合，ネイティブの OpenVMS I64 コンパイラを使用してそのアプリケーションを再コンパイルし，再リンクすることにより，OpenVMS I64 システムで実行可能なバージョンを作成できます。本書では，移行するアプリケーションを評価する方法，および，より複雑で特殊な場合の対処方法について説明します。

1.1 VAX システムと I64 システムの互換性

OpenVMS I64オペレーティング・システムは， OpenVMS VAX のユーザ，システム管理，およびプログラミングの環境とできるだけ互換性を維持するように設計されています。一般的なユーザとシステム管理者に対しては， OpenVMS I64 は OpenVMS VAX と同じインタフェースを備えています。プログラマに対しては，「再コンパイル，再リンク，実行」という移行のモデルにできるだけ近づけることを目的にしています。

OpenVMS VAX システムで動作しているアプリケーションの場合，ほとんどの部分は I64 システムでも変更されません。

ユーザ・インタフェース

- DIGITAL **コマンド言語** (DCL)

  DIGITAL コマンド言語 (DCL) は OpenVMS に対する標準的なユーザ・インタフェースであり， OpenVMS I64 でも変更されません。 OpenVMS VAX で使用できるすべてのコマンド，修飾子，およびレキシカル関数は OpenVMS I64 でも使用できます。

- **コマンド・プロシージャ**

  OpenVMS VAX の以前のバージョンを対象に作成されたコマンド・プロシージャは， OpenVMS I64 システムでも全く変更せずに動作します。ただし，ビルド・プロシージャなどの特定のコマンド・プロシージャは，新しいコンパイラ修飾子やリンカ・スイッチに対応できるように変更しなければならないことがあります。リンカ・オプション・ファイルも変更が必要な場合があり，特に共有イメージ (shareable image) の場合は変更が必要となります。

- DECwindows

  ウィンドウ・インタフェースである DECwindows Motif は変更されません。

- DECforms

  DECforms インタフェースは変更されません。

- **エディタ**

  2 つの標準的な OpenVMS エディタである EVE と EDT は変更されません。

システム管理インタフェース

システム管理ユーティリティはほとんど変更されません。ただし，主な例外が 1 つあります。それはデバイス構成機能で， VAX システムでは System Generation ユーティリティ (SYSGEN) で提供される機能ですが， OpenVMS I64 では System Management ユーティリティ (SYSMAN) で提供されます。

プログラミング・インタフェース

概して，システム・サービスおよびランタイム・ライブラリ (RTL) 呼び出しインタフェースは変更されません。引数の定義を変更する必要はありません。相違点がいくつかありますが，これらの相違点は，次の 2 種類に分類されます。

- 一部のシステム・サービスと RTL ルーチン (メモリ管理システムと例外処理サービスなど) は， VAX システムと OpenVMS I64 システムとでは，少し異なる方法で動作します。詳しくは『OpenVMS System Services Reference Manual』と『OpenVMS RTL Library (LIB$) Manual』を参照してください。
- 一部の RTL ルーチンは VAX アーキテクチャに密接に関係しており，OpenVMS I64 システムでは意味がありません。これらのルーチンは次のとおりです。

ルーチン名	制約事項
LIB$DECODE_FAULT	VAX 命令をデコードする
LIB$DEC_OVER	VAX プロセッサ・ステータス・ロングワード (PSL) のみに適用される
LIB$ESTABLISH	OpenVMS I64 システムでは，類似する機能をコンパイラがサポートする
LIB$FIXUP_FLT	VAX PSL のみに適用される
LIB$FLT_UNDER	VAX PSL のみに適用される
LIB$INT_OVER	VAX PSL のみに適用される
LIB$REVERT	OpenVMS I64 システムではコンパイラがサポートする
LIB$SIM_TRAP	VAX のコードに適用される
LIB$TPARSE	動作ルーチンのインタフェースの変更が必要である。LIB$TABLE_PARSE に置き換えられている

これらのサービスとルーチンを呼び出す VAX イメージの大部分は，トランスレートし， OpenVMS I64 の TIE (Translated Image Environment) のもとで実行すれば，正しく動作します。 TIE についての詳しい説明は，『DECmigrate for OpenVMS AXP Systems Translating Images』を参照してください。

データ

- ODS-2 データ・ファイルのディスク上での形式は，VAX システムと Integrity サーバ・システムとで同じです。しかし，ODS-1 ファイルは OpenVMS I64 ではサポートされません。
- レコード管理サービス (RMS) とファイル管理インタフェースは変更されていません。
- IEEE リトル・エンディアン・データ型である S 浮動小数点 (S_floating) と T 浮動小数点 (T_floating) が追加されました。
- 大部分の VAX データ型は OpenVMS I64 上のコンパイラでサポートされています。詳細は，第 2.6.2 項を参照してください。

データベース

標準的な弊社のデータベースは，VAX システムと OpenVMS I64 システムで同様に機能します。

ネットワーク・インタフェース

VAX システムと OpenVMS I64 システムはどちらも次のインタフェースをサポートします。

- インターコネクト
  - Ethernet
  - X.25
- プロトコル
  - DECnet (バージョン 8.2 のフェーズ IV, オプションの DECnet-Plus キットのフェーズ V)
  - TCP/IP
  - OSI
  - LAD/LAST
  - LAT (ローカル・エリア・トランスポート)
- 周辺装置接続
  - SCSI
  - Ethernet
  - PCI

1.2 ユーザ作成デバイス・ドライバ

OpenVMS VAX のデバイス・ドライバを直接 OpenVMS I64 にポーティングする方法はありません。弊社では，まず OpenVMS VAX デバイス・ドライバを OpenVMS Alpha にポーティングすることをお勧めします。いったんこれを行うと， OpenVMS I64 へのポーティングは簡単です。

VAX から Alpha へのドライバのポーティングについては，アーカイブ扱いのマニュアル『Creating an OpenVMS AXP Step 2 Device Driver From an OpenVMS VAX Device Driver』を参照してください。本書のまえがきの「関連資料」の項に，このマニュアルの Web 上の場所が記載されています。このデバイス・ドライバ・マニュアルで説明されている手順を実行した後で， OpenVMS Alpha V7.0 で 64 ビットのサポートが導入されたときの OpenVMS カーネルでの変更に関係する変更を，デバイス・ドライバに対してさらに行います。詳細は，『OpenVMS Guide to Upgrading Privileged-Code Applications』を参照してください。

OpenVMS VAX から OpenVMS Alpha にデバイス・ドライバをポーティングする際には，以下の推奨事項に従ってください。

- AP (R12) よりも大きな番号のレジスタを明示的に使用しないようにします。これにより， Alpha Macro-32 コードが簡単に再コンパイルできる可能性が高くなります。
- デバイス・ドライバで， PFN が 32 ビット (VAX および Alpha の制限) しかないと仮定していないことを確認してください。 OpenVMS I64 では， HP Integrity サーバでサポートされている 50 ビットの物理アドレスをサポートしています。そのため， PFN フィールドは 64 ビット PTE のうち 32 ビット以上を必要とします。 PFN についての詳細は，『OpenVMS Guide to Upgrading Privileged-Code Applications』を参照してください。
- いくつかの VAX オプション・カード用の OpenVMS VAX ドライバは，バスのサポートを明確に参照している可能性があります。たとえば， VAX Q バス・アダプタがある場合，同等の PCI アダプタを見つけてドライバを書き直す必要があります。
- 「クラス・ドライバ」または同等のドライバがある場合 (通常は/NOADAPTER スイッチ付きでロードされる)，特定のバス・オプション・カードに結びつけられていません。これは， I64 にポーティングする有力な候補となります。
- ドライバが OpenVMS バージョン 7.xとバージョン 8.xの両方のバージョンで動作する場合は，両方のバージョン用にコンパイルおよびリンクする必要があります。これは，バージョン 8.2 で内部のデータ構造体が変更されているためです。たとえば，バージョン 8.2 上でリンクされたドライバは，バージョン 7.3-2 では動作せず，逆も同様です。

多くの OpenVMS VAX デバイス・ドライバは，VAX Macro-32 で作成されています。そのため，ドライバがそれほど大きくなければ，全体または一部を C で書き直すことも検討してください。

新しい OpenVMS Alpha デバイス・ドライバを作成する方法についての詳細は，『Writing OpenVMS Alpha Device Drivers in C』を参照してください。

1.3 移行プロセス

VAX プログラムを I64 システムで実行できるように変換するプロセスは，以下の段階に分類されます。

1. 移行するコードを評価します。
   - アプリケーションのモジュールとその環境を確認します。他のプログラムに依存する部分があるかどうかも確認します。
   - 各モジュールのコードを調べ，移行にとって障害となる部分があるかどうかを確認します。
   - アプリケーションの各部分を I64 システムに移行するための最適な方法を判断します。
2. 移行計画を作成します。
3. 移行環境を設定します。
4. アプリケーションを移行します。
5. 移行したアプリケーションをデバッグし，テストします。
6. 移行したソフトウェアをソフトウェア・システムに統合します。

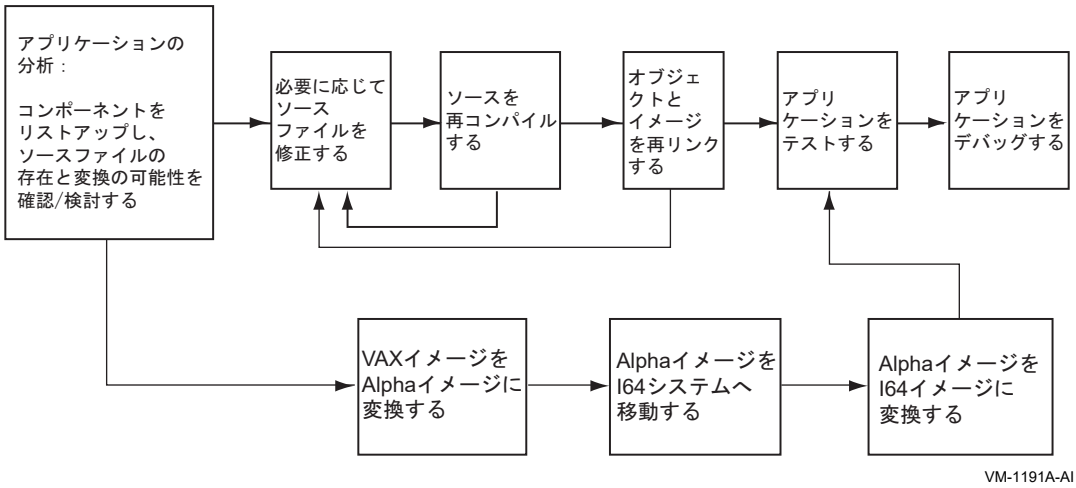
アプリケーションを OpenVMS I64 に移行するのに役立つように，多くのツールと弊社によるサービスが提供されます。これらのツールについては，本書で実際のプロセスを説明するときに示します。移行サービスについては，第 1.5 節で概要を説明します。

1.4 移行の手段

I64 システムで実行するためにプログラムを変換する方法としては，次の 2 種類の方法があります。

- 再コンパイルと再リンクを行い，ネイティブ I64 イメージを作成します。
- トランスレーションにより，ネイティブ I64 イメージを作成します。一部のルーチンは，TIE のもとでエミュレートされます。

図 1–1 VAX アプリケーションを I64 システムに移行する方法



これらの 2 種類の方法は，図 1–1 に示すとおりです。第 2.2 節では，移行方式を選択するときに考慮しなければならない事柄を説明しています。

再コンパイルと再リンク

プログラムを OpenVMS VAX から OpenVMS I64 に変換するための，もっとも効果的な方法は，ネイティブ I64 コンパイラ (C や Fortran など) を使用してソース・コードを再コンパイルし，その後， OpenVMS リンカを使用して，作成されたオブジェクト・ファイルと必要な他の共有イメージを再リンクする方法です。この方法では，Integrity システムのスピードを完全に活用できる，ネイティブ I64 イメージが作成されます。

トランスレーション

VAX から I64 にイメージをトランスレートするには，まず VEST ユーティリティを使用して Alpha にトランスレートし，その後 Alpha イメージを I64 にトランスレートする必要があります。

トランスレーション・プロセスでは， VAX との高い互換性を維持できますが，トランスレートされたイメージは，再コンパイルされたイメージほど高い性能を実現できないため，トランスレーションは，再コンパイルが不可能な場合や実用的でない場合の代替手段として使用してください。たとえば，以下の状況ではトランスレーションが適しています。

- ターゲット・システムで適切なコンパイを使用できない場合
- ソース・ファイルを入手できない場合

詳細は，第 2.4 節を参照してください。

1.5 弊社から提供される移行サポート

弊社では，アプリケーションを OpenVMS I64 に移行するためのさまざまなサービスを提供しています。

弊社では，お客様のニーズに合わせてサービスのレベルをカスタマイズします。 VAX から Integrity への利用可能な移行サービスには，以下のものがあります。

- 移行評価
- アプリケーション移行の詳細分析と設計
- システム移行の詳細分析と設計
- アプリケーションの移行
- システムの移行

お客様に適したサービスを判断するには，弊社のサポート担当または正規の販売代理店にお問い合わせください。

1.5.1 移行評価サービス

移行評価サービスでは， Integrity プラットフォームに移行する VAX システムとアプリケーション環境を評価します。移行の目的が確認され，完全な現在の構成の状態が完成します。希望する最終状態が決定され，リスクと制約事項を明確にします。最後に，いくつかの移行シナリオが作成されます。

1.5.2 アプリケーション移行の詳細分析と設計サービス

アプリケーション移行の詳細分析と設計サービスでは，自社開発したアプリケーションを詳細に分析し，すべてのモジュールについて VAX に依存しているかどうかのレポートを作成し，アプリケーションを I64 に移行するにあたって必要な変更に関する助言を行います。性能と機能に関する受け入れ条件が規定されます。

1.5.3 システム移行の詳細分析および設計サービス

システム移行の詳細分析および設計サービスでは，現在のシステム環境を詳細に分析します。これには，ハードウェア，ソフトウェア (自社開発アプリケーションを除く，商用および非商用ソフトウェア)，およびネットワーク・コンポーネントが含まれます。最適なツールと移行方法が決定され，現在の状態から将来の状態にいたる手順を反映したプロジェクト計画が作成されます。

1.5.4 アプリケーション移行サービス

アプリケーション移行サービスでは，自社開発アプリケーションを VAX プラットフォームから Integrity プラットフォームに移行します。各コード・モジュールは，ソース・コードが入手できるかどうかによって再コンパイルまたはトランスレートされます。まず， VAX への依存が取り除かれます。次に，アプリケーション全体が再リンクされ， Integrity プラットフォーム上でテストされます。次に，アプリケーションをターゲット・システムに展開します。

1.5.5 システム移行サービス

システム移行サービスでは， OpenVMS システム (単一ノードまたはクラスタ) を VAX プラットフォームから Integrity プラットフォームに移行します。顧客のシステムの使用可否と性能要件が確認され，受け入れテストの方法と条件が決定されます。

2
移行方法の選択

アプリケーションの評価を行えば，どのような作業が必要であるかを判断でき，移行計画を作成することができます。

評価のプロセスは，次の 3 つの段階に分けることができます。

- 一般的なモジュールの確認と，他のソフトウェアに依存する部分の確認
- 移行に影響するコーディングの様式を判断するためのソース分析
- 移行方式の選択，つまり，ソース・コードから再構築するのか，トランスレートするのかの選択

これらの各段階の評価を終了すれば，効果的な移行計画を作成するのに必要な情報を得ることができます。

2.1 移行のための棚卸し

移行のためのアプリケーションの評価の第一段階は，何を移行しなければならないかを正確に判断することです。移行対象には，アプリケーション自体だけでなく，アプリケーションを正しく実行するために必要なすべてのものが含まれます。アプリケーションの評価を開始するにあたり，以下のことを確認してください。

- アプリケーションの各モジュール
  - メイン・プログラムのソース・モジュール
  - 共有イメージ
  - オブジェクト・モジュール
  - ライブラリ (オブジェクト・モジュール，共有イメージ，テキスト，またはマクロ)
  - データ・ファイルとデータベース
  - メッセージ・ファイル
  - CLD ファイル
  - DECwindows サポートのための UIL ファイルと UID ファイル
- アプリケーションが依存する他のソフトウェア (例)
  - ランタイム・ライブラリ
  - 弊社のレイヤード・プロダクト

  - 他社製品

    他のコードへの依存関係を調べる場合には，VEST と/DEPENDENCY 修飾子を使用してください。VEST/DEPENDENCY コマンドは，アプリケーションが依存している実行イメージや共有イメージを示します。たとえば，ランタイム・ライブラリやシステム・サービス，その他のアプリケーションを識別します。VEST/DEPENDENCY の使い方についての詳しい説明は，『DECmigrate for OpenVMS AXP Systems Translating Images』を参照してください。

- 必要なオペレーティング環境
  - システム属性

    アプリケーションを実行および管理するために，どのような種類のシステムが必要か。たとえば，必要なメモリ・サイズ，必要なディスク領域などです。

  - ビルド・プロシージャ

    このプロシージャには，コード管理システム (CMS) やモジュール管理システム (MMS) などのツールが含まれます。

  - テスト・スイート

    テスト・スイートを使用することで，移行したアプリケーションが正しく動作するかどうかを確認し，その性能を評価することができます。

以下のように，これらの多くは，すでに OpenVMS I64 に移行されています。

- OpenVMS とともに提供される弊社のソフトウェア
  - RTL (ランタイム・ライブラリ)
  - 他の共有ライブラリ。たとえば，呼び出し可能なユーティリティ・ルーチンやアプリケーション・ライブラリ・ルーチンを提供するライブラリなど。
- 弊社のレイヤード・プロダクト
  - コンパイラとコンパイラ RTL
  - データベース・マネージャ
  - ネットワーク環境
- 他社製品

  現在多くの他社製アプリケーションが，OpenVMS I64 で実行可能です。アプリケーションが移行されているかどうかについては，各アプリケーション・ベンダーにお問い合わせください。

ビルド・プロシージャとテスト・スイートも含めて，アプリケーションと開発環境を移行する作業は，お客様が実行しなければなりません。

2.2 移行方法の選択

アプリケーションのモジュールを調査した後，アプリケーションの各部分を移行する方法を決定しなければなりません。つまり，再コンパイルと再リンクを実行するのか，トランスレートするのかを判断しなければなりません。大部分のアプリケーションは再コンパイルし，再リンクするだけで移行できます。アプリケーションがユーザ・モードだけで実行され，標準的な高級言語で作成されている場合には，おそらく再コンパイルと再リンクだけで十分です。主な例外については，第 2.5 節を参照してください。

この章では，移行のために追加作業が必要となる一部のアプリケーションで移行方法をどのように選択すればよいかについて説明します。この判断を下すには，アプリケーションの各部分でどの方法が可能であるかということと，各方法でどれだけの作業量が必要になるかということを知っておかなければなりません。

注意

以降のプロセスでは，可能なかぎりアプリケーションを再コンパイルすることとし，再コンパイルできない部分や，移行の過程で一時的な対処を目的にトランスレーションを用いるものと仮定しています。

以下の手順を実行して，アプリケーションの移行方法を選択してください。

1. 2 種類の移行方法のどちらを使用できるかを判断する

   ほとんどの場合，プログラムを再コンパイルおよび再リンクすることも，VAX イメージをトランスレートすることもできます。第 2.3 節では，どちらか一方の移行方法だけしか使用できないケースについて説明します。

2. 再コンパイルに影響を与えるアーキテクチャへの依存部分を識別する

   アプリケーションが全般的には再コンパイルに適している場合でも，Intel Itanium アーキテクチャと互換性のない VAX アーキテクチャの機能に依存するコードが含まれている可能性があります。

   第 2.5 節では，これらの依存性について説明し，このような依存部分を識別し，その問題を解決するのに必要な作業の種類と作業量を見積もるのに必要な情報を示します。

   第 2.7 節では，アプリケーションの評価で発生した疑問点に答えるのに役立つツールと方法を説明します。

3. 再コンパイルするのか，トランスレートするのかを決定する

   アプリケーションを評価した後，使用する移行方法を決定する必要があります。第 2.4 節では，各方法の利点と欠点のバランスを取ることにより，どちらの方法を使用するのかを判断する処理について説明します。

プログラムの再コンパイルおよび再リンクができない場合や，VAX イメージで VAX アーキテクチャ固有の機能を使用している場合には，そのイメージをトランスレートすることになります。第 2.4.1 項では，トランスレートしたイメージの互換性と性能を向上させるための方法について説明します。

図 2–1 プログラムの移行

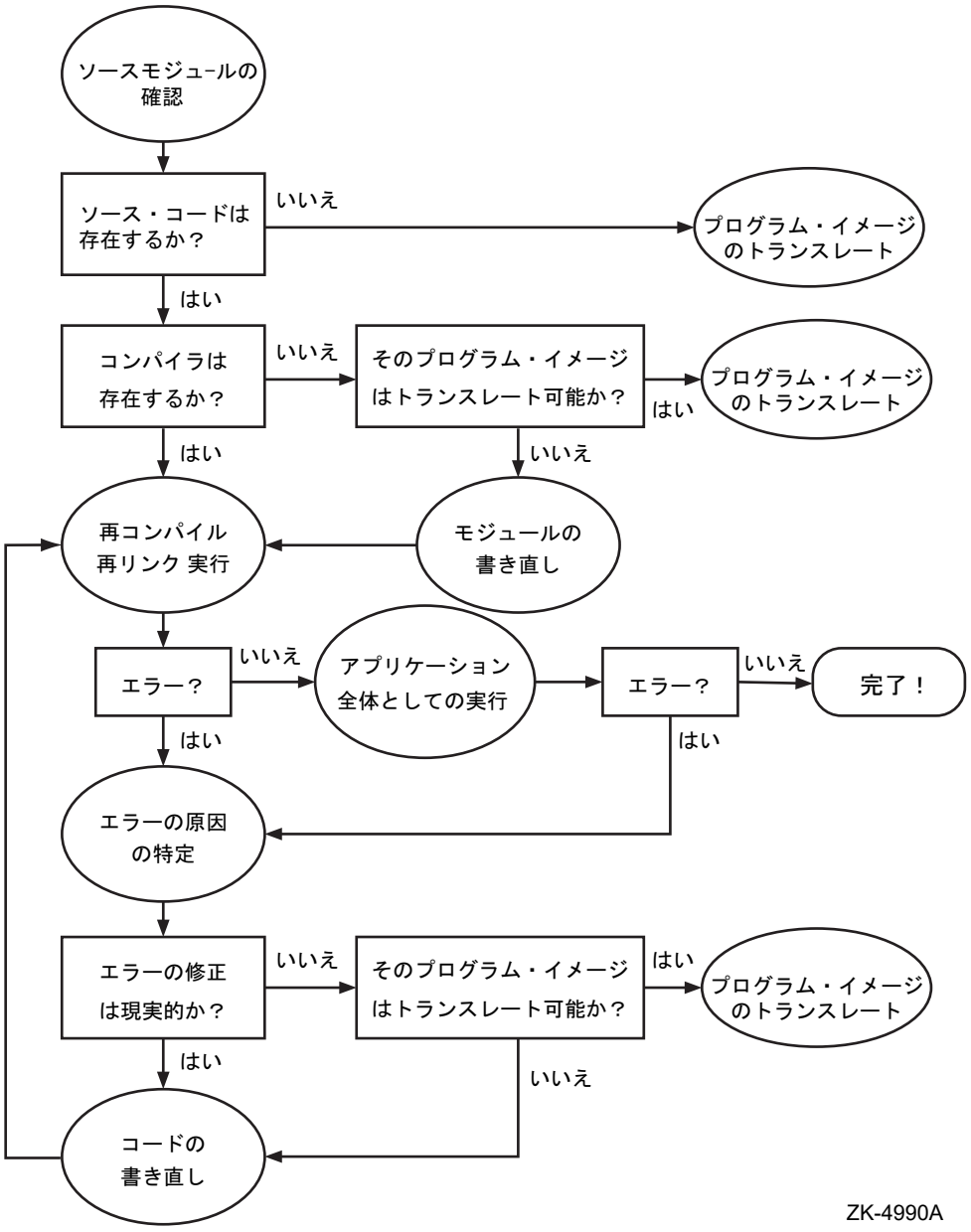


図 2–1 に示すように，評価のプロセスは，一連の質問と，これらの質問に答えるのに役立つ作業で構成されています。弊社では，これらの質問に答えるのに役立ついくつ

かのツールを提供しています。これらのツールについては，移行プロセスの適切な時点で説明します。

2.3 どの移行方法が可能か？

ほとんど場合，アプリケーションを再コンパイルおよび再リンクすることも，トランスレートすることもできます。しかし，アプリケーションの設計によっては，以下に示すように，2 種類の移行方法のどちらか一方だけしか使用できないことがあります。

- 再コンパイルできないプログラム

  以下のイメージはトランスレートする必要があります。

  – I64 コンパイラがまだ提供されていないプログラミング言語で作成されたソフトウェア

  – ソース・コードを入手できない実行イメージと共有イメージ

  – H 浮動小数点または 56 ビットの D 浮動小数点データを必要とするプログラム

- トランスレートできないイメージ

  以下のイメージでは，ソース・コードを再コンパイルおよび再リンクする必要があります (変更が必要となる可能性もあります)。

  – OpenVMS VAX バージョン 4.0 以前に作成されたイメージ

  – OpenVMS VAX バージョン 5.5 以降を使って作成されたイメージ。トランスレートされた RTL とシステム・サービスがそれ以降更新されていないため

  – Ada で作成されたイメージ

  – Ada で作成されたイメージを呼び出す/から呼び出されるイメージ

  – PDP–11互換モードを使用するイメージ

  – ベースを指定したイメージ

  – VAX アーキテクチャ用のコーディング方式を使用しているイメージ。このようなコードとしては，以下のコードがあります。

    – 内部アクセス・モードまたは高い IPL で実行されるコード (たとえば，VAX デバイス・ドライバなど)

    – システム空間のアドレスを直接参照するコード

    – 文書化されていないシステム・サービスを直接参照するコード

    – スレッド・コードを使用するコード，たとえば，スタックを切り換えるコード

    – VAX ベクタ命令を使用するコード

    – 特権付き VAX 命令を使用するコード

- 戻りアドレスを調べたり変更するコードや，プログラム・カウンタ (PC) を元に判断を下すコード
- 512 バイトのサイズのメモリ・ページに依存するため，アクセス違反動作の詳細に依存するコード
- マシンのページ境界以外の境界にグローバル・セクションをアラインするコード (たとえば， 512 バイトのメモリ・ページ・サイズに依存するコード)
- VAX P0 空間または P1 空間の大部分を使用するコード，トランスレートされたイメージの実行時サポート・ルーチンが使用する空間に敏感に反応するコード

VEST は以下のようなイメージもトランスレートできますが，トランスレートされたイメージの実行時の性能は， TIE が解釈しなければならない VAX コードの量が多いために低下します。

- TIE によって実行時に作成されるコードを除き，それ自体を変更する VAX コードまたは実行時に作成される VAX コードを含むイメージ。
- 命令ストリームを調べるコードを含むイメージ。ただし， TIE が実行時にこのようなコードを解釈する場合を除きます。

どのイメージをトランスレートできるかについての詳細は，『DECmigrate for OpenVMS AXP Systems Translating Images』を参照してください。

2.4 再コンパイルするかトランスレートするかの判断

イメージに対して 2 つの方法のどちらも使用できる場合には， I64 システム上でイメージのネイティブ・バージョンとトランスレートされたバージョンを実行した場合の性能と，イメージのトランスレートおよびネイティブ I64 イメージに変換するのに必要な作業を見積もり，これらのバランスを取る必要があります。

一般に，アプリケーションを構成する各イメージは異なるモードで実行できます。たとえば，ネイティブ I64 イメージからトランスレートされた共有イメージを呼び出したり，その逆に呼び出すことが可能です。 2 つのアーキテクチャが混在したアプリケーションについての詳細は，第 2.4.2 項を参照してください。

表 2–1 では， 2 種類の移行方法を比較しています。

表 2–1 移行方法の比較

要素	再コンパイル/再リンク	トランスレート
性能	完全な I64 の機能	通常，ネイティブ I64 の 25 ～ 40%の性能; VAX での性能と同等
必要な作業	容易な場合も困難な場合もある	容易
スケジュールの制約	ネイティブ・コンパイラが提供されるかどうかに応じて異なる	なし。ただちに可能
サポートされるプログラム		
– バージョン	VAX VMS バージョン 4.0 以前のソースが受け付けられる	VAX/VMSバージョン 5.5 またはそれ以降のイメージのみがサポートされる
– 制約事項	特権コードがサポートされる	ユーザ・モードのコードだけがサポートされる
VAX との互換性	高い。ほとんどのコードは問題なく再コンパイル/再リンクできる	エミュレーションによって実現される
今後のサポートと保守	通常のソース・コードの保守	VAX のソース・コードの保守。各新バージョンの再コンパイルと再トランスレートがある

アプリケーションをどのような方法で移行するかを決定するには，以下の事項を考慮してください。

- アプリケーションをソース・コードから完全に再構築しますか，それとも一部の機能に対してはバイナリ・イメージを使用しますか？

  バイナリ・イメージを使用する場合には，それらをトランスレートしなければなりません。

- アプリケーションを構成するすべてのイメージのソース・コードを入手できますか？

  ソース・コードを入手できないイメージは，トランスレートしなければなりません。

- どのイメージがアプリケーションの性能に大きな影響を与えますか？

  I64 システムの速度を完全に利用したいイメージでは，再コンパイルしなければなりません。

  – 性能に大きな影響を与えるイメージを識別するには， PCA を使用します。

  – ネイティブ I64 コンパイラが作成するイメージでのみ， I64 の処理機能を効率よく使用し，最適な性能を実現できます。トランスレートされた VAX イメージはネイティブ I64 コードの 1/3 程度の速度，またはそれ以下の速度で実行されます。実際の速度は，使用するトランスレーション・オプションに応じて異なります。

- 2 つの方法を使用した場合，各イメージを変換するのに要する作業量はどの程度ですか？
  - VAX イメージを I64 イメージにトランスレートするには， 2 つの手順からなる処理を実行します。
    1. VEST ユーティリティを使用して， VAX イメージを Alpha イメージにトランスレートします。
    2. HP OpenVMS Migration Software for Alpha to Integrity Servers (OMSAI) ユーティリティを使用して， Alpha イメージを I64 イメージにトランスレートします。
  - VAX アーキテクチャ固有の機能に依存するコードや， VAX の呼び出し規則に依存しているコードは直接再コンパイルすることはできません。このようなコードはトランスレーションして実行するか，コードを変更して再コンパイルおよび再リンクしなければなりません。

アーキテクチャへの依存は，以下のようにいくつかの方法で取り除くことができます。

- アーキテクチャに依存するコード・シーケンスは，プラットフォームから独立した方法で，同じ操作をサポートする高級言語のレキシカル要素に置き換えます。
- プロセッサ・アーキテクチャに適した方法で作業を実行するために， OpenVMS システム・サービスに対する呼び出しを使用します。
- ソース・コードの変更量をできるだけ少なくし，正しくプログラムが動作することを保証するために，高級言語のコンパイラ・スイッチを使用します。

表 2–2 は，各プログラムのアーキテクチャに依存する部分が，プログラムを I64 システムに移行するために使用する方法の選択に，どのような影響を与えるかを示しています。詳細は，次の章以降を参照してください。

表 2–2 移行方式の選択: アーキテクチャに依存する部分の取り扱い

再コンパイル/再リンクした VAX ソース	トランスレートされた VAX イメージ
データ・アラインメント[1]	
デフォルトでは，大部分のコンパイラはデータを自然な境界にアラインする。 VAX アラインメントを維持するための修飾子についての説明は，第 9 章を参照。	アラインされていないデータもサポートされるが，/OPTIMIZE=ALIGNMENT 修飾子を使用すれば，データがロングワードにアラインされていることを仮定することにより，実行速度を向上できる。

[1]アラインされていないデータは性能上の問題となります。アラインされていないデータを参照した場合，VAX システムでは性能がある程度低下するだけですが， I64 システムでアラインされていないデータをメモリからロードしたり，アラインされていないデータをメモリに格納すると，アラインされた操作の場合より最高 1000 倍も時間がかかる可能性があります。

(次ページに続く)

表 2–2 (続き) 移行方式の選択: アーキテクチャに依存する部分の取り扱い

再コンパイル/再リンクした VAX ソース	トランスレートされた VAX イメージ
データ型	
H 浮動小数点は X 浮動小数点に変更する。 D 浮動小数点の場合， D53 形式の 10 進 15 桁の精度で十分なときは， D 浮動小数点を G 浮動小数点に変更する。アプリケーションで 10 進 16 桁の精度 (D56 形式) が必要な場合には，トランスレートしなければならない。 COBOL のパック 10 進数は演算のために自動的にバイナリ形式に変換される。 データ型についての詳細は，第 7 章を参照。	D 浮動小数点の 10 進 16 桁の精度が必要な場合には，/FLOAT=D56_FLOAT 修飾子を使用する。この修飾子を使用した場合，性能は，デフォルトの/FLOAT=D53_FLOAT を使用した場合より低下する。
読み取り/変更/書き込み操作の不可分性	
サポートは各コンパイラが提供しているオプションに応じて異なる (詳細は第 6 章を参照)。	/PRESERVE=INSTRUCTION_ATOMICITY 修飾子を使用する。実行速度は半分に低下する。
ページ・サイズ	
デフォルトでは， OpenVMS リンカは大きい I64 スタイルのページを作成する。	512 バイト・ページのイメージの大部分はサポートされる。しかし， VEST はゆるやかな保護を割り当てるため，アクセス違反を生成するために厳しい保護に依存しているイメージをトランスレートした場合， I64 システムで正しく実行されない。
読み取り/書き込みの順序	
適切な同期命令 (MF) をソース・コードに追加することによりサポートされるが，性能は低下する (詳細は第 6 章を参照)。	/PRESERVE=READ_WRITE_ORDERING 修飾子を使用する。
VAX アーキテクチャおよび呼び出し規則への明示的な依存[2]	
サポートされない。依存する部分は削除しなければならない。	サポートされる。

[2]VAX アーキテクチャ固有の機能や呼び出し規則への依存としては， VAX 呼び出し規則， VAX 例外処理， VAX AST パラメータ・リスト， VAX 命令の形式と動作，および VAX 命令の実行時作成への明示的な依存などがある。

2.4.1 アプリケーションのトランスレート

アプリケーションを再コンパイルできない場合や， VAX アーキテクチャ固有の機能をアプリケーションで使用している場合には，アプリケーションをトランスレートすることができます。アプリケーションの一部だけをトランスレートすることもでき，

段階的に移行するための手段として一時的にアプリケーションの各部分をトランスレートすることもできます。

再コンパイルに影響を与える多くの相違点について第 2.5 節で説明していますが，これらの相違点は，トランスレートされた VAX イメージの性能にも影響を与える可能性があります。詳細は，『DECmigrate for OpenVMS AXP Systems Translating Images』および『HP OpenVMS Migration Software for Alpha to Integrity Servers: **イメージ変換ガイド**』を参照してください。

表 2–2 では，トランスレートされたイメージにおけるさまざまなアーキテクチャ依存の考慮事項の概要を示しています。

2.4.2 ネイティブ・イメージとトランスレートされたイメージの混在

一般に，I64 システムではネイティブ I64 イメージとトランスレートされたイメージを組み合わせて使用できます。たとえば，ネイティブ I64 イメージからトランスレートされた共有イメージを呼び出したり，その逆の呼び出しを実行できます。

ネイティブ・イメージとトランスレートされたイメージを組み合わせて実行するには，各イメージが VAX プラットフォームと Integrity プラットフォームの呼び出し規則の差異を吸収する必要があります。

- ネイティブ I64 イメージのルーチン・インタフェース・セマンティックとデータ・アラインメント規則は，VAX イメージと同じです。
- すべてのエントリ・ポイントは CALL*x*です。つまり，外部 JSB エントリ・ポイントは存在しません。高級言語で作成されたコードの場合には，多くの場合これが該当します。
- ネイティブ・イメージでのインバウンド呼び出しとアウトバウンド呼び出しは Ada では書かれていません。

トランスレートされたイメージがネイティブ・イメージを呼び出す場合や，その逆の呼び出しを行う場合は，**ジャケット・ルーチン**を使用して間接的に呼び出します。ジャケット・ルーチンはプロシージャの呼び出しフレームと引数リストを解釈し，呼び出し先プロシージャでの対応する呼び出しフレームと引数リストを作成し，呼び出し先プロシージャに制御を渡します。呼び出し先プロシージャが戻ると，ジャケット・ルーチンを通じて，制御が呼び出し元に戻ります。ジャケット・ルーチンは呼び出し先ルーチンが返したレジスタの値を呼び出し元ルーチンのレジスタに書き込み，呼び出し元プロシージャに制御を戻します。

OpenVMS I64 オペレーティング・システムは，ほとんどの呼び出しに対してジャケット・ルーチンを自動的に作成します。自動的なジャケット機能を使用するには，アプリケーションのネイティブ I64 部分を作成するときに，コンパイラ修飾子/TIE とリンカ・オプション/NONATIVE_ONLY を使用します。

場合によっては，アプリケーション・プログラムは特別に作成したジャケット・ルーチンを使用しなければなりません。たとえば，以下のようなライブラリに対する非標準呼び出しの場合には，ジャケット・ルーチンを作成しなければなりません。

- 外部 JSB エントリ・ポイントを含む VAX 共有ライブラリ
- 転送ベクタに読み取り/書き込みデータを含むライブラリ
- VAX 固有の関数が格納されているライブラリ
- ライブラリのネイティブ・バージョンとトランスレートされたバージョンの間で共用しなければならない資源を使用するライブラリ
- VAX イメージが提供していたすべてのシンボルを提供またはエクスポートしないネイティブ I64 ライブラリ

  (エクスポートという用語は，ルーチンがイメージのグローバル・シンボル・テーブル (GST) に登録されていることを意味します。)

これらの状況でジャケット・イメージを作成する方法については，『DECmigrate for OpenVMS AXP Systems Translating Images』を参照してください。

OpenVMS I64オペレーティング・システムに付属しているトランスレートされた共有イメージ (たとえば，ネイティブ I64 コンパイラのない言語のランタイム・ライブラリなど) には，ジャケット・ルーチンが添付されており，ネイティブ I64 イメージから呼び出すことができるようになっています。

2.5 再コンパイルに影響を与えるコーディングの様式

多くのアプリケーション，特に標準のコーディング様式のみを使用しているアプリケーションや，移植性を念頭において作成されているアプリケーションはほとんど問題なく， OpenVMS VAX から OpenVMS I64 に移行できます。しかし， VAX 固有の機能に依存し，その機能が Intel Itanium アーキテクチャと互換性のないようなアプリケーションを再コンパイルする場合には，ソース・コードを変更しなければなりません。次の例は，典型的な互換性のない機能を示しています。

- VAX システムで高い性能を実現したり， VAX アーキテクチャ固有の機能を利用するために使用されているVAX MACROアセンブリ言語
- 特権コード
- VAX アーキテクチャ固有の機能

これらの互換性のない機能がアプリケーションで全く使用されていない場合には，この章のこの後の部分を読む必要はありません。

2.5.1 VAX MACROアセンブリ言語

I64 システムでは，VAX MACROはアセンブリ言語ではなく，コンパイル型言語の 1 つでしかありません。しかし，高級言語のための I64 コンパイラと異なり，VAX MACRO–32 Compiler for OpenVMS I64 は常に高度に最適化されたコードを生成するわけではありません。したがって，VAX MACRO–32 Compiler for OpenVMS I64 は移行の補助手段としてのみ使用するようにし，新しいコードを作成する場合は使用しないでください。

VAX システムでアセンブリ言語を使用しなければならなかった多くの理由は，次のように， I64 システムでは解消されました。

- EPIC プロセッサでは，アセンブリ言語を使用しても性能が向上するわけではありません。 I64 コンパイラ・セットなどに含まれている EPIC コンパイラは，アーキテクチャや実装に固有の機能を利用した最適化されたコードを，プログラマが手作業で最適化するよりも簡単かつ効率よく生成することができます。
- 新しいシステム・サービスにより，これまでアセンブリ言語を必要としていたいくつかの機能を実行することができます。

MACRO コードの移行についての詳細は，『OpenVMS MACRO-32 Porting and User's Guide』を参照してください。

2.5.2 特権コード

内部アクセス・モード (カーネル・モード，エグゼクティブ・モード，またはスーパバイザ・モード) で実行されたり，システム空間を参照する VAX コードは， VAX アーキテクチャに依存したコーディング様式を使用している可能性が高く，また，OpenVMS I64 には存在しない VAX データ・セルを参照している可能性があります。このようなコードは，変更しなければ I64 システムに移行できません。これらのプログラムは再コーディング，再コンパイル，および再リンクが必要です。

この種類に分類されるコードは次のとおりです。

- ユーザ作成システム・サービスや他の特権付き共有イメージ

  詳細は，『OpenVMS Programming Concepts Manual』および『OpenVMS Linker Utility Manual』を参照してください。
- 弊社以外から提供されたデバイス・ドライバと性能モニタ
- 特殊な特権を使用するコード。たとえば，$CMEXEC または$CMKRNL システム・サービスを使用するコードや， PFNMAP オプション付きで$CRMPSC システム・サービスを使用するコード

  メモリ・マッピングについての詳細は，第 5 章を参照してください。

- 次のように， OpenVMS の内部ルーチンまたはデータを使用するコード
  - システム・アドレス空間をアクセスするためにシステム・シンボル・テーブル (SYS.STB) に対してリンクするコード
  - SYS$LIBRARY:LIB を用いてコンパイルするコード

OpenVMS エグゼクティブを参照する内部モード・コードを移行する場合には，弊社のサービス窓口にご連絡ください。

2.5.3 VAX アーキテクチャ固有の機能

高い性能を実現するために， Intel Itanium アーキテクチャは VAX アーキテクチャと大きく異なっています。したがって， VAX アーキテクチャ固有の機能を利用してコードを作成することに慣れているソフトウェア開発者は， I64 システムに正しく移行するために，コードで使用しているアーキテクチャ固有の機能について理解しておかなければなりません。

この後の節では，一般的なアーキテクチャ固有の機能と，それらの機能を識別する方法および対処方法について簡単に説明します。これらのアーキテクチャ固有の機能の識別方法と対処方法についての詳しい説明は，第 4 章から第 8 章までを参照してください。

2.6 アプリケーションで VAX アーキテクチャに依存する部分の識別

ネイティブ I64 コードを生成するコンパイラを使用して，アプリケーションを正しく再コンパイルできる場合でも，アプリケーションには VAX アーキテクチャに依存する部分が含まれている可能性があります。 OpenVMS I64 オペレーティング・システムは OpenVMS VAX と高い互換性を維持するように設計されています。しかし，VAX アーキテクチャと Intel Itanium アーキテクチャには基本的な違いがあるため，特定の VAX アーキテクチャ機能に依存するアプリケーションでは，問題が発生する可能性があります。ここでは，アプリケーションで特に確認しなければならない分野を示しています。

2.6.1 データ・アラインメント

データ・アドレスがデータ・サイズ (バイト数) の整数倍である場合には，データは**自然なアラインメント**になります。たとえば，ロングワードは 4 の倍数であるアドレスで自然なアラインメントになり，クォドワードは 8 の倍数であるアドレスで自然なアラインメントになります。構造体の場合も，すべてのメンバが自然なアラインメントになっているときは，その構造体も自然なアラインメントになります。

メモリ内で自然なアラインメントでないデータにアクセスすると，VAX システムでも I64 システムでも性能が大幅に低下します。VAX システムでは，大部分の言語で，デフォルトの設定によりデータはバイト境界にアラインされます。これは，VAX アーキテクチャではアラインされていないデータを参照したときに，性能の低下を最低限に抑えるためのハードウェア・サポートが準備されているためです。しかし，I64 システムでは，性能を良くするために，各データを自然なアラインメントにすることがデフォルトの設定です。この結果，Intel Itanium システムで自然なアラインメントになっているデータを参照する操作は，アラインされていないデータを参照する操作より 10 ～ 1000 倍も速くなります。

I64 コンパイラは，アラインメントに関する大部分の問題を自動的に修正し，修正できない問題には警告を発します。

問題の検出

アラインされていないデータを検出する方法には，以下のものがあります。

- 大部分の I64 コンパイラが備えている修飾子を使用する方法。この修飾子を使用すれば，コンパイラは，コンパイル時における，アラインされていないデータへの参照を報告できます。たとえば，HP Fortran プログラムの場合には，/WARNING=ALIGNMENT 修飾子を使用してコンパイルします。
- アラインされていないデータを実行時に検出するために，OpenVMS デバッガ (SET BREAK/UNALIGNED コマンド) または DEC PCA (Performance and Coverage Analyzer) を使用する方法。

問題の対処方法

アラインされていないデータに対処するには，以下の方法を用います。

- 自然なアラインメントでコンパイルするか，または言語がこの機能を備えていない場合には，自然なアラインメントになるようにデータを移動します。データが確実にアラインされるように間隔を空けると，メモリ・サイズが増えるという問題があります。メモリを節約すると同時に，確実にデータを自然にアラインするための方法として，サイズの大きい変数を先に宣言する方法があります。
- 構造体内で強制的に自然なアラインメントが実現されるように，高級言語命令を使用します。たとえば HP C では，自然なアラインメントがデフォルトのオプションです。VAX Cのデフォルトのアラインメントと一致しなければならないデータ構造 (たとえばディスク上のデータ構造など) を定義するには，#PRAGMA NO_MEMBER_ALIGNMENT文を使用します。HP Fortran の場合には，ローカル変数はデフォルトで自然なアラインメントになります。レコード構造と共通ブロックのアラインメントを制御するには，/ALIGN 修飾子を使用します。
- VAX と互換性のある，アラインされていないデータ構造を生成するコンパイラ修飾子を使用する方法。これらの修飾子を使用すると，機能的には正しいものの，実行速度が遅くなる可能性のある I64 プログラムが作成されます。

注意

自然なアラインメントに変換されたソフトウェアは，同じ OpenVMS Cluster 環境内の VAX システム，またはネットワークによって接続された VAX システムでトランスレートされた他のソフトウェアと互換性がなくなる可能性があります。次の例を参照してください。

- **既存のファイル形式は，アラインされていないデータを含むレコードを定義している可能性があります。**
- **トランスレートされたイメージは，アラインされていないデータをネイティブ・イメージに渡したり，ネイティブ・イメージからそのようなデータが渡されることを必要とする可能性があります。**

このような場合には，アプリケーションのすべての部分が，同じデータ型，つまりアラインされたデータ型またはアラインされていないデータ型を要求するように変更しなければなりません。

データのアラインメントについての詳細は，第 7 章および第 8.4.2 項を参照してください。

2.6.2 浮動小数点演算

ここでは， OpenVMS VAX システムと OpenVMS I64 システムの浮動小数点演算の相違点について説明します。

VAX アーキテクチャでは， VAX の浮動小数点形式がハードウェアでサポートされています。 Intel Itanium アーキテクチャでは，単精度と倍精度の IEEE 浮動小数点形式を使用した浮動小数点演算が，ハードウェアで実現されています。

元のアプリケーションが，デフォルトの浮動小数点形式を使用して OpenVMS VAX または OpenVMS Alpha 用に作成されている場合は，コンパイラの変換機能を使用して VAX の浮動小数点形式を使用し続けるか， IEEE の浮動小数点形式を使用するようにアプリケーションを変換することで， OpenVMS I64 にポーティングすることができます。 VAX の浮動小数点形式は，以前生成したバイナリ浮動小数点データにアクセスする必要がある場合に使用します。コンパイラは，データを VAX 形式と IEEE 形式の間で変換するために必要なコードを生成します。

変換処理についての詳細は，ホワイト・ペーパー『Intel Itanium アーキテクチャにおける OpenVMS 浮動小数点演算について』を参照してください。このホワイト・ペーパーがある Web 上の場所については，本書の「まえがき」の「関連資料」を参照してください。

IEEE 浮動小数点形式は， VAX の浮動小数点形式が必要でない場合に使用します。 IEEE 浮動小数点形式を使用することで，コードの効率がより高まります。

2.6.3 データ型

Intel Itanium アーキテクチャは，VAX 固有のデータ型の大部分をサポートしています。ただし，H 浮動小数点データ型などの一部の VAX データ型は全くサポートされておらず，F 浮動小数点などの他のデータ型は，IEEE 形式に変換して目的の演算を実行することでサポートされています (表 2–3 を参照)。アプリケーションが，実際のネイティブ・データ型のサイズまたはビット表現に依存していないかどうか確認してください。

表 2–3 浮動小数点データ型のサポート

データ型	VAX	I64
G 浮動小数点	サポートされる。	コンパイル・コマンドで/FLOAT=G_FLOAT を指定して，IEEE T 浮動小数点に自動的に変換することでサポートされる。D56 浮動小数点の代わりに D53 浮動小数点を使用すると，精度が 3 ビット失われ，若干異なる結果となる。
D 浮動小数点	サポートされる。	コンパイル・コマンドで/FLOAT=D_FLOAT を指定して，IEEE T 浮動小数点に自動的に変換することでサポートされる。
F 浮動小数点	サポートされる。	コンパイル・コマンドで/FLOAT=D_FLOAT または/FLOAT=G_FLOAT を指定して，IEEE S 浮動小数点に自動的に変換することでサポートされる。
H 浮動小数点 (128 ビット浮動小数点)	サポートされる。	サポートされない。H 浮動小数点の完全なサポートは，DECmigrate を使用することで得ることができる。これを使用して，H 浮動小数点を含むコード・モジュールをトランスレートするか，サポートされているデータ型を使用してアプリケーションを作成し直すことができる。
S 浮動小数点 (IEEE)	サポートされない。	サポートされる。
T 浮動小数点 (IEEE)	サポートされない。	サポートされる。
X 浮動小数点 (128 ビット浮動小数点 (Itanium, IEEE に類似))	サポートされない。	サポートされる。X 浮動小数点データ形式は，H 浮動小数点と違うが，どちらも同じ範囲の値を扱うことができる。Fortran アプリケーションでは，X 浮動小数点メモリ形式とディスク上の H 浮動小数点の自動的な変換が可能である。それには論理名 FOR$CONVERTnnn，OPTIONS 文，コンパイラ修飾子/CONVERT，OPEN 文での CONVERT=keywordのいずれかを使用する。

Intel Itanium プロセッサでは，性能の向上のために，パック 10 進数 (packed decimal) データ型，H 浮動小数点データ型，G 浮動小数点データ型，D 浮動小数点データ型，および F 浮動小数点データ型をソフトウェアによって以下のように実現します。

- 10 進数

  18 桁の 10 進数データは 64 ビットの 2 進数に内部的に変換されます。この結果，COBOL できわめて高い性能を実現できます。

- H 浮動小数点

  I64 コンパイラは H 浮動小数点データをサポートしません。しかし， Translated Image Environment (TIE) は，トランスレートされたイメージで， H 浮動小数点データをエミュレートによってサポートします。

- D 浮動小数点， G 浮動小数点， F 浮動小数点

  それぞれの VAX 浮動小数点データ型は，要求された演算を行う前に，同等の IEEE の値に変換されます。範囲と精度が若干異なるため，結果は VAX と多少異なる可能性があります。

問題への対処方法

データ型に関する問題に対処するには，次の方法を使用します。

- 可能な場合には， D 浮動小数点， F 浮動小数点， G 浮動小数点，または H 浮動小数点の代わりに， IEEE S 浮動小数点または IEEE T 浮動小数点を使用してください。
- 可能な場合には，パック 10 進数データ型の代わりに整数データ型を使用してください。

I64 のデータ型についての詳細は，第 7 章を参照してください。特に，第 7.2 節では， VAX データ型に依存していないかどうかを確認する方法について説明しています。

浮動小数点データ型についての詳細は，ホワイト・ペーパー『Intel Itanium アーキテクチャにおける OpenVMS 浮動小数点演算について』を参照してください。このホワイト・ペーパーがある Web 上の場所については，本書の「まえがき」を参照してください。

2.6.4 データへの共用アクセス

不可分な操作とは，次のような操作です。

- 中間結果または部分的な結果を他のプロセッサや装置から参照できない
- 操作を中断できない (つまり，起動した後，操作は完全に終了するまで継続されます)

OpenVMS I64 では，データをメモリから読み取る操作，データを変更する操作，およびデータをメモリに格納する操作は，複数の命令で構成され，これらの命令の間で割り込みが発生する可能性があります。この結果，アプリケーションで共用メモリ内のデータを不可分な操作によって変更したい場合には，操作の不可分性を保証するための作業が必要になります。

以下の条件のいずれかを満たしている場合，アプリケーションは操作が不可分に実行されることに依存している可能性があります。

- プロセス内の AST ルーチンがメインライン・コードとデータを共用している

- プロセスが，同じ CPU (つまり，ユニプロセッサ・システム) で実行される別のプロセスと，**書き込み可能なグローバル・セクション**のデータを共用している
- プロセスが別の CPU (つまり，マルチプロセッサ・システム) で並列に実行される別のプロセスとの間で，書き込み可能なグローバル・セクションのデータを共用している

問題の検出
不可分性への依存を検出するには，共用変数 (複数の実行スレッドによってアクセスされる書き込み可能な項目) の使用を再確認し，不可分性を暗黙にまたは明示的に仮定している部分がないかどうかを調べます。

問題への対処方法
命令の不可分性に関する一般的な問題を解決するには，以下の方法を用います。

- 可能な場合には，共用変数を保護するために不可分性を保証する言語構造を使用してください。たとえば， C では VOLATILE 宣言を使用します。
- 不可分性を仮定するのでなく，明示的に**同期**を使用します。
- OpenVMS のロック・サービス (たとえば$ENQ と$DEQ) または LIB$ルーチンを使用します。
- AST スレッドとの同期をとるために，メインライン・コードで$SETAST システム・サービスを使用して AST をブロックし，命令が終了した後で AST を再度有効にします。

同期についての詳細は，第 6 章を参照してください。

2.6.5 ページ・サイズに関する検討

ページ・サイズは，メモリ管理ルーチンとシステム・サービスが割り当てる仮想メモリのサイズを管理します。たとえば，混在アーキテクチャの OpenVMS Cluster システムでは，アプリケーションが特定のデータ・バッファのサイズを VAX ページ・サイズに基づいて決定できます。ページ・サイズは，メモリ内のコードとデータに保護を割り当てる基準にもなります。

OpenVMS VAX オペレーティング・システムは 512 バイトの倍数でメモリを割り当てます。 I64 でのページ・サイズは，ハードウェア・プラットフォームの属性ではなく， SYSGEN パラメータの設定によって変わります。これは，実行時の設定であり，ブートするたびに設定が変わる可能性がある点に注意してください。

ページ・サイズは，ワーキング・セット・クォータなど，システム資源を管理するときの基礎的な要素です。さらに， VAX システムでのメモリ保護は， 512 バイトのメモリ領域ごとに設定できます。 I64 システムでは，メモリ保護の最小単位は，システムのページ・サイズに応じて， VAX システムの場合よりはるかに大きくなります。

注意

ページ・サイズを大きくすることで影響を受けるのは，512 バイトのページ・サイズに明示的に依存しているアプリケーションだけです。たとえば，以下のアプリケーションが考えられます。

- **次の目的で "512"を使用しているアプリケーション**
  - **メモリの使用量を計算するため**
  - **必要なページ・テーブルのサイズを計算するため**
- **512 バイト単位で保護を変更するアプリケーション**
- **システム・サービス** Create and Map Section ($CRMPSC) **を使用して，ファイルをプロセス・アドレス空間内の特定の位置にマッピングするアプリケーション (たとえば，使用可能なメモリが制限されているときにメモリを再利用するため)**

問題の検出

VAX ページ・サイズを使用している箇所を検出するには， 512 バイトのまとまりで仮想メモリを操作するコードか，メモリ保護の最小単位として 512 バイトを使用しているコードを探します。コードに記述されている 10 進数の 511， 512，または 513，16 進数の 1FF， 200，または 201 などの数値を検索してください。

問題への対処方法

VAX と I64 のページ・サイズの相違によって発生する問題に対処するには，以下のような方法を使用できます。

- ハードコードされたページ・サイズの参照をシンボル参照による値に変更します (実行時に$GETSYI を呼び出すことにより割り当てられます)。
- ページ・サイズとディスク (ファイル)・ブロック・サイズが等しいと仮定しているコードを見直します。 I64 システムでは，この仮定は正しくありません。
- メモリ管理関連システム・サービス (たとえば，$CRMPSC，$MGBLSC) を使用して，ファイルを固定のページ・サイズから計算されるアドレス範囲 (グローバル・セクション) にマッピングできるものと仮定しないでください。このような場合には， SEC$M_EXPREG フラグを使用してください。

ページ・サイズについての詳細は，第 5 章を参照してください。

2.6.6 マルチプロセッサ・システムでの読み取り/書き込み操作の順序

VAX アーキテクチャでは，マルチプロセシング・システムのプロセッサが複数のデータをメモリに書き込む場合，これらは指定した順にメモリに書き込まれます。このように書き込みの順序が定義されているため，書き込み結果はプログラムと入出力装置で指定した順に他の CPU からも確認することができます。

このように，指定した順に書き込み結果を他の CPU からも確認できることを保証することは，システムが実現できる性能の最適化を制限してしまいます。また，キャッシュがより複雑になり，キャッシュ性能の最適化も制限されます。

システム全体の性能を向上するために， Integrity サーバ・システムをはじめとする EPIC システムでは，メモリへの読み取りと書き込みの順序を変更できます。したがって，メモリへの書き込みをシステム内の他の CPU から確認すると，その順序は書き込みを依頼した順序と異なる可能性があります。

注意

この節の説明はマルチプロセッサ・システムを対象にしています。ユニプロセッサ・システムでは，メモリ・アクセスはすべて，プログラムで要求した順に終了します。

問題の検出

マルチプロセッサ・システムで実行される可能性のあるアプリケーションで，読み取り/書き込みの順序に依存している部分を見つけるには，データが書き込まれる順序に依存するアルゴリズムを識別しなければなりません。たとえば，同期をとるためにフラグ受け渡しプロトコルを使用している箇所を調べなければなりません。

問題への対処方法

読み取り/書き込み操作の順序に関する問題に対処するには，次の方法を用います。

- フラグ受け渡しプロトコルの代わりに，同期をとるためにシステムで提供されるルーチンを使用します。たとえば， OpenVMS ロック・システム・サービス ($ENQ，$DEQ) を使用します。
- Intel Itanium アーキテクチャでは，メモリ・フェンス命令が準備されており，この命令を使用すると，ハードウェアはフェンスの前のすべてのメモリ読み取りと書き込みを終了した後で，フェンスの後の読み取りと書き込みを実行します。 I64 の一部の言語では，この命令を挿入する方法が提供されていますが，この命令を使用すると性能が低下します。

同期についての詳細は，第 6 章を参照してください。

2.6.7 VAX プロシージャ呼び出し規則への明示的な依存

OpenVMS の呼び出し規則では， VAX プログラムと I64 プログラムとで，呼び出し規則が大きく異なります。 VAX プロシージャ呼び出し規則の詳細な部分に明示的に依存するアプリケーション・プログラムを， I64 システムでネイティブ・コードとして実行する場合は，変更が必要です。たとえば，以下のようなコードは変更する必要があります。

- 引数ポインタ (AP) を基準にして引数の相対位置を判断するコード

  ただし，多くの場合，VAX MACRO–32 Compiler for OpenVMS I64 はこの問題を自動的に補正します。

- スタック上の戻りアドレスを変更するコード
- 呼び出しフレームの内容を利用するコード

VAX コンパイラでも I64 コンパイラでも，プロシージャ引数にアクセスするための方法が提供されています。コードでこれらの標準メカニズムを使用している場合には，単に再コンパイルするだけで， I64 システムで正しく実行できるようになります。コードでこれらのメカニズムを使用していない場合には，これらのメカニズムを使用するように変更しなければなりません。これらの標準メカニズムについての説明は，『OpenVMS Calling Standard』を参照してください。

トランスレートされたコードは， VAX プロシージャ呼び出しの動作をエミュレートします。ここに示した依存関係を持つイメージは，トランスレートすることにより I64 システムで正しく実行できるようになります。

2.6.8 VAX の例外処理メカニズムへの明示的な依存

例外処理の方法は， VAX システムと Integrity サーバ・システムとで異なります。算術演算例外が， VAX システムと Integrity システムでディスパッチされる方法の違いについては，第 8 章を参照してください。ここでは主に，コードで動的に条件ハンドラを設定するメカニズムと，条件ハンドラが例外状態にアクセスするメカニズムについて説明します。

2.6.8.1 動的な条件ハンドラの設定

VAX システムには，アプリケーションが実行時に動的に条件ハンドラを設定する方法が数多く用意されています。 OpenVMS の呼び出し規則では，条件ハンドラのアドレスをプロシージャの呼び出しフレームの先頭に書き込むことによって，そのフレームの中で起きた例外に対処する条件ハンドラとして設定できます。これにより，VAX 上のプログラムが条件ハンドラの設定を容易に行えるようになっていますが，OpenVMS の呼び出し規則では， I64 プロシージャ用に，このような書き込み可能領域は確保されていません。

たとえば，VAX MACROプログラムは次の命令を使用して，条件ハンドラのアドレスを呼び出しフレームに移動できます。

```
MOVAB    HANDLER,(FP)
```

VAX MACRO–32 Compiler for OpenVMS I64 はこの文を解析し，条件ハンドラを設定するための適切な I64 アセンブリ言語コードを作成します。詳細は，『OpenVMS MACRO-32 Porting and User's Guide』を参照してください。

注意

VAX システムでは，ランタイム・ライブラリ・ルーチン LIB$ESTABLISH **と，その逆の操作をする** LIB$REVERT **を使用すれば，アプリケーションは現在のフレームに対して条件ハンドラを設定したり，解除することができます。これらのルーチンは，** I64 **システムには存在しません。しかし，コンパイラではこれらの呼び出しを正しく取り扱うことができます (たとえば，**

Fortran の組み込み関数によって)。詳細は，第 9 章と，アプリケーションに関連するコンパイラのマニュアルを参照してください。

トランスレートされたコードは，条件ハンドラを動的に設定するための VAX メカニズムをエミュレートします。

I64 コンパイラによっては，動的に条件ハンドラを設定するためのメカニズムを提供しているものもあります。

条件ハンドラについての詳細は，第 8 章を参照してください。

2.6.8.2 シグナル・アレイとメカニズム・アレイ内のデータのアクセス

VAX システムと I64 システムはどちらも，例外処理で例外時のプロセッサ・ステータス，例外時のプログラム・カウンタ (PC)，シグナル・アレイ，およびメカニズム・アレイをスタックに格納します。

シグナル・アレイとメカニズム・アレイの内容は， VAX システムと I64 システムとで異なります。メカニズム・アレイの形式も 2 つのプラットフォームで異なります。シグナル・アレイまたはメカニズム・アレイ内のデータにアクセスする条件ハンドラが，両方のシステムで正しく動作するためには，オフセットをハードコードするのではなく，適切な CHF$シンボルを使用しなければなりません。適切な CHF$シンボルについての詳細は，『OpenVMS Programming Concepts Manual』を参照してください。

注意

条件ハンドラは，引数ポインタ (AP) からのハードコードされたオフセットを使用して，メカニズム・アレイ内の情報を正しく見つけることができません。

2.6.9 VAX AST パラメータ・リストの変更

OpenVMS VAX は， 5 つのロングワード引数を AST サービス・ルーチンに渡します。VAX MACROで作成されている AST サービス・ルーチンは，引数ポインタ (AP) からのオフセットを使用してこの情報にアクセスします。 OpenVMS VAX では，AST サービス・ルーチンは，保存されているレジスタやリターン・プログラム・カウンタ (PC) も含めて，これらの引数を変更できます。これらの変更は， AST ルーチンが終了した後，割り込まれたプログラムに影響を与える可能性があります。

I64 システムの AST パラメータ・リストも 5 つのパラメータで構成されますが，AST プロシージャを対象にした引数は AST パラメータだけです。他の引数も存在しますが，これらの引数は， AST プロシージャが終了した後は使用されません。したがって，これらの引数を変更しても， AST 終了時に再開される操作のスレッドに影響を与えないため，このような影響に依存するプログラムを I64 システムで実行する

には，従来の引数受け渡しメカニズムを使用するようにプログラムを変更しなければなりません。

2.6.10 VAX 命令の形式と動作への明示的な依存

VAX MACRO命令の実行動作や VAX 命令のバイナリ・エンコーディングに特に依存するプログラムは， I64 システムでネイティブ・コードとして実行するために再コンパイルまたは再リンクする前に，変更しなければなりません。

たとえば，次のコーディング方式は I64 システムでは機能しません。

- VAX MACROで VAX 命令ブロックをプログラムのデータ領域に組み込み， PC を変更してこのコード・ブロックに制御を渡すようなコーディング様式
- プロセッサ状態ロングワード (PSL) の条件コードや他の情報を調べるようなコーディング様式

VAX MACROコードの移行についての詳細は，『OpenVMS MACRO-32 Porting and User's Guide』を参照してください。

2.6.11 VAX 命令の実行時作成

実行時に VAX 命令を作成し，実行する従来の方法は，ネイティブ I64 モードでは機能しません。実行時に作成される VAX 命令は，トランスレートされたイメージでエミュレーションによって実行されます。

特定の VAX 命令を実行時に作成するコードについての詳細は，『OpenVMS MACRO-32 Porting and User's Guide』を参照してください。

2.7 VAX システムと I64 システムの間で互換性が維持されない部分の識別

アプリケーションの各モジュールで，アーキテクチャ間の互換性が維持されない部分を識別するには，まず I64 コンパイラを使用して，モジュールのテスト・コンパイルを実行します。このときに役立つ診断用コンパイラ・スイッチについての説明は，各言語プロセッサのマニュアルを参照してください。

多くのモジュールはエラーなしにコンパイルし，実行できます。しかし，エラーが発生した場合には，モジュールを変更しなければなりません。

HP コンパイラは，移植に関する問題を見つけるのに役立つメッセージを出力します。たとえば， MACRO–32 コンパイラでは，/FLAG 修飾子でいくつかのオプションを指定できます。どのオプションを指定したかに応じて，アラインされていないスタックおよびメモリ参照，実行時コード生成 (自己変更コードなど)，ルーチン間の分岐，その他のさまざまな条件が報告されます。

HP Fortran コンパイラの/CHECK 修飾子は，ユーザが指定したさまざまなオプションに関するメッセージを出力します。

注意

モジュールを単独でエラーなしに実行できる場合でも，アプリケーションの他の部分と一緒に実行したときにだけ，同期に関する問題が発生する可能性があります。

再コンパイルおよび再リンクした後，モジュールを正しく実行できない場合には，以下の方法を使用して， I64 システムでプログラムを正しく実行するために，どの部分を変更しなければならないかを評価します。

- ソース・コードの確認

  移行プロセスのこの段階でコードを再確認すれば，多くの問題を前もって解決でき，この後の移行段階で多くの時間と作業を節約できます。コードを確認するには，付録 A に示したチェックリストを使用し，さらに第 4 章に示したガイドラインに従ってください。移行に関するこれらの問題点は，第 2.5 節にまとめられています。

  アプリケーション全体のコードを直接確認することが現実的でない場合には，自動的な検索処理を使用すると便利です。たとえば， DCL の SEARCH コマンドとエディタを組み合わせて使用して，アーキテクチャへの依存を検出することができます。

- 初期のテスト・コンパイルでコンパイラが生成したメッセージの確認

  コンパイラは次の情報を提供します。

  – VAX コンパイラと I64 コンパイラの違い

  – データ・アラインメント

  コンパイラによっては， VAX アーキテクチャと Intel Itanium アーキテクチャのその他の違いも検出します。

- VEST を使用したイメージの分析

  プログラムを再コンパイルおよび再リンクする場合でも，分析ツールとして VEST を使用できます。この結果， I64 システムでプログラムをもっとも効率よく実行するのに必要な変更作業に関して，役立つ多くの情報が得られます。たとえば， VEST は以下の問題を検出するのに役立ちます。

  – アラインされていない静的データ (構造体のアラインされていないフィールドも含めたデータ宣言) とアラインされていないスタック操作

  – 浮動小数点データ型の参照 (H 浮動小数点と D 浮動小数点)

  – パック 10 進数の参照

  – 特権コード

  - 標準的でないコーディング様式
    - システム・サービスを使用しない， OpenVMS データまたはコードの参照
    - 初期化されていない変数
  - 同期に関するある種の問題，たとえば，インターロック命令のマルチプロセス使用

  ただし， VEST は一部の問題を検出できません。次の例を参照してください。

  - アラインされていない変数 (動的に作成された構造体内の変数)
  - 同期に関する問題の大部分

- PCA (Performance and Coverage Analyzer) の使用によるイメージの実行

  PCA は次の問題を検出できます。

  - 実行時アラインメント・フォルト
  - アプリケーションのどの部分がもっとも頻繁に実行され，性能に大きく影響するか

アプリケーションのすべてのイメージを I64 システムでエラーなしに実行できた場合には，イメージを組み合わせて実行し，イメージ間の同期に関する問題が発生しないかどうかをテストしなければなりません。テストについての詳細は，第 3.5 節を参照してください。

3
アプリケーションの移行

アプリケーションを実際に OpenVMS I64 システムに移行する作業は，次に示すように複数の段階に分かれています。

1. 移行環境を設定する
2. 移行評価の基礎とするために VAX システムでアプリケーションをテストする
3. I64 システムで実行するためにアプリケーションを変換する
4. 移行したアプリケーションをデバッグおよびテストする
5. 移行したアプリケーションをソフトウェア・システムに統合する
6. 特定の種類のコードを変更する

3.1 移行環境の設定

ネイティブな I64 環境は， VAX システムと同様に完全な開発環境です。移行したアプリケーションのコンパイル，リンク，デバッグおよびテストは， Integrity サーバ・システム上で行わなければなりません。

I64 移行環境の重要な要素は HP からサポートされ， HP はアプリケーションの変更，デバッグ，およびテストのために必要な支援を提供します。

3.1.1 ハードウェア

移行のためにどのハードウェアが必要かを計画する場合，複数の問題を検討しなければなりません。まず，通常の VAX 開発環境でどのような資源が必要であるかを検討してください。

- CPU
- ディスク
- メモリ

I64 への移行環境にとって必要な資源を見積もるには，次の点に注意してください。

- I64 システムでは，イメージ・サイズが従来より大きくなる

  VAX システムと I64 システムとの間で，コンパイルしたイメージとトランスレートしたイメージを比較してください。
- I64 システムでは，ページ・サイズと物理メモリ・サイズが従来より大きくなる

- CPU の要件

VAX イメージを Integrity サーバ上で動作するようにトランスレートすると，一般に多くの CPU 時間が必要となります (実際に必要な CPU 時間を予測することは困難です。必要な CPU 時間は，アプリケーションのサイズよりもアプリケーションの複雑さに依存するからです)。また，イメージ・トランスレータを使用する場合，ログ・ファイルのためのディスク空間， Alpha イメージのためのディスク空間 (必要とする場合)，フローグラフのためのディスク空間などが大量に必要になります。新しいイメージには，元の VAX 命令，トランスレートされた Alpha 命令，新しい Itanium 命令が含まれます。したがって，元の VAX イメージより必ず大きくなります。

望ましい構成は次のとおりです。

- 256 MB のメモリを搭載した 6 VUP 以上のマルチプロセシング・システム
- 1 GB のシステム・ディスク
- 各アプリケーションあたり 1 GB のディスク

マルチプロセシング・システムでは，各プロセッサが別々のアプリケーションのイメージ分析を行うことができます。

コンピュータ資源が不足する場合には，次のいずれかの処置で対処してください。

- コンパイラや VEST は，作業量の少ない時刻にバッチ・ジョブとして実行してください。
- 移行作業のために必要な機器をリースしてください。

3.1.2 ソフトウェア

効率のよい移行環境を構築するには，以下の要素を確認しなければなりません。

- 移行ツール

  以下のツールも含めて，互換性のある移行ツールが必要です。

  - コンパイラ
  - トランスレーション・ツール
    - VAX イメージを Alpha にトランスレートするための VEST
    - HP OpenVMS Migration Software for Alpha to Integrity Servers (OMSAIS)
  - RTL
  - システム・ライブラリ
  - C プログラムのためのインクルード・ファイル
- コンパイル・プロシージャとリンク・プロシージャ

これらのプロシージャは新しいツールおよび新しい環境に適合するように調整しなければなりません。

- テスト・ツール

  OpenVMS VAX のテスト・ツールを OpenVMS I64 にポーティングしなければなりません。また，アプリケーションでの OpenVMS I64 固有の動作をテストするために設計されたテスト・ツールも必要です。最初に VAX から Alpha にトランスレートし，その後 I64 にトランスレートする場合は，最初の手順が正常に完了していることを確認する必要があるため， Alpha 用のテスト・ツールも必要になります。

- ソースの管理とイメージのビルドのためのツール
  - CMS (コード管理システム)
  - MMS (モジュール管理システム)

トランスレーション
ソフトウェア・トランスレータ OMSAIS は Alpha システムと I64 システムの両方で動作します。 TIE はトランスレートされたイメージを実行するために必要であり，OpenVMS I64 の一部です。したがって，トランスレートされたイメージの最終的なテストは I64 システムまたは I64 Migration Center で実行しなければなりません。

3.2 アプリケーションの変換

コードを完全に分析し，移行計画の作成を完了してある場合には，この最終段階は簡単に終了できます。多くのプログラムは全く変更せずに再コンパイルまたはトランスレートできます。直接に再コンパイルまたはトランスレートできないプログラムでも，多くの場合，簡単な変更だけで I64 システムで実行できるようになります。

コードの実際の変換についての詳しい説明は， OpenVMS I64 の移行に関する次のマニュアルを参照してください。

- 『DECmigrate for OpenVMS AXP Systems Translating Images』
- 『OpenVMS MACRO-32 Porting and User's Guide』
- 『HP OpenVMS Migration Software for Alpha to Integrity Servers: **イメージ変換ガイド**』

これらのマニュアルについての説明は，本書のまえがきを参照してください。

3.2.1 再コンパイルと再リンク

一般に，アプリケーションを移行する場合には，コードの変更，コンパイル，リンク，およびデバッグを繰り返し実行しなければなりません。これらの処理を実行することにより，開発ツールから指摘されたすべての構文エラーとロジック・エラーを解

決します。通常，構文エラーは簡単に修正できますが，ロジック・エラーを修正するには，コードの大幅な変更が必要になります。

時が経つにつれて，言語規格とコンパイラ品質の両方が変化します。コードの移植性によって，規格へのより厳密な準拠またはその他の品質向上を示すコンパイラ・メッセージが表示されることがあります。コードの移植性が高いほど，メッセージが表示される機会は減ります。

新しいコンパイラ・スイッチやリンカ・スイッチに対応するために，コンパイル・コマンドとリンク・コマンドは，ある程度変更しなければなりません。たとえば，さまざまなバージョンの OpenVMS I64 で移植可能にするために，リンカはデフォルトのページ・サイズを 64KB として設定します。これにより OpenVMS I64 イメージは，デフォルトのページ・サイズが最大 64 KB までのシステムで実行可能になります。

I64 システムでソフトウェアを開発し移行するために，多くのネイティブ・コンパイラとその他のツールが提供されています。

3.2.1.1 ネイティブ I64 コンパイラ

既存の VAX ソースを再コンパイルおよび再リンクすると，ネイティブ I64 イメージが作成されます。このイメージは， Intel Itanium アーキテクチャの性能上の利点をすべて利用して， I64 環境で実行されます。 I64 コードでは，一連の高度な最適化コンパイラを使用します。

OpenVMS I64 では，次の言語に対してネイティブ I64 コンパイラが提供されています。

- GNAT Pro Ada[1]
- BASIC
- BLISS (Freeware CD で提供)
- C
- C++
- COBOL
- Fortran
- MACRO–32
- Pascal

他の言語[1]で作成されたユーザ・モードの VAX プログラムは， VEST および OMSAI を使用してトランスレートすることにより， I64 システムで実行できます。また，他社からも他の言語のためのコンパイラが提供されます。

[1] Ada Core Technologies が販売している GNAT Pro は， OpenVMS I64 での推奨 Ada 95 コンパイラです。詳細は，第 9 章を参照してください。
[1] PL/I プログラムはトランスレートできません。

一般に，I64 コンパイラには，VAX アーキテクチャに依存するコードを少し変更するだけで I64 システムでも実行できるようにするための，コマンド・ライン修飾子と言語セマンティックが準備されています。このような VAX アーキテクチャへの依存のリストについては，表 2–2 を参照してください。

OpenVMS I64 システムの一部のコンパイラは，OpenVMS VAX システムでの対応するコンパイラでサポートされていない新しい機能をサポートしています。互換性を維持するために，一部のコンパイラは互換性モードをサポートしています。たとえば，OpenVMS I64 システム用の HP C コンパイラでは，VAX C互換性モードがサポートされており，/STANDARD=VAXC 修飾子を指定することにより起動されます。

場合によっては，互換性が制限されます。たとえば，VAX Cでは，特殊な VAX ハードウェア機能にアクセスできるようにするための組み込み関数が実装されています。VAX コンピュータのハードウェア・アーキテクチャは Integrity サーバのハードウェア・アーキテクチャと異なるため，これらの組み込み関数は，/STANDARD=VAXC 修飾子を使用した場合でも，OpenVMS I64 システム用の HP C コンパイラでは使用できません。

また，コード内に存在するアーキテクチャ依存部分をコンパイラである程度補正することもできます。たとえば，MACRO–32 コンパイラには/PRESERVE 修飾子があり，粒度と不可分性のどちらか一方または両方を維持できます。

HP C コンパイラには，各データ型の typedef を定義しているヘッダ・ファイルがあります。これらの typedef は，int64 などの汎用データ型の名前を__int64 などのマシン固有のデータ型に変換します。たとえば，64 ビットの長さのデータ型が必要な場合には，int64 typedef を使用します。

移植性をサポートするすべての機能の詳細については，コンパイラのマニュアルを参照してください。

I64 コンパイラを使用して，OpenVMS VAX プログラムを OpenVMS I64 システムに移行する処理の詳細については，第 9 章を参照してください。

3.2.1.2 OpenVMS I64 用のVAX MACRO–32 コンパイラ

既存のVAX MACROコードを OpenVMS I64 システムで動作するマシン・コードに変換するには，MACRO–32 Compiler for OpenVMS I64 を使用します。OpenVMS I64 にはその目的で，このコンパイラが含まれています。

一部のVAX MACROコードは変更せずにコンパイルできますが，大部分のコード・モジュールでは，エントリ・ポイント指示文を追加する必要があります。また，多くのコード・モジュールではその他の変更も必要です。

注意

MACRO–32 コンパイラは，ソース・コードに LIB$ESTABLISH が含まれている場合には，それを呼び出そうとします。

MACRO–32 プログラムが，0(FP) にルーチン・アドレスを格納することにより動的ハンドラを確立する場合には，OpenVMS I64 システムでコンパイルしても，そのプログラムは正しく動作します。ただし，JSB (Jump to Subroutine) ルーチンの内部から条件ハンドラ・アドレスを設定することはできません。条件ハンドラ・アドレスの設定は，必ず CALL_ENTRY ルーチンの内部から行います。

コンパイラは OpenVMS I64 システム用に最適化されたコードを生成しますが，VAX MACRO言語には高レベルの制御機能をプログラマに提供する多くの機能があるため， OpenVMS I64 システム用の最適なコードを生成することが困難になっています。 OpenVMS I64 用に新たなプログラムを開発する場合には，中級言語または高級言語を使用することをお勧めします。MACRO–32コンパイラについての詳細は，『OpenVMS MACRO-32 Porting and User's Guide』を参照してください。

3.2.1.3 I64 の開発ツール

コンパイラに加えて，ネイティブ I64 アプリケーションの開発，デバッグ，展開にさまざまなツールが利用できます。表 3–1 にこれらのツールの要約を示します。

表 3–1 OpenVMS の開発ツール

ツール	説明
OpenVMS リンカ	OpenVMS リンカは， I64 オブジェクト・ファイルを受け付け， I64 イメージを生成します。 OpenVMS リンカについての詳細は，第 4 章を参照してください。
OpenVMS DEBUG	OpenVMS I64 上の OpenVMS DEBUG は，シンボリックなソース・レベル・デバッガで，いくつかのグラフィカル・インタフェースを備えています。ユーザ・モード・アプリケーションのデバッグを目的としており，コマンドとインタフェースは OpenVMS VAX 上のものと同じです。 OpenVMS DEBUG についての詳細は，『OpenVMS デバッガ説明書』を参照してください。
DELTA デバッガ	OpenVMS I64 上の DELTA デバッガは，高いモード (スーパバイザ・モード，エグゼクティブ・モード，カーネル・モード) で動作するプロセス・ベースのアプリケーション用の，シンボリックでない命令レベル・デバッガです。詳細は，『OpenVMS Delta/XDelta Debugger Manual』を参照してください。
System Code Debugger (SCD)	SCD は，オペレーティング・システムとデバイス・ドライバのコードをデバッグするための，グラフィカルでシンボリックなソース・レベル・デバッガです。詳細は，『OpenVMS System Analysis Tools Manual』を参照してください。
XDelta デバッガ	XDelta デバッガは，オペレーティング・システムとデバイス・ドライバのコードをデバッグするために使用する，シンボリックでないデバッガです。 OpenVMS I64 上の XDELTA についての詳細は，『OpenVMS Delta/XDelta Debugger Manual』を参照してください。

(次ページに続く)

表 3–1 (続き) OpenVMS の開発ツール

ツール	説明
OpenVMS ライブラリアン・ユーティリティ	OpenVMS ライブラリアン・ユーティリティは，I64 ライブラリを作成します。
OpenVMS メッセージ・ユーティリティ	OpenVMS メッセージ・ユーティリティを使用すれば，OpenVMS システム・メッセージに独自のメッセージを追加できます。
IAS (Itanium アセンブラ)	OpenVMS I64 システム用の IAS アセンブラは，Intel Itanium プロセッサ向けのネイティブ・アセンブラです。このアセンブラはオペレーティング・システムには添付されていませんが，OpenVMS I64 ディストリビューションに付属している Open Source CD に収録されています。
ANALYZE/IMAGE	ANALYZE/IMAGE ユーティリティは，I64 イメージを分析できます。
ANALYZE/OBJECT	ANALYZE/OBJECT ユーティリティは，I64 オブジェクトを分析できます。
DECset	DECset は総合的な開発ツール群であり，Language Sensitive Editor (LSE)，Digital Test Manager (DTM)，Code Management System (CMS)，および Module Management System (MMS) が含まれています。
コマンド定義ユーティリティ	コマンド定義ユーティリティ (CDU) を使用すると，アプリケーション開発者が DCL コマンドを作成することができます。
System Dump Analyzer (SDA)	SDA は，OpenVMS I64 システム固有の情報を表示するように拡張されました。
Crash Log Utility Extractor (CLUE)	CLUE は，クラッシュ・ダンプと各クラッシュ・ダンプの重要なパラメータの履歴を記録し，重要な情報を抽出して要約するためのツールです。

3.2.2 トランスレーション

I64 システムで実行するために VAX イメージをトランスレートする処理については，『DECmigrate for OpenVMS AXP Systems Translating Images』および『HP OpenVMS Migration Software for Alpha to Integrity Servers: **イメージ変換ガイド**』を参照してください。

3.3 システム・クラッシュの分析

OpenVMS では，システム・クラッシュを分析するために 2 つのツールを使用できます。それは System Dump Analyzer (SDA) と Crash Log Utility Extractor (CLUE) です。

3.3.1 System Dump Analyzer

OpenVMS I64 システムの System Dump Analyzer (SDA) ユーティリティは，OpenVMS VAX システムで提供されるユーティリティとほとんど同じです。多くのコマンド，修飾子，表示は同一ですが， Crash Log Utility Extractor (CLUE) ユーティリティの機能にアクセスするためのいくつかのコマンドも含めて，コマンドと修飾子が追加されています。また，プロセッサ・レジスタや構造体など OpenVMS I64 システム固有の情報を表示できるように，一部の表示も変更されています。

SDA インタフェースは少しだけ変更されていますが， VAX と I64 のダンプ・ファイルの内容と，ダンプからシステム・クラッシュを分析するための全体的な処理は， 2 種類のシステムで大きく違います。 I64 の実行パスは， VAX の実行パスよりも複雑な構造体とパターンをスタックに残します。

VAX コンピュータで SDA を使用するには，まず， VAX システムの OpenVMS 呼び出し規則を十分に理解しておく必要があります。同様に， I64 システムで SDA を使用するには，まず， I64 システムの OpenVMS 呼び出し規則を十分理解しておかなければ，スタックを見てクラッシュのパターンを解読できるようになりません。

SDA は以下のように変更されています。

- SHOW CRASH コマンドと SHOW STACK コマンドの表示には，致命的なシステム例外バグ・チェックのデバッグを簡単にするための追加情報が含まれるようになりました。
- SHOW EXEC コマンドの表示には，イメージの**スライシング**を使用してロードされた場合，エグゼクティブ・イメージに関する追加情報が含まれるようになりました。スライシングは，エグゼクティブ・イメージの場合はエグゼクティブ・イメージ・ローダが実行し，ユーザ・モード・イメージの場合は， OpenVMS インストール・ユーティリティが実行する機能です。エグゼクティブ・イメージ (またはユーザ・モード・イメージ) をスライシングすると，数が制限されているトランスレーション・バッファ・エントリの競合を削減できるため，性能が大幅に向上します。
- SDA の新しいコマンドである MAP コマンドを使用すると，メモリ内のアドレスをマップ・ファイルのイメージ・オフセットにマップできます。
- FPCR という新しいシンボルがシンボル・テーブルに追加されました。このシンボルは浮動小数点レジスタを表します。

3.3.2 Crash Log Utility Extractor

Crash Log Utility Extractor (CLUE) は，クラッシュ・ダンプの履歴と，各クラッシュ・ダンプの重要なパラメータを記録し，重要な情報を抽出して要約するためのツールです。クラッシュ・ダンプは，システム・クラッシュが発生するたびに上書きされるため，最新のクラッシュに対してだけしか使用できませんが，クラッシュ履歴ファイル (OpenVMS VAX) と，各クラッシュに対して個別のリスト・ファイルを持つ要約

クラッシュ履歴ファイル(OpenVMS I64)は，システム・クラッシュを永久的に記録します。

OpenVMS VAX と OpenVMS I64 でのインプリメント上の相違点は，表 3–2 に示すとおりです。

表 3–2 OpenVMS VAX と OpenVMS I64 の CLUE の相違点

属性	OpenVMS VAX	OpenVMS I64
アクセス方式	独立したユーティリティとして起動される。	SDA を通じてアクセスされる。
履歴ファイル	各クラッシュのクラッシュ・ダンプ・ファイルから 1 行の要約情報と詳細情報を格納した累積ファイル。	各クラッシュ・ダンプに対して 1 行の要約だけを格納した累積ファイル。各クラッシュの詳細情報は別のリスト・ファイルに格納される。
クラッシュ・ダンプのデバッグ以外の使い方	なし	CLUE コマンドは会話方式で使用でき，実行中のシステムを確認できる。

3.4 基準情報を得るための VAX 上でのアプリケーションのテスト

テストの最初の手順は，アプリケーションの実行結果の基準データを取るために，VAX アプリケーションに対してテスト・スイートを実行することです。この作業は，アプリケーションを I64 にポーティングする前に実行してもポーティングした後で実行してもかまいません。その後，これらのテストの結果と， I64 システム上での同様のテストの結果を比較します。

3.5 移行したアプリケーションのテスト

移行したバージョンの機能を VAX バージョンと比較するために，アプリケーションをテストしなければなりません。

第 3.4 節で説明したように，テストではまず VAX アプリケーションに対してテスト・スイートを実行することにより，アプリケーションの基準となる結果を確立することです。

I64 システムでアプリケーションが動作するようになったら，以下の 2 種類のテストを実施します。

- アプリケーションの VAX バージョンに対して使用される標準テスト
- アーキテクチャの変更による問題を特に確認するための新しいテスト

3.5.1 VAX テストの I64 へのポーティング

新しいアーキテクチャを使用することにより，アプリケーションの一部が変更されるため， OpenVMS I64 にアプリケーションを移行した後でそのアプリケーションをテストすることは特に重要です。アプリケーションの変更によってエラーが発生するだけでなく，新しい環境では， VAX バージョンで検出されなかった問題が発生する可能性があります。

移行されたアプリケーションをテストするには，以下の手順を実行します。

1. 移行する前に，アプリケーションの完全な標準データを入手する。
2. アプリケーションだけでなく，テスト・スイートも移行する (そのテストが I64 でまだ使用できない場合)。
3. I64 システムでテスト・スイートを検証する。
4. 移行したアプリケーションに対して移行したテストを実行する。

ここでは，回帰テストとストレス・テストが役立ちます。ストレス・テストは，同期に関するプラットフォームの相違点をテストするために特に重要です。特に，複数の実行スレッドを使用するアプリケーションの場合は，ストレス・テストが役立ちます。

3.5.2 新しい I64 テスト

標準テストは，移行したアプリケーションの機能を検証するために非常に有効ですが，移行に伴って発生する問題点を検証するためのテストも追加しなければなりません。特に以下の点に注意してください。

- コンパイラの相違点
- アーキテクチャの相違点
- 統合 (異なる言語で作成されたモジュールなど)

3.5.3 潜在的なバグの発見

最善を尽くしたとしても， OpenVMS VAX システムでは問題にならなかった潜在的なバグが検出されることがあります。

たとえば， VAX システムでは，プログラムで一部の変数を初期化しなくても問題になりませんが， I64 システムでは算術演算例外が発生することがあります。同様に，使用できる命令やコンパイラの最適化方法が変更されたために， 2 つのアーキテクチャ間で移行したときに，問題が発生することもあります。これまで隠れていたバグをすべて解決できるような簡単な方法はありませんが，プログラムを移植した後，他のユーザが使用を開始する前に，プログラムをテストする必要があります。

3.6 移行したアプリケーションのソフトウェア・システムへの統合

再コンパイルまたはトランスレーションによってアプリケーションを移行した後，他のソフトウェアとのやりとりによる問題や，移行によって発生した問題がないかどうかを確認しなければなりません。

相互操作性に関する問題の原因として，以下のことが考えられます。

- OpenVMS Cluster 環境の I64 システムと VAX システムは，別々のシステム・ディスクを使用しなければなりません。アプリケーションが正しいシステム・ディスクを参照しているかどうかを確認する必要があります。
- アーキテクチャが混在する環境では，アプリケーションが正しいイメージ名のものを参照していることを確認してください。
  - イメージのネイティブ VAX バージョンとネイティブ I64 バージョンが同じ名前である
  - VAX イメージをトランスレートしたバージョンでは，名前の最後に "_TV_AV"が追加されている。 Alpha イメージをトランスレートしたバージョンでは，名前の最後に "_AV"が追加されている
- 再コンパイルしたイメージではデータが自然にアラインされていることが必要ですが，トランスレートされたイメージでは， VAX 形式でデータがアラインされており， I64 形式でデータがアラインされていない可能性があります。

3.7 特定の種類のコードの変更

以下のコーディング手法および特定の種類のコードは， I64 システムで実行するために変更する必要があります。

- Alpha システムと VAX システムのどちらかで動作するように記述された条件付きコード
- VAX アーキテクチャに依存する OpenVMS システム・サービスを使用しているコード
- それ以外で VAX アーキテクチャに依存しているコード
- 浮動小数点データ型を使用しているコード
- コマンド定義ファイルを使用しているコード
- スレッドを使用しているコード (特に，個別に作成されたタスクまたはスタックを切り替えるコード)
- 特権コード

3.7.1 条件付きコード

ここでは，I64 に移行するときに，OpenVMS コードを条件付きにする方法について説明します。このコードは，VAX と I64 の両方を対象にコンパイルされるか，または VAX，Alpha，および I64 を対象にコンパイルされます。シンボル EVAX は削除されておらず，新しいシンボル ALPHA が追加されています。EVAX を ALPHA で置き換える必要はありませんが，必要であれば置き換えてもかまいません。MACRO および BLISS で使用可能なアーキテクチャ・シンボルは，VAX，EVAX，ALPHA，および I64 です。

3.7.1.1 MACRO のソース

MACRO–32 ソース・ファイルの場合，アーキテクチャ・シンボルは ARCH_DEFS.MAR に定義されています。このファイルは，コマンド行で指定されるプレフィックス・ファイルです。VAX システムでは，VAX は 1 として定義されており，Alpha，EVAX，および I64 は未定義です。I64 システムでは，I64 が 1 として定義されており，VAX，EVAX，および ALPHA は未定義です。

以下の例は，VAX システムと I64 システムの両方で実行できるように MACRO–32 ソース・コードを条件付けする方法を示しています。

VAX 固有のコードの場合

```
.IF DF VAX
        .
        .
        .
.ENDC
```

I64 固有のコードの場合

```
.IF DF I64
        .
        .
        .
.ENDC
```

3.7.1.2 BLISS のソース

BLISS ソース・ファイル (BLISS-32 または BLISS-64) の場合，マクロ VAX，EVAX，ALPHA，および I64 が ARCH_DEFS.REQ で定義されています。VAX では，VAX に 1 が定義されており，Alpha，EVAX および I64 は未定義です。I64 では，I64 に 1 が定義されており，VAX，EVAX，および ALPHA は 0 として定義されています。BLISS の条件付き文でこれらのシンボルを使用するには，ARCH_DEFS.REQ が必要です。

注意

%BLISS(xxx)，%TARGET(xxx)，および%HOST(xxx) という構造は推奨されません。ただし，これらの構造を必ず変更しなければならないというわけではなく，必要に応じて変更すればよいと理解してください。

ソース・ファイルに次の文を追加します。

```
REQUIRE 'SYS$LIBRARY:ARCH_DEFS';
```

VAX システムと I64 システムの両方で実行するには，ソース・ファイルで以下の文を使用してコードを条件付けします。

VAX 固有のコードの場合

```
%if VAX %then
        .
        .
        .
%fi
```

I64 固有のコードの場合

```
%if I64 %then
        .
        .
        .
%fi
```

3.7.1.3 C のソース

C ソース・ファイルの場合，適切なプラットフォーム上のコンパイラで，__vax，__VAX，__ia64，および__ia64__というシンボルが提供されます。シンボルはコンパイル・コマンド行で定義できますが，この方法は推奨できません。また，arch_defs.hを使用して定義する方法も推奨できません。標準的な C のプログラミング手法では，#ifdefを使用します。

VAX 固有のコードの場合

```
#ifdef __vax
        .
        .
        .
#endif
```

I64 固有のコードの場合

```
#ifdef __ia64
        .
        .
        .
#endif
```

3.7.1.4 既存の条件付きコード

既存の条件付きコードは，変更が必要かどうか確認しなければなりません。次のBLISS のコードについて考えます。

```
%IF VAX %THEN
  vvv
  vvv
%FI

%IF EVAX %THEN
  aaa
  aaa
%FI
```

コードが本当にアーキテクチャ固有で， I64 コードを追加する場合は，次のコードを追加します。

```
%IF I64 %THEN
  iii
  iii
%FI
```

しかし，既存の VAX/EVAX 条件が実際には 64 ビットではなく 32 ビットを，あるいは OpenVMS の新規則ではなく旧規則 (たとえば，データ構造の拡張の有無や呼び出すルーチンの異同) を反映するのであれば，以下の条件付きコードの指定方法の方がより適切だと考えられます。これは， Alpha と I64 のコードは同じであり， 64 ビット・コードと VAX のコードとは区別する必要があるからです。

```
%IF VAX %THEN
  vvv
  vvv
%ELSE
  aaa
  aaa
%FI
```

3.7.2 VAX アーキテクチャに依存しているシステム・サービス

特定のシステム・サービスは， OpenVMS VAX ではアプリケーション内で問題なく動作しても， I64 へのポーティングがうまくいかないことがあります。ここでは，このようなシステム・サービスと，それに代わるサービスについて説明します。

3.7.2.1 SYS$GOTO_UNWIND

OpenVMS VAX では， SYS$GOTO_UNWIND システム・サービスは，プロシージャ起動コンテキスト・ハンドルを 32 ビット・アドレスで受け取ります。このシステム・サービスを呼び出している箇所を 64 ビット起動ハンドル用の SYS$GOTO_UNWIND_64 に変更しなくてはなりません。ソース・コードを変更して， 64 ビット値の領域を割り当ててください。また， OpenVMS I64 の起動コンテキスト・ハ

ンドルを返すライブラリ・ルーチンも異なります。詳細は，『『OpenVMS Calling Standard』』を参照してください。

SYS$GOTO_UNWIND サービスは，プログラミング言語機能をサポートする際によく使用されます。多くの場合は，コンパイラやランタイム・ライブラリでの変更になりますが， SYS$GOTO_UNWIND を直接使用している場合は，変更が必要です。

3.7.2.2 SYS$LKWSET と SYS$LKWSET_64

システム・サービス SYS$LKWSET および SYS$LKWSET_64 は変更されています。詳細は，第 3.7.9 項を参照してください。

3.7.3 VAX アーキテクチャに依存するその他の機能を含むコード

VAX でのコーディング手法には， I64 で使用すると異なる結果になるものがいくつかあります。ここでは，そのためにアプリケーションを変更しなければならない場合について説明します。

3.7.3.1 初期化されたオーバレイ・プログラム・セクション

初期化されたオーバレイ・プログラム・セクションの取り扱いは， I64 システムでは異なります。 OpenVMS VAX システムでは，オーバレイ・プログラム・セクションの異なる部分を複数のモジュールで初期化することができます。このような初期化は， OpenVMS I64 システムでは認められていません。

3.7.3.2 条件ハンドラでの SS$_HPARITH の使用

OpenVMS VAX では，多くの算術演算エラー条件に対して SS$_HPARITH または「エラー・コード」が通知されます。 OpenVMS I64 では，算術演算エラー条件に対して SS$_HPARITH が通知されることはありません。 OpenVMS I64 では，より細分化されたエラー・コード SS$_FLTINV と SS$_FLTDIV が通知されます。

これらの細分化されたエラー・コードを検出するように，条件ハンドラを更新してください。両方のアーキテクチャで共通のコードを使用するには，コードでSS$_HPARITHを参照している箇所において， OpenVMS I64 ではSS$_FLTINVとSS$_FLTDIVも検出するように変更します。

3.7.3.3 メカニズム・アレイ構造体

OpenVMS I64 のメカニズム・アレイ構造体は， OpenVMS VAX の場合と大きく異なります。戻り状態コード RETVAL は， Alpha プラットフォームと I64 プラットフォームの両方の戻り状態レジスタを表すように拡張されています。詳細は，『OpenVMS Calling Standard』を参照してください。

3.7.3.4 VAX のオブジェクト・ファイル形式への依存

OpenVMS I64 システムで作成されるオブジェクト・ファイルの形式は VAX の形式と異なるため，コードが VAX のオブジェクト・ファイルのレイアウトに依存している場合は，変更しなくてはなりません。

オブジェクト・ファイルの形式は，『System V Application Binary Interface』 2001 年 4 月 24 日付けドラフトに記述されている ELF (Executable and Linkable Format) の 64 ビット版に準拠しています。このドキュメントは Caldera が発行しており，次の Web サイトで入手できます。

caldera.com/developers/gabi

また，オブジェクト・ファイルの形式は，『Intel® Itanium® Processor-specific Application Binary Interface (ABI)』 2001 年 5 月発行 (ドキュメント番号 245270-003) に記載されている I64 固有の拡張にも準拠しています。 OpenVMS オペレーティング・システム固有のオブジェクト・ファイルおよびイメージ・ファイル機能をサポートするのに必要な拡張や制限事項も，将来のリリースでドキュメントされる予定です。

イメージ内のデバッガが使用する部分は， DWARF Version 3 業界標準に準拠しています。このドキュメントは次の Web サイトで入手できます。

http://www.eagercon.com/dwarf/dwarf3std.htm

OpenVMS I64 でのデバッグ・シンボル・テーブルの表現は，このドキュメントに記述されている業界標準の DWARF デバッグ・シンボル・テーブルの形式です。 DWARF Version 3 形式に対する弊社の拡張については，将来のリリースで公開される予定です。

3.7.4 浮動小数点データ型を使用するコード

OpenVMS VAX では， VAX 浮動小数点データ型をハードウェアでサポートしています。 OpenVMS I64 では， IEEE 浮動小数点データ型をハードウェアでサポートしており， VAX 浮動小数点データ型をソフトウェアでサポートしています。

ほとんどの OpenVMS I64 のコンパイラでは， VAX 浮動小数点データ型を生成できるように，/FLOAT=D_FLOAT および/FLOAT=G_FLOAT 修飾子が用意されています。これらの修飾子を指定しないと， IEEE 浮動小数点データ型が使用されます。

I64 BASIC コンパイラを使用してデフォルトの浮動小数点データ型を指定するには，/REAL_SIZE 修飾子を使用します。指定できる値は， SINGLE (Ffloat)， DOUBLE (Dfloat)， GFLOAT， SFLOAT， TFLOAT，および XFLOAT です。

VAX 浮動小数点を使用するオプションを指定した OpenVMS アプリケーションを I64 でコンパイルすると，コンパイラは浮動小数点形式を変換するコードを自動的に生成します。アプリケーションで一連の算術演算を実行すると，このコードは以下の処理を行います。

1. VAX 浮動小数点形式を，長さに応じて IEEE 単精度または IEEE 倍精度浮動小数点形式に変換します。
2. 算術演算を IEEE 浮動小数点演算で実行します。

3. 演算結果を IEEE 形式から VAX 形式に戻します。

算術演算が実行されない場合 (VAX 浮動小数点データをフェッチした後にストアを行う場合) は，変換は行われません。コードではこのような状況を移動として扱います。

VAX 形式と IEEE 形式には以下の相違点があるため，算術演算の結果が異なる場合がまれにあります。

- 表現される数値の範囲
- 丸めの規則
- 例外の動作

これらの違いにより，特定のアプリケーションでは問題が発生することがあります。

OpenVMS VAX と OpenVMS I64 の浮動小数点データ型の相違点，およびこれらの相違点がポーティング後のアプリケーションに与える影響の詳細については，第 9 章およびホワイト・ペーパー『Intel® Itanium®アーキテクチャにおける OpenVMS 浮動小数点演算について』を参照してください。このホワイト・ペーパーが入手できる Web サイトについては，「まえがき」の「関連資料」を参照してください。

注意

浮動小数点に関する上記のホワイト・ペーパーが作成された後で，/IEEE_MODE のデフォルトが FAST から DENORM_RESULTS に変更されました。つまり，デフォルト設定で浮動小数点演算を実行した場合，VAX 形式の浮動小数点演算や/IEEE_MODE=FAST を使用したときに致命的な実行時エラーになっていたケースで，Infinity または Nan として出力される値が生成される場合があります (業界標準の動作)。また，値が非常に小さいために正規化された値として表現できなくなると，ただちに 0 になるのではなく，結果がデノーマル範囲になることが認められるため，このモードでの非ゼロ最小値は，はるかに小さな値になります。このデフォルトは，プロセス起動時に I64 で設定されるデフォルトと同じです。

3.7.4.1 LIB$WAIT に関する問題と解決策

OpenVMS I64 に移植したコードで LIB$WAIT が使用されていると，予測外の結果が発生することがあります。以下に C の例を示します。

```
float wait_time = 2.0;
lib$wait(&wait_time);
```

OpenVMS I64 システムでは，このコードは S_FLOATING を LIB$WAIT に送ります。しかし，LIB$WAIT は F_FLOATING を想定しているため，FLTINV 例外が発生します。

LIB$WAIT には 3 つの引数を指定することができます。 LIB$WAIT で 3 つの引数を使用して上記のコードを書き直すと， I64 システムと Alpha システムの両方で正常に動作するコードを作成できます。以下の変更後のコードは，/FLOAT 修飾子を指定せずにコンパイルすると，正常に動作します。

```
#ifdef __ia64
 int float_type = 4;  /* use S_FLOAT for I64 */
#else
 int float_type = 0;  /* use F_FLOAT for Alpha */
#endif
 float wait_time = 2.0;
 lib$wait(&wait_time,0,&float_type);
```

よりよいコーディング手法として，アプリケーションで (SYS$STARLET_C.TLB から) LIBWAITDEF をインクルードし，浮動小数点データ型の名前を指定する方法があります。このようにして作成したコードは，保守がより容易になります。

LIBWAITDEF は以下のシンボルをインクルードします。

- LIB$K_VAX_F
- LIB$K_VAX_D
- LIB$K_VAX_G
- LIB$K_VAX_H
- LIB$K_IEEE_S
- LIB$K_IEEE_T

以下の例では，コードに libwaitdef.h をインクルードし，浮動小数点データ型の名前を指定する方法を示しています。この例でも，プログラムのコンパイル時に/FLOAT 修飾子が指定されないことを前提にしています。

```
#include <libwaitdef.h>
.
.
.
#ifdef __ia64
 int float_type = LIB$K_IEEE_S;  /* use S_FLOAT for IPF */
#else
 int float_type = LIB$K_VAX_F;  /* use F_FLOAT for Alpha */
#endif
 float wait_time = 2.0;
 lib$wait(&wait_time,0,&float_type);
```

3.7.5 コマンド・テーブル宣言に関する注意

OpenVMS I64 へ移植するコードに正しくないコマンド・テーブル宣言があると，予想外の結果となる場合があります。たとえば，あるアプリケーションで， CLD ファイルからオブジェクト・モジュールを作成するのに CDU を使用しているとします。このアプリケーションは CLI$DCL_PARSE を呼び出し，コマンド行を解析します。CLI$DCL_PARSEは次のようなエラーで失敗する場合があります。

```
%CLI-E-INVTAB, command tables have invalid format - see documentation
```

このコードは，コマンド・テーブルを外部データ・オブジェクトとして定義するように修正する必要があります。

たとえば， VAX あるいは Alpha アプリケーションの BLISS モジュールで，コマンド・テーブル (DFSCP_CLD) が次のように間違って宣言されていたとします。

```
EXTERNAL ROUTINE DFSCP_CLD
```

これは次のように変更する必要があります。

```
EXTERNAL DFSCP_CLD
```

Fortran モジュールでコマンド・テーブルが次のように誤って宣言されているとします。

```
EXTERNAL DFSCP_CLD
```

これは次のように変更する必要があります。

```
INTEGER DFSCP_CLD
CDEC$ ATTRIBUTES EXTERN :: DFSCP_CLD
```

同様に， C 言語で作成されたアプリケーションで次のようなコマンド・テーブルが定義されていたとします。

```
int jams_master_cmd();
```

このコードは，次のように外部参照に変更する必要があります。

```
extern void* jams_master_cmd;
```

正しい宣言に変更すると， VAX， Alpha，および I64 のすべてのプラットフォームで動作するようになります。

3.7.6 スレッドを使用するコード

OpenVMS I64 では，これまでに OpenVMS でサポートされたすべてのスレッド・インタフェースがサポートされます。スレッドを使用する大部分の OpenVMS VAX コードは，全く変更なしで OpenVMS I64 へのポーティングが可能です。ここでは，その例外について説明します。スレッドを使用するコードのポーティングで発生する最大の問題は，スタック領域の使用です。 I64 のコードは，対応する VAX のコードよりはるかに大きいスタック領域を使用します。このため，スレッド化されたプログラムが VAX で問題なく動作していても， I64 ではスタック・オーバフローが発生することがあります。

オーバフローの問題を軽減するために， OpenVMS I64 ではデフォルトのスタック・サイズが拡大されています。しかし，アプリケーションで特定のスタック・サイズを要求している場合は，そのデフォルトの変更は影響しないため，より大きなスタックを要求するようにアプリケーションのソース・コードを変更しなければなりません。そのような場合は，まず， 8 KB ページを 3 つ (24576 バイト) 追加してみることをお勧めします。

必要なスタック・サイズを拡大した結果，もう 1 つの副作用として， P0 空間に対する要求が拡大します。スレッド・スタックは P0 ヒープから割り当てられます。スタックが大きくなると，プロセスのメモリ・クォータを超過してしまう可能性があります。極端な場合， P0 空間が完全に満杯になり，プロセスで同時に使用されるスレッドの数を減らさなければならなくなることもあります (またはその他の変更を行って， P0 メモリに対する要求を軽減します)。

『HP OpenVMS Version 8.2 リリース・ノート[翻訳版]』を参照して， OpenVMS でスレッドのサポートに関して行われた最新の機能向上について十分理解しておくことをお勧めします。 POSIX Threads C 言語ヘッダ・ファイル PTHREAD_EXCEPTION.H で 1 つの変更が行われたため，以前の動作に依存しているアプリケーションのポーティングを行うと，問題が発生します。

OpenVMS I64 では， DCL の THREADCP コマンドはサポートされません。 OpenVMS I64 では，スレッド関係のイメージ・ヘッダ・フラグの状態のチェックおよび修正には， OpenVMS VAX の THREADCP コマンドの代わりに， DCL コマンド SET IMAGE および SHOW IMAGE が使用できます。詳細は『OpenVMS DCL ディクショナリ』を参照してください。 THREADCP コマンドについては，『Guide to the POSIX Threads Library』で説明しています。

スレッド関係のイメージ・フラグの設定を変更したい場合は，次のように新しいコマンド SET IMAGE を使用する必要があります。

```
$ SET IMAGE/FLAGS=(MKTHREADS,UPCALLS) FIRST.EXE
```

3.7.6.1 スレッド・ルーチン cma_delay および cma_time_get_expiration

2つの既存のスレッドAPIライブラリ・ルーチンcma_delayとcma_time_get_expirationは，VAX F_FLOAT形式を使用する浮動小数点形式のパラメータを受け付けます。これらのルーチンを呼び出すアプリケーション・モジュールは，/FLOAT=D_FLOAT修飾子または/FLOAT=G_FLOAT修飾子を指定してコンパイルし，VAX F FLOATがサポートされるようにしなければなりません(ただし，アプリケーションで倍精度バイナリ・データも使用している場合は，/FLOAT=G_FLOAT修飾子を使用する必要があります)。浮動小数点のサポートについての詳細は，コンパイラのマニュアルを参照してください。

cma_delayあるいはcma_time_get_expirationのいずれかを使用するC言語モジュールが，IEEE浮動小数点モードで間違ってコンパイルされると，次のようなコンパイラの警告メッセージが表示されます。

```
cma_delay (
    ^
%CC-W-LONGEXTERN, The external identifier name exceeds 31
characters; truncated to "CMA_DELAY_NEEDS_VAX_FLOAT_____".
```

このような警告メッセージの原因となるようなオブジェクト・ファイルがリンクされている場合，リンカはこのシンボルに対して未定義シンボル・メッセージを表示します(リンカが生成したイメージをこの後で実行すると，ルーチンの呼び出しはACCVIOで失敗します)。これらのコンパイラおよびリンカの診断メッセージの目的は，これらのCMAルーチンではVAX形式の浮動小数点値を使用する必要があり，この要件を満たさない形でコンパイルが行われたことを警告することです。

注意

これらのルーチンについては，ユーザ向けのドキュメントには記載されていませんが，既存のアプリケーションで使用できるように，サポートが継続されています。

3.7.7 アラインされていないデータを含むコード

データ参照の性能を最適なレベルに向上させるには，データを自然にアラインすることをお勧めします。データが自然にアラインされていない場合，OpenVMS I64での性能が大幅に低下します。

データのアドレスが，バイト数で表したデータ・サイズの整数倍になっている場合，そのデータは**自然にアライン**されています。たとえば，ロングワードは，4の倍数のアドレスで自然にアラインされ，クォドワードは8の倍数のアドレスで自然にアラインされます。構造体のすべてのメンバが自然にアラインされると，その構造体も自然にアラインされます。

自然なアラインメントができない場合もあるので，OpenVMS I64 システムでは，アラインされていないデータを参照することによる影響を管理するための支援機能を提供しています。I64 コンパイラは，発生する可能性のある大部分のアラインメントの問題を自動的に修正し，修正できない問題にはメッセージを出力します。

自然にアラインされていない共用データがあると，性能が低下するだけでなく，プログラムが正常に実行されない原因にもなります。したがって，共用データは自然にアラインする必要があります。データの共用は，1 つのプロセス内のスレッドの間，プロセスと AST の間，複数のプロセスで使われるグローバル・セクションにおいて発生します。

問題の検出

自然にアラインされていないデータのインスタンスを検出するには，アラインされていないデータへの参照をコンパイラがコンパイル時に報告する修飾子を使用します。この修飾子は I64 のほとんどのコンパイラで提供されています。表 3–3 にこの修飾子を示します。

表 3–3 コンパイル時の参照を報告するためのコンパイラ・スイッチ

コンパイラ	修飾子
BLISS	/CHECK=ALIGNMENT
C	/WARN=ENABLE=ALIGNMENT
Fortran	/WARNING=ALIGNMENT
HP Pascal	/USAGE=PERFORMANCE

自然にアラインされていないデータを実行時に検出する OpenVMS デバッガの修飾子など，他の支援機能は，将来のリリースで計画されています。

問題への対処方法

アラインされていないデータをなくすには，次の方法を使用します。

- 自然なアラインメントでコンパイルするか，または言語がこの機能を備えていない場合には，自然なアラインメントになるようにデータを移動します。データが確実にアラインされるように間隔を空けると，メモリ・サイズが増えるという問題があります。メモリを節約すると同時に，確実にデータを自然にアラインする方法として，サイズの大きい変数を先に宣言する方法があります。
- 構造体内で強制的に自然なアラインメントが実現されるように，高級言語命令を使用します。
- データ項目をクォドワード境界にアラインします。

注意

自然なアラインメントに変換されたソフトウェアは，同じ OpenVMS Cluster 環境内のシステム，またはネットワークによって接続されたシステムで実行されるトランスレートされた他のソフトウェアと互換性がなくなる可能性があります。次の例を参照してください。

- **既存のファイル形式は，アラインされていないデータを含むレコードを定義している可能性があります。**
- **トランスレートされたイメージは，アラインされていないデータをネイティブ・イメージに渡したり，ネイティブ・イメージからそのようなデータが渡されることを必要とする可能性があります。**

このような場合には，アプリケーションのすべての部分が同じデータ型，つまり，アラインされたデータ型またはアラインされていないデータ型を要求するように変更しなければなりません。

3.7.8 OpenVMS VAX の呼び出し規則に依存するコード

OpenVMS VAX の呼び出し規則に明示的に依存するアプリケーションは，変更が必要になる可能性があります。 OpenVMS I64 の呼び出し規則は， Intel の呼び出し規則をベースにしており，部分的に OpenVMS 固有の変更が加えられています。 OpenVMS I64 の呼び出し規則で導入された大きな相違点は次のとおりです。

- フレーム・ポインタ (FP) はない。
- 複数のスタックが使用される。
- 4 つのレジスタだけが呼び出し時に保存される。
- 従来のレジスタ番号が変更されている。

詳細は，『OpenVMS Calling Standard』を参照してください。

3.7.9 特権コード

ここでは，確認が必要で，場合によっては変更も必要な特権コードについて説明します

3.7.9.1 SYS$LKWSET と SYS$LKWSET_64 の使用

プログラム自身をメモリへロックするのに SYS$LKWSET または SYS$LKWSET_64 システム・サービスを使用しており， VAX システムでアプリケーションを実行しない場合は，これらを (OpenVMS Version 8.2 で導入された) LIB$LOCK_IMAGE という新しい RTL ルーチンに置き換えることをお勧めします。同様に， SYS$ULWSET および SYS$ULWSET_64 も LIB$UNLOCK_IMAGE という新しい RTL ルーチンに置き換えます。

カーネル・モードに入り， IPL を 2 より大きな値に上げるプログラムは，プログラム・コードとデータをワーキング・セット内にロックしなければなりません。コードとデータのロックが必要なのは，システムが PGFIPLHI バグ・チェックでクラッシュするのを回避するためです。

VAX システムでは通常，ロックが必要なのはプログラムで明示的に参照されるコードとデータだけでした。 Alpha システムでは，プログラムで参照されるコード，データ，およびリンケージ・データをロックしなければなりません。 I64 システムでは，コード，データ，ショート・データ，およびリンカで生成されたコードをロックする必要があります。ポーティングを容易にするため，また，ショート・データとリンカ生成データのアドレスはイメージ内で簡単に検出できないという理由から， Alpha と I64 の SYS$LKWSET および SYS$LKWSET_64 システム・サービスに変更が加えられています。

OpenVMS Version 8.2 では， SYS$LKWSET および SYS$LKWSET_64 システム・サービスは，渡された最初のアドレスを調べます。このアドレスがイメージの内部にある場合は，これらのサービスはイメージ全体をワーキング・セット内にロックしようとします。正常終了状態コードが返されれば，プログラムが IPL を 2 より大きな値に上げても，システムが PGFIPLHI バグ・チェックでクラッシュすることはありません。

ワーキング・セット内でイメージのロックが成功した回数を数えるカウンタが，内部 OpenVMS イメージ構造で管理されています。このカウンタは，ロックされたときに増分され，アンロックされたときに減分されます。カウンタが 0 になると，イメージ全体がワーキング・セットからアンロックされます。

特権プログラムが Alpha および I64 用であり VAX では使われない場合は，コード，データ，およびリンケージ・データを検索してこれらの領域をワーキング・セット内にロックするようなコードをすべて削除することができます。この種のコードは，LIB$LOCK_IMAGE ルーチンおよび LIB$UNLOCK_IMAGE ルーチン (OpenVMS Version 8.2 で提供) の呼び出しで置き換えることができます。これらのルーチンの方がプログラムにとって単純であり，コードの理解と保守も容易になります。

プログラムのイメージが大きすぎるために，ワーキング・セット内にロックできない場合は，状態コード SS$_LKWSETFUL が返されます。この状態が返された場合は，ユーザのワーキング・セット・クォータを拡大することができます。また，イメージを 2 つの部分に分割することもできます。ユーザ・モードのコードが含まれる部分と，カーネル・モードのコードが含まれる共用イメージに分割します。カーネル・モード・ルーチンを開始するときに，このルーチンは LIB$LOCK_IMAGE を呼び出して，イメージ全体をワーキング・セット内にロックしなければなりません。カーネル・モード・ルーチンを終了する前に，ルーチンで LIB$UNLOCK_IMAGE ルーチンを呼び出す必要があります。

3.7.9.2 SYS$LCKPAG と SYS$LCKPAG_64 の使用

コードをメモリ内にロックするためにアプリケーションでシステム・サービス SYS$LCKPAG または SYS$LCKAPG_64 を使用している場合，このサービスを使用している部分を検討してください。 I64 では，このサービスによりイメージ全体がワーキング・セット内にロックされることはありません。

コードが IPL を上げて実行しているときにページ・フォルトが発生しないように，ワーキング・セット内にイメージをロックしようとしているのかもしれません (第 3.7.9.1 項を参照)。 LIB$LOCK_IMAGE ルーチンおよび LIB$UNLOCK_IMAGE ルーチンについては，『OpenVMS RTL Library (LIB$) Manual』を参照してください。

3.7.9.3 ターミナル・ドライバ

OpenVMS I64 のターミナル・クラス・ドライバのインタフェースは， CALL ベース・インタフェースです。これは，引数の受け渡しにレジスタを使用する OpenVMS VAX の JSB ベースのインタフェースと大きく異なります。

OpenVMS I64 のターミナル・クラス・ドライバのインタフェースについては，『OpenVMS Terminal Driver Port Class Interface for Itanium』を参照してください。このドキュメントは次の Web サイトで入手できます。

~~hp.com/products1/evolution/alpha_retaintrust/openvms/resources~~

3.7.9.4 保護されたイメージ・セクション

保護されたイメージ・セクションは，通常，ユーザ作成のシステム・サービスを実装するための共用イメージで使用されます。これらのイメージ・セクションは，ソフトウェアおよびハードウェアのメカニズムによって保護され，特権のないアプリケーションがこれらのセクションの完全性を危うくすることがないようになっています。VAX および Alpha と I64 のハードウェアのページ保護機能の違いにより，わずかな制限が追加になりますので，保護されたイメージに対して変更が必要になる場合があります。

VAX および Alpha では，特権モード (カーネル・モードまたはエグゼクティブ・モード) で書き込み可能なデータ・セクションは，非特権 (ユーザ) モードで読み取りが可能です。ハードウェア保護機能は，そのようなページに対して，どのモードからも実行アクセスは許可しません。書き込み可能かつ実行可能としてリンクされた保護されたイメージ・セクションは，内部モードでの読み取り，書き込み，実行のみを許可し，ユーザ・モードでのアクセスは許可しません。内部モードでの書き込みが可能なデータへのユーザ・モード・アクセスも，書き込み可能セクションにコードを置くことも一般的ではないため， 1 つのセクションで両方を必要とするアプリケーションはほとんどないと考えられます。

このように VAX と Alpha では例外がありましたが， I64 では，すべての保護された書き込み可能イメージ・セクションは，ユーザ・モードでの書き込みから保護されます。保護されたイメージ・セクションに対するユーザ書き込みを許可する場合は，$SETPRT/$SETPRT_64 システム・サービスを使用してページ保護を変更しなければなりません。

4

再コンパイルと再リンクの概要

この章では，アプリケーションを構成するソース・ファイルを再コンパイルおよび再リンクすることにより，VAX システム上で動くアプリケーションを I64 システムに移行するプロセスについて，その概要を説明します。

一般に，アプリケーションが高級プログラミング言語で作成されている場合には，わずかな作業で I64 システムで実行できるようになります。高級言語では，アプリケーションを下位のマシン・アーキテクチャへの依存から分離します。さらに，I64 システム上のプログラミング環境のほとんどの部分は，VAX システムのプログラミング環境と同じです。I64 版の各言語のコンパイラと OpenVMS リンカ・ユーティリティ (linker) を使用すれば，アプリケーションを構成するソース・ファイルを再コンパイルおよび再リンクして，ネイティブ I64 イメージを作成できます。

アプリケーションがVAX MACROで作成されている場合は，最低限の作業だけで I64 システム上でも実行できる可能性があります。ただし一般には，VAX のアーキテクチャに依存する何らかの要素が含まれており，それに応じた変更を加えなければなりません。

内部モードや高い割り込み優先順位レベル (IPL) で動作する特権アプリケーションは，コード内でオペレーティング・システムの内部的な動作に関するさまざまな仮定を行っているので，大幅な書き換えが必要になります。一般に，このようなアプリケーションは OpenVMS VAX オペレーティング・システムのメジャー・リリースのたびに大幅な変更を加える必要が生じます。

注意

高級言語で作成されたアプリケーションであっても，アーキテクチャ固有の機能に依存することがあります。また，新しいプラットフォームへの移行の際に，アプリケーション内の隠れたバグが表面化することもあります。

4.1 最新版のコンパイラによる VAX でのコンパイル

ネイティブ I64 コンパイラで再コンパイルする前に，まず VAX で動作するコードを最新版のコンパイラでコンパイルすることをお勧めします。これにより，コンパイラのバージョンの変更に起因する問題が見つかる可能性があります。たとえば，新しいバージョンのコンパイラでは，以前は無視されていたプログラミング言語標準への準拠が強制され，アプリケーション・コード中の潜在的な問題が見つかる場合があります。また，新しいバージョンのコンパイラでは，以前のバージョンを作成した時点で

は有効でなかった標準への準拠が適用されることもあります。 OpenVMS VAX 上でこのような問題を修正することで， I64 へのアプリケーションのポーティングが簡単になります。

4.2 ネイティブ I64 コンパイラによるアプリケーションの再コンパイル

VAX システムでサポートされる言語の多くは， I64 システムでもサポートされます。たとえば Fortran や C などです。 I64 システムで使用できる一般的なプログラミング言語のコンパイラについての詳細は，第 9 章を参照してください。

I64 システムで使用できるコンパイラは，それぞれ VAX システムの対応するコンパイラと互換性を維持するように設計されています。各コンパイラは言語標準規格に準拠し， VAX の言語拡張機能の大部分をサポートします。コンパイラは， VAX システムの場合と同じデフォルトのファイル・タイプで出力ファイルを作成します。たとえば，オブジェクト・モジュールのファイル・タイプは.OBJ です。

OpenVMS I64 コンパイラの一覧については，第 3.2.1.1 項を参照してください。

しかし， VAX システムのコンパイラがサポートしている一部の機能は， I64 システムの同じコンパイラでサポートされません。さらに， I64 システムのいくつかのコンパイラは， VAX システムの対応するコンパイラがサポートしていない新機能をサポートしています。互換性を維持するために，一部のコンパイラは互換モードをサポートしています。たとえば， OpenVMS I64 システム用の HP C コンパイラはVAX C互換モードをサポートしています。このモードは/STANDARD=VAXC 修飾子を指定することにより起動されます。

VAX アプリケーションがかなり前に作成された場合，特に古いコンパイラで作成された場合は，移行の準備として，最新版のコンパイラで再作成することをお勧めします。 I64 コンパイラは，古いコンパイラよりも，最新の VAX コンパイラと機能的に互換性があります。

4.3 I64 システムでのアプリケーションの再リンク

ソース・ファイルを正しく再コンパイルした後，アプリケーションを再リンクしてネイティブ I64 イメージをビルドしなければなりません。リンカは現在の VAX システムと同じファイル・タイプで出力ファイルを生成します。たとえば，デフォルトでは，リンカはイメージ・ファイルのファイル・タイプとして.EXE を使用します。

I64 システムではある種のリンク作業を実行する方法が異なるため，アプリケーションをビルドするために使用する LINK コマンドを変更しなければなりません。以下のリストは，アプリケーションのビルド手順に影響を与える可能性のある，リンカの変

更点を示します。詳細は，『OpenVMS Linker Utility Manual』を参照してください。

- **共有イメージ内でのユニバーサル・シンボルの宣言**— アプリケーションが共有イメージを作成する場合には，おそらくアプリケーションのビルド・プロシージャにVAX MACROで作成された転送ベクタ・ファイルが含まれており，共有イメージ内のユニバーサル・シンボルが宣言されています。 I64 システムでは，転送ベクタ・ファイルを作成する代わりに，オプション SYMBOL_VECTOR=optionを指定することにより，リンカ・オプション・ファイルでユニバーサル・シンボルを宣言しなければなりません。
- OpenVMS **エグゼクティブに対するリンク**— VAX システムでは，ビルド・プロシージャにシステム・シンボル・テーブル・ファイル (SYS.STB) をインクルードすることにより， OpenVMS エグゼクティブに対してリンクします。 I64 システムでは，/SYSEXE 修飾子を指定することにより， OpenVMS エグゼクティブに対してリンクします。
- **共有イメージの暗黙の処理**— I64 システムでは，イメージが直接呼び出す共有イメージを，ビルド・プロシージャで指定しなければなりません。

  VAX システムでは，共有イメージとリンクされるイメージの一覧が，リンカによってイメージの依存関係リストに設定されます。たとえば，メイン・イメージを LIBRTL.EXE に対してリンクし， LIBRTL.EXE を LIBOTS.EXE に対してリンクする場合，メイン・イメージが直接 LIBOTS.EXE を呼び出していなくても，LIBRTL と LIBOTS の両方がメイン・イメージの依存関係リスト (共有イメージ・リストとも呼ばれます) に設定されます。 LIBOTS が互換性のない方法で変更された場合は，メイン・イメージを再リンクしなければなりません。

  I64 システムでは，直接リンクしたイメージだけがリンカによってイメージの依存関係リストに設定されます。 I64 で，イメージを LIBRTL.EXE に対してリンクし， LIBRTL.EXE が LIBOTS.EXE に対してリンクされている場合，メイン・イメージの依存関係リストには， LIBRTL だけが設定されます。 LIBOTS.EXE が互換性のない方法で変更された場合， LIBRTL.EXE は再リンクする必要がありますが，メイン・イメージは再リンクの必要はありません。
- **ベースが指定されていないクラスタ**— VAX では， CLUSTER オプションでベース・アドレスを指定することができます。しかし， I64 では， CLUSTER オプションでベース・アドレスを指定することはできません。表 4–2 の CLUSTER=オプションの説明を参照してください。
- OpenVMS I64 **では初期化されたオーバレイ・プログラム・セクションの扱いが異なる**— VAX システムではオーバレイ・プログラム・セクションのさまざまな部分に対して初期化を実行することができます。同じ部分に対して再度初期化を行うと，以前のモジュールによる初期化が上書きされます。リンクするイメージのそれぞれのバイトについて，それに対して実行された最後の初期化が，各バイトごとの最終的な内容になります。 I64 システムで初期化を行うと，セクション全体がコンパイラで初期化されます。このセクションに対する以降の初期化は，新しい初期化内容が既存の内容と同じ場合にだけ実行できます。

リンカは，I64 システム固有のさまざまな修飾子とオプションをサポートします。これらの修飾子を表 4–1 に示します。表 4–2 は，VAX システムではサポートされ，I64 システムではサポートされないリンカ修飾子を示しています。表 4–3 は，VAX システムではサポートされ，I64 システムでは無視されるリンカ修飾子を示しています。

表 4–1 OpenVMS I64 システム固有のリンカ修飾子とオプション

修飾子	説明
/DEMAND_ZERO[=PER_PAGE]	実行イメージと共有イメージの両方で，デマンド・ゼロ・イメージ・セクション (OpenVMS I64 ではセグメントと呼ばれる) を有効にする。キーワード PER_PAGE を指定すると，各セグメントの後続のゼロが圧縮される (後続ページのゼロのデマンド・ゼロ圧縮)
/DSF	OpenVMS I64 デバッガで使用するために，デバッグ・シンボル・ファイル (DSF) と呼ぶファイルを作成するようにリンカに要求する。
/FP_MODE=keyword	OpenVMS I64 リンカは，プログラムの初期浮動小数点モードを，メイン遷移アドレスを提供するモジュールの浮動小数点モードを使用して決定する。/FP_MODE 修飾子は，メイン遷移アドレスを提供するモジュールが初期浮動小数点モードを持たない場合にだけ，初期浮動小数点モードを設定するために使用する。/FP_MODE 修飾子は，メイン遷移モジュールによって提供される初期浮動小数点モードを指定変更することはない。
/FULL[=(keyword [,...])]	キーワード GROUP_SECTIONS は，マップで使用されているすべてのグループを出力する (グループは OpenVMS I64 用のオブジェクト言語の新しい概念。詳細は，『OpenVMS Linker Utility Manual』を参照)。キーワード NOSECTION_DETAILS は，OpenVMS I64 リンカが長さゼロのコントリビューションをマップのプログラム・セクション一覧に表示しない場合に指定する。デフォルトは/FULL=SECTION_DETAILS。
/GST	共有イメージのグローバル・シンボル・テーブル (GST) を作成することをリンカに要求する (デフォルト)。通常は，リンクできない共有イメージとともにアプリケーションを出荷する場合に，/NOGST として指定する。
/INFORMATIONALS	リンク操作で情報メッセージを出力することをリンカに要求する (デフォルト)。/NOINFORMATIONALS を指定する方が一般的であり，その場合には情報メッセージは出力されない。
/NATIVE_ONLY	作成しているイメージに，コンパイラが作成したプロシージャ・シグネチャ・ブロック (PSB) 情報を渡さないようにリンカに要求する (デフォルト)。 リンク時に/NONATIVE_ONLY を指定した場合には，イメージ・アクティベータは，ジャケット・ルーチンを起動するために，リンク操作に対する入力ファイルとして指定されたオブジェクト・モジュールで提供された PSB 情報を使用する。ネイティブ I64 イメージが，トランスレートされたイメージと連携動作するためには，ジャケット・ルーチンが必要である。

(次ページに続く)

表 4–1 (続き) OpenVMS I64 システム固有のリンカ修飾子とオプション

修飾子	説明
/SEGMENT_ ATTRIBUTE=(segment_attribute [, ...])	OpenVMS I64 リンカに対して，セグメントの特定の属性を設定するように指示する。 OpenVMS I64 リンカでは，セグメント属性として， DYNAMIC=address_ region， SHORT=WRITE， CODE=address_region, および SYMBOL_VECTOR=[NO]SHORT を指定できる。アドレス領域は，キーワード P0 および P2 を使用して指定できる。
/SYSEXE	リンク操作で解決されなかったシンボルを解決するために， OpenVMS エグゼクティブ・イメージ (SYS$BASE_IMAGE.EXE) を処理することをリンカに要求する。

オプション	説明
SYMBOL_TABLE=option	共有イメージに関連するシンボル・テーブル・ファイルに，ユニバーサル・シンボルだけでなく，グローバル・シンボルも登録することをリンカに要求する。デフォルトでは，リンカはユニバーサル・シンボルだけを登録する。
SYMBOL_VECTOR=option	I64 共有イメージでユニバーサル・シンボルを宣言するために使用する。

表 4–2 I64 システムではサポートされない OpenVMS VAX リンカ修飾子とオプション

修飾子	説明
/DEBUG=file_spec	/DEBUG 修飾子でオブジェクト・ファイルを指定して，ユーザ作成デバッガ・モジュールを実行時にアクティブにする機能はサポートされない。
/SYSTEM[=base_address]	システム・イメージを作成することをリンカに指示し，オプションで，イメージをメモリにロードするアドレスを指定することができる。システム・イメージを RUN コマンドでアクティブにすることはできない。ブートストラップするか，メモリにロードしなければならない。

オプション	説明
BASE=option	リンカがイメージに割り当てる基底アドレス (開始アドレス) を指定する。
CLUSTER=cluster_name, base_ address	CLUSTER オプションでは基底アドレス・キーワードはヌルでなくてはならない。ベースが指定されたイメージ (ベース・イメージ・セクションがあるイメージ) は， I64 ではサポートされない。
UNIVERSAL=option	共有イメージ内のシンボルを，ユニバーサルとして宣言する。これによりリンカは，共有イメージの GST 内に登録する。

表 4–3 I64 システムでは無視される OpenVMS VAX リンカ修飾子とオプション

修飾子	説明
/ALPHA	OpenVMS Alpha イメージを生成するようにリンカに指示する。
/HEADER	/SYSTEM 修飾子と同時に指定すると，システム・イメージにイメージ・ヘッダを含めるようにリンカに指示する。 I64 リンカは，イメージに対して/HEADER 修飾子が指定された場合は無視する。 VAX リンカと Alpha リンカは，実行イメージまたは共有イメージに対して指定された場合に無視する。
/VAX	OpenVMS VAX イメージを生成するようにリンカに指示する。

オプション	説明
DZRO_MIN=option	イメージ・セクションからページを取り出し，新しく作成したデマンド・ゼロ・イメージ・セクションに配置する前に，イメージ・セクション内でリンカが探さなくてはならない，最低限の連続した初期化されていないページの数を指定する。デマンド・ゼロ・イメージ・セクション (初期化済みのデータが格納されていないイメージ・セクション) を作成することで，リンカはイメージのサイズを削減することができる。
ISD_MAX=option	イメージ中に作成できるイメージ・セクションの最大数を指定する。

4.4 VAX システムと I64 システムの算術演算ライブラリ間の互換性

OpenVMS Mathematics (MTH$) ランタイム・ライブラリに対して標準的な VMS 呼び出しインタフェースを使用する算術演算アプリケーションを， OpenVMS I64 システムに移行するときには， MTH$ルーチンの呼び出しを変更する必要はありません。これは， MTH$ルーチンを HP Portable Mathematics Library (DPML) for OpenVMS I64 システムの対応する math$に変換するためのジャケット・ルーチンが提供されているためです。ただし， JSB エントリ・ポイントとベクタ・ルーチンに対して実行される呼び出しは， DPML でサポートされません。 DPML ルーチンは OpenVMS MTH$ RTL のルーチンと異なっており，算術演算の結果の精度にわずかな違いが発生する可能性があります。

将来のライブラリとの互換性を維持し，移植可能な算術演算アプリケーションを開発するには，この呼び出しインタフェースを使用するのではなく，選択した高級言語 (たとえば HP C や HP Fortran など) を通じて提供される DPML ルーチンを使用することをお勧めします。 DPML ルーチンを使用すれば，性能と精度も大幅に向上できます。

DPML ルーチンについての詳細は，『Compaq Portable Mathematics Library』を参照してください。

4.5 ホスト・アーキテクチャの判断

アプリケーションが OpenVMS VAX システムで実行されているのか，I64 システムで実行されているのかを，アプリケーションで判断しなければならないことがあります。プログラムの内部から$GETSYI システム・サービス (または LIB$GETSYI RTL ルーチン) を呼び出し，ARCH_TYPE 項目コードを指定すれば，この情報を入手できます。アプリケーションが VAX システムで実行されている場合には，$GETSYI システム・サービスは値 1 を返します。アプリケーションが I64 システムで実行されている場合には，$GETSYI システム・サービスは値 3 を返します。

例 4–1 は，F$GETSYI DCL コマンドを呼び出し，ARCH_TYPE 項目コードを指定することにより，DCL コマンド・プロシージャでホスト・アーキテクチャを判断する方法を示しています ($GETSYI システム・サービスを呼び出して I64 システムのページ・サイズを取得する例については，第 5.4 節を参照してください)。

例 4–1 アーキテクチャ・タイプを判断するための ARCH_TYPE キーワードの使用

```
$! Determine architecture type
$ type_symbol = f$getsyi("arch_type")
$ if type_symbol .eq. 1 then goto ON_VAX
$ if type_symbol .eq. 2 then goto ON_ALPHA
$ if type_symbol .eq. 3 then goto ON_I64
$ !
$ ! Unknown architecture
$ !
$ exit
$ ON_VAX:
$ !
$ ! Do VAX-specific processing
$ !
$ exit
$ ON_ALPHA:
$ !
$ ! Do Alpha-specific processing
$ !
$ exit
$ ON_I64:
$ !
$ ! Do I64-specific processing
$ !
$ exit
```

しかし，ARCH_TYPE 項目コードは，バージョン 5.5 またはそれ以降のバージョンを実行している VAX システムでしか使用できません。アプリケーションが，これ以前のバージョンのオペレーティング・システムでホスト・アーキテクチャを判断しなければならない場合には，表 4–4 に示した$GETSYI システム・サービスの他の項目コードを使用しなければなりません。

表 4–4 ホスト・アーキテクチャを指定する$GETSYI 項目コード

キーワード	使用方法
ARCH_TYPE	VAX システムでは 1 を返す。 Alpha システムでは 2 を返す。 I64 システムでは 3 を返す。 I64 システムと， OpenVMS バージョン 5.5 以降のバージョンの VAX システムでサポートされる。
ARCH_NAME	VAX システムでは“VAX”というテキスト文字列を返し， Alpha システムでは“Alpha”というテキスト文字列を返し， I64 システムでは“IA64”というテキスト文字列を返す。 I64 システムと， OpenVMS バージョン 5.5 以降のバージョンを実行している VAX システムでサポートされる。
HW_MODEL	ハードウェア・モデルを識別する整数を返す。値 4096 は I64 システムを示す。
CPU	特定の CPU を識別する整数を返す。 128 という値は，システムを“VAX でない”として識別する。このコードは ARCH_TYPE コードおよび ARCH_NAME コード以前の OpenVMS のバージョンでサポートされる。

5

ページ・サイズの拡大に対するアプリケーションの対応

この章では，アプリケーションで VAX のページ・サイズに依存している部分を識別する方法を説明し，これらの問題に対する対処方法を示します。

5.1 概要

ページ・サイズは，オペレーティング・システムが操作するメモリの基本単位であり，一般にアプリケーションのレベルでは意識する必要はありません。特に，高級プログラミング言語や中級プログラミング言語で作成されたアプリケーションの場合には，ページ・サイズを直接操作することはほとんどありません。しかし，アプリケーションでシステム・サービスやランタイム・ライブラリ・ルーチンを呼び出し，次のようなメモリ管理機能を実行する場合には，ページ・サイズに依存する部分がアプリケーションに含まれている可能性があります。

- 仮想メモリの割り当て
- セクションをプロセスの仮想アドレス空間にマッピングする操作
- メモリをワーキング・セット内にロックする操作
- 仮想アドレス空間のセグメントの保護

これらを実行するシステム・サービスやランタイム・ライブラリ・ルーチンは，メモリをページ単位で操作します。これらのルーチンに対する引数として指定する値では，1 ページが 512 バイトであるものと仮定しています。これは VAX アーキテクチャで定義されているページ・サイズです。OpenVMS I64 では，デフォルトのページ・サイズは 8K バイトであり，将来の OpenVMS のリリースで 16K バイト，32K バイト，または 64K バイトのページ・サイズをサポートします。この後の節では，ルーチンを調べる方法について詳しく説明します。

このようなページ・サイズの違いは，上位のルーチンを使用したメモリ割り当てには影響を与えません。たとえば，C の`malloc` ルーチンや`free`ルーチンなど，仮想メモリ領域を操作するランタイム・ライブラリ・ルーチンや，言語固有のメモリ割り当てルーチンは，ページ・サイズの違いの影響を受けません。

5.1.1 互換性機能

システム・サービスやランタイム・ライブラリ・ルーチンは，I64 システムにおいてもできるかぎり VAX システムと同じインタフェースや戻り値を維持しています。たとえば I64 システムでは，引数としてページ・カウント値を受け付けるルーチンのこれらの引数を**ページレット**と呼ぶ 512 バイトの量として解釈することにより，CPU 固有のページ・サイズと区別します。各ルーチンはページレットの値を CPU 固有のページに変換します。ページ・カウント値を返すルーチンは，CPU 固有のページからページレットに変換することにより，アプリケーションで期待される戻り値が 512 バイト単位で表現されるようにします。

注意

I64 システムでは，ページ・フレーム・セクションを作成する際に，SEC$M_PFNMAP フラグ付きで$CRMPSC ルーチンを使用することはできません。システム・サービス SYS$CREATE_GPFN，SYS$CRMPSC_GPFN_64，SYS$CRMPSC_PFN_64 のいずれかを呼び出さなければなりません。また，SEC$M_ARGS64 フラグを使用して，64 ビットのstart_pfn引数を指定したことを示さなければなりません。なお，64 ビットのシステム・サービスは，符号拡張した 32 ビット・アドレスを使用して呼び出すことができます。

5.1.2 特定のページ・サイズに依存する可能性のあるメモリ管理ルーチンのまとめ

互換性があるにもかかわらず，I64 システムでの一部のルーチンの動作は VAX システムでの動作と異なっており，ソース・コードを変更しなければならない可能性があります。たとえば I64 システムでは，セクション・ファイルをマッピングするシステム・サービス ($CRMPSC と$MGBLSC) は，CPU 固有のページ境界にアラインされたアドレス値を引数として指定しなければなりません。VAX システムでは，これらのルーチンは引数のアドレス値を VAX ページ境界になるように調整します。I64 システムでは，これらのアドレスは CPU 固有のページ境界には調整されません。

表 5–1 は，ページ・サイズに依存する部分を含んでいる可能性のあるメモリ管理ルーチンと，それらのルーチンがサポートする引数を示しています。この表には，各引数の機能と，これらの引数が I64 システムでどのように解釈されるかが示されています。この表には，ルーチンが受け付けるすべての引数が示されているわけではありません。ルーチンとその引数リストについての詳細は，『OpenVMS System Services Reference Manual』を参照してください。

表 5–1 メモリ管理ルーチンでページ・サイズに依存する可能性のある部分

ルーチン	引数	I64 システムでの動作
ワーキング・セット・リミットの調整 ($ADJWSL)	pagcntは，現在のワーキング・セット・リミットに加算される (またはワーキング・セット・リミットから減算される) ページ数を指定する。	値をページレットとして解釈し，CPU 固有のサイズのページを表現するために切り上げるか，または切り捨てる。
	wsetlmは，現在のワーキング・セット・リミットの値を示す。	値をページレットとして解釈し，CPU 固有のサイズのページを表現するために切り上げるか，または切り捨てる。
プロセスの作成 ($CREPRC)	quotaは，デフォルトのワーキング・セット・サイズ，ページング・ファイル・クォータ，ワーキング・セット拡張クォータなど，ページ・カウントを指定する複数のクォータ記述子を受け付ける。	値をページレットとして解釈し，CPU 固有のサイズのページを表現するために切り上げるか，または切り捨てる。
仮想アドレスの作成 ($CRETVA)	inadrは，割り当てられるメモリの先頭のアドレスと末尾のアドレスを指定する。末尾アドレスが先頭アドレスと同じである場合には，1 ページが割り当てられる。	CPU 固有のページ境界にアラインされるように，アドレスは切り上げられるか，または切り捨てられる。
	retadrは，呼び出しの影響を受けるメモリの実際の先頭アドレスと末尾アドレスを示す。	変更なし。
マップ・セクションの作成 ($CRMPSC)	inadrは，再マップされる領域を定義する先頭アドレスと末尾アドレスを指定する。末尾アドレスが先頭アドレスと同じである場合には，1 ページがマッピングされる。ただし，SEC$M_EXPREG フラグが設定されている場合には，割り当てが P0 空間であるか P1 空間であるかを判断し，その結果を元に先頭アドレスが解釈される。	アドレスは CPU 固有のページにアラインされていなければならない (SEC$M_EXPREG フラグが設定されていない場合)。切り上げや切り捨ては実行されない (マッピングについての詳細は，第 5.3 節を参照)。
	retadrは，呼び出しの影響を受けるメモリの実際の先頭アドレスと末尾アドレスを示す。	**使用可能な**アドレス範囲の先頭アドレスと末尾アドレスを返す。これはマッピングされた合計サイズと異なる可能性がある。relpag引数を指定した場合には，この引数も指定しなければならない。
	flagsは，作成またはマッピングされるセクションの種類と属性を指定する。	フラグ・ビット SEC$M_NO_OVERMAP は，既存のアドレス空間をマッピングしてはならないことを示す。
	relpagは，セクション・ファイルのマッピングを開始するページ・オフセットを指定する。	セクション・ファイルへのインデックスとして解釈され，単位はページレットであると解釈される。
	pagcntは，マッピングされるファイル内のページ数 (ブロック数) を指定する。	ページレットとして解釈される。切り上げや切り捨ては実行されない。

(次ページに続く)

表 5–1 (続き) メモリ管理ルーチンでページ・サイズに依存する可能性のある部分

ルーチン	引数	I64 システムでの動作
	pfcは，ページ・フォルトが発生したときにマッピングしなければならないページ数を指定する。	CPU 固有のサイズのページとして解釈される。この引数の値を指定する場合には，各物理ページに対して少なくとも 16 ページレットがマッピングされることを考慮しなければならない。これは，I64 システムが 8K バイト， 16K バイト，32Kバイト， 64K バイトの物理ページ・サイズをサポートするからである。システムでは，物理ページより小さいサイズをマッピングすることはできない。
仮想アドレスの削除 ($DELTVA)	inadrは，割り当てが解除されるメモリの先頭アドレスと末尾アドレスを指定する。	CPU 固有のページ境界にアラインされるように，アドレスは切り上げられるか，または切り捨てられる。
	retadrは，削除されたメモリの実際の先頭アドレスと末尾アドレスを示す。	変更なし。
プログラム領域/制御領域の拡張 ($EXPREG)	pagcntは， 512 バイト単位で割り当てるメモリ・サイズを指定する。	ページレット単位として解釈される。
	retadrは，呼び出しの影響を受けるメモリの実際の先頭アドレスと末尾アドレスを示す。	変更なし。
ジョブ情報/プロセス情報の取得 ($GETJPI)	itmlstは，プロセスに関して返される情報を指定する。	JPI$_WSEXTENT など，多くの項目はページレット単位の値として解釈される。詳細は『OpenVMS System Services Reference Manual』を参照。
キュー情報の取得 ($GETQUI)	itmlstは，func引数によって指定された機能を実行する際に使用する情報を指定する。	いくつかの項目はページレット単位の値として解釈される。詳細は『OpenVMS System Services Reference Manual』を参照。
システムワイド情報の取得 ($GETSYI)	itmlstは， 1 つ以上のノードに関して返される情報を指定する	一部の項目はページレット単位の値として解釈される。SYI$_PAGE_SIZEという追加項目は，ノードがサポートするページ・サイズを指定する。詳細は『OpenVMS System Services Reference Manual』を参照。
ユーザ認証情報の取得 ($GETUAI)	itmlstは，ユーザのユーザ登録ファイルからどの情報が返されるかを指定する。	一部の項目はページレット単位の値を返す。詳細は『OpenVMS System Services Reference Manual』を参照。
ロック・ページ ($LCKPAG)	inadrは，ロックされるメモリの先頭アドレスと末尾アドレスを指定する。	CPU 固有のページ境界にアラインされるように，アドレスは切り上げられるか，または切り捨てられる。

(次ページに続く)

表 5–1 (続き) メモリ管理ルーチンでページ・サイズに依存する可能性のある部分

ルーチン	引数	I64 システムでの動作
	retadrは，ロックされたメモリの実際の先頭アドレスと末尾アドレスを示す。	変更なし。
プロセス・ワーキング・セット中のロック・イメージ ($LOCK_IMAGE)	addressは，ワーキング・セット中にロックする，イメージ中のバイトのアドレスを指定する。	VAX および I64 でのみ使用可能。
グローバル・セクションのマップ ($MGBLSC)	inadrは，再マップする領域を定義する先頭アドレスと末尾アドレスを指定する。末尾アドレスが先頭アドレスと同じである場合には，1 ページがマッピングされる。ただし，SEC$M_EXPREG フラグがセットされている場合には，割り当てが P0 空間なのか P1 空間なのかを判断し，その結果に従って先頭アドレスが解釈される。	アドレスは CPU 固有のページにアラインしなければならない (SEC$M_EXPREG フラグが設定されていない場合)。アドレスの切り上げや切り捨ては実行されない (マッピングについての詳細は第 5.3 節を参照)。
	retadrは，呼び出しの影響を受けたメモリの実際の先頭アドレスと末尾アドレスを示す。	マッピングされたメモリの**使用可能な**部分の先頭アドレスと末尾アドレスを返す。
	relpagは，セクション・ファイルのマッピングを開始するページ・オフセットを指定する。	セクション・ファイルに対するインデックスとして解釈され，単位はページレットであると解釈される。
ワーキング・セットのパージ ($PURGWS)	inadrは，パージされるメモリの先頭アドレスと末尾アドレスを指定する。	CPU 固有のページ境界にアラインされるように，アドレスは切り上げられるか，または切り捨てられる。
プロテクションの設定 ($SETPRT)	inadrは，保護対象のメモリの先頭アドレスと末尾アドレスを指定する。	CPU 固有のページ境界にアラインされるように，アドレスは切り上げられるか，または切り捨てられる。
	retadrは，保護されたメモリの実際の先頭アドレスと末尾アドレスを示す。	変更なし。
ユーザ認証ファイルの設定 ($SETUAI)	itmlstは，ユーザ登録ファイルのどの情報を設定するかを指定する。	いくつかの項目はページレット単位の値として解釈される。詳細は，『OpenVMS System Services Reference Manual』を参照。
ジョブ・コントローラへ送信 ($SNDJBC)	itmlstは，func引数によって指定された機能を実行する際に使われる情報を指定する。	いくつかの項目はページレット単位の値として解釈される。詳細は『OpenVMS System Services Reference Manual』を参照。
ページのアンロック ($ULKPAG)	inadrは，アンロックするメモリの先頭アドレスと末尾アドレスを指定する。	CPU 固有のページ境界にアラインされるように，アドレスは切り上げられるか，または切り捨てられる。
	retadrは，アンロックされたメモリの実際の先頭アドレスと末尾アドレスを示す。	変更なし。

(次ページに続く)

表 5–1 (続き) メモリ管理ルーチンでページ・サイズに依存する可能性のある部分

ルーチン	引数	I64 システムでの動作
プロセス・ワーキングセットからのイメージのアンロック ($UNLOCK_IMAGE)	addressは，ワーキング・セットからアンロックするイメージ内のバイトのアドレスを指定する。	VAX および I64 でのみ使用可能。
	retadrは，アンロックされたメモリの実際の先頭アドレスと末尾アドレスを示す。	変更なし。
セクションのアップデート ($UPDSEC)	inadrは，ディスクに書き込むセクションの先頭アドレスと末尾アドレスを指定する。	CPU 固有のページになるように要求は切り上げられるか，または切り捨てられる。ディスク上の記憶領域で表現される実際のアドレス範囲だけがディスクに書き込まれる。
	retadrは，ディスクに書き込まれたメモリの実際の先頭アドレスと末尾アドレスを示す。	CPU 固有のページ境界にアラインされるように，アドレスは切り上げられるか，または切り捨てられる。

表 5–2 に示したランタイム・ライブラリ・ルーチンは，メモリ・ページを割り当てるか，または解放します。互換性を維持するために，これらのルーチンでは，ユーザが指定したページ・カウント情報をページレットの値として解釈します。

表 5–2 ランタイム・ライブラリ・ルーチンでページ・サイズに依存する可能性のある部分

ルーチン	引数	I64 システムでの動作
LIB$GET_VM_PAGE	number-of-pagesは，割り当てる連続ページのページ数を指定する。	ページレット単位の値として解釈され， CPU 固有のページに切り上げられるか，または切り捨てられる。
LIB$FREE_VM_PAGE	number-of-pagesは，割り当てを解除する連続ページのページ数を指定する。	ページレット単位の値として解釈され， CPU 固有のページに切り上げられるか，または切り捨てられる。

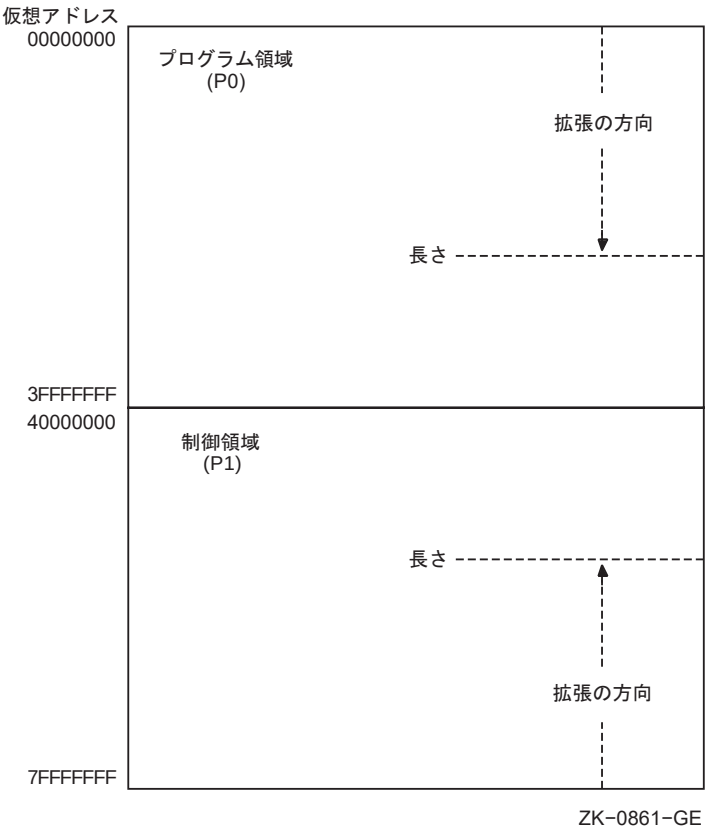
5.2 メモリ割り当てルーチンの確認

アプリケーションが実行するメモリ割り当てを変更しなければならないかどうかを判断するには，メモリがどこで割り当てられるかを確認しなければなりません。メモリ割り当てを実行するシステム・サービス・ルーチン ($EXPREG と$CRETVA) を使用すると，次の 2 種類の方法でメモリを割り当てることができます。

- アプリケーションの仮想アドレス空間の P0 領域または P1 領域のサイズを拡張する方法
- 指定した位置から始まる，アプリケーションの既存の仮想アドレス空間の領域を再要求する方法

Intel Itanium アーキテクチャでは， VAX アーキテクチャと同じ仮想アドレス空間レイアウトを定義しており， VAX システムの場合と同じ方向に P0 領域と P1 領域を拡大できます。図 5–1 はこのレイアウトを示しています。

図 5–1 仮想アドレスのレイアウト



5.2.1 拡張された仮想アドレス空間でのメモリの割り当て

アプリケーションで$EXPREG システム・サービスを使用して仮想アドレス空間を拡張することによりメモリを割り当てる場合には，ソース・コードを変更する必要はありません。これは， VAX システムで引数として指定した値が I64 システムでも正しく動作するからです。この理由は次のとおりです。

- I64 システムでは，$EXPREG システム・サービスは要求されたメモリのサイズ (pagcnt引数でページ・カウントとして指定した値) を 512 バイト単位と解釈します。これは VAX システムの場合と同じです。したがって，アプリケーションで指定した値は同じサイズのメモリを要求します。ただし，システム・サービスはページ・カウントを CPU 固有のページに切り上げるため，アプリケーションに対してシステムが実際に割り当てるメモリ・サイズは， VAX システムの場合より I64

システムの場合の方が大きくなる可能性があります。割り当てられたメモリ全体はアプリケーションで使用できます。アプリケーションは通常，必要なバッファを確保するためにメモリを割り当てます。しかし，バッファのサイズは各プラットフォームで変化しないため，指定した値はアプリケーションの必要条件を満足できます。

- 割り当ては仮想アドレス空間の拡張された領域で実行されるため，要求したサイズとシステムが実際に割り当てたサイズに違いがあっても，アプリケーションの機能に影響を与えません。

対処方法

アプリケーションを変更する必要はありません。しかし，$EXPREG システム・サービスが返すメモリ・サイズは， Intel Itanium アーキテクチャを実現した各システムで異なる可能性があるため，システムが割り当てた正確なメモリ範囲を確認しておくことをお勧めします。正確なメモリ範囲を確認するには，$EXPREG システム・サービスに対して省略可能な引数であるretadr引数を指定します(アプリケーションでこの引数がまだ指定されていない場合)。retadr引数には，システム・サービスが割り当てたメモリの先頭アドレスと末尾アドレスが格納されます。

たとえば，例 5–1 のプログラムは，$EXPREG システム・サービスを呼び出すことにより 10 ページの追加メモリを要求します。このプログラムを VAX システムで実行した場合には，$EXPREG システム・サービスは 5120 バイトの追加メモリを割り当てます。このプログラムを I64 システムで実行した場合には，$EXPREG システム・サービスは少なくとも 8192 バイトを割り当てます。また， Intel Itanium アーキテクチャの特定の実装でのページ・サイズによっては，それ以上のサイズのメモリを割り当てることもあります。

例 5–1 仮想アドレス空間の拡張によるメモリの割り当て

```
#include  <ssdef.h>
#include  <stdio.h>
#include  <stsdef.h>
#include  <descrip.h>
#include  <dvidef.h>

#define  PAGE_COUNT 10  1
#define  P0_SPACE   0
#define  P1_SPACE   1

main( argc, argv )
int argc;
char *argv[];
{
     int    status = 0;
     long   bytes_allocated, addr_returned[2];

2    status = SYS$EXPREG( PAGE_COUNT, &addr_returned, 0, P0_SPACE);

     bytes_allocated = addr_returned[1] - addr_returned[0];
```

(次ページに続く)

例 5–1 (続き) 仮想アドレス空間の拡張によるメモリの割り当て

```
    if( status == SS$_NORMAL)
       printf("bytes allocated = %d\n", bytes_allocated );
    else
       return (status);
}
```

以下の項目は，例 5–1 に示した番号に対応します。

1 この例では，要求するページ数を意味するシンボルとして PAGE_COUNT を定義しています。

2 この例では，仮想アドレス空間の P0 領域の末尾に 10 ページを追加することを要求しています。

5.2.2 既存の仮想アドレス空間でのメモリの割り当て

アプリケーションで$CRETVA システム・サービスを使用することにより，仮想アドレス空間内のメモリを再割り当てする場合には，$CRETVA に対する引数のうち，次の引数の値を変更しなければならない場合があります。

- VAX ページ境界にアラインするために，inadr引数に指定するアドレスを 512 の倍数になるように明示的に調整している場合には，アドレスを変更しなければなりません。 I64 システムでは，$CRETVA システム・サービスが先頭アドレスを CPU 固有のページ境界に合わせて切り捨てますが，この値は各システムで異なります。
- inadr引数にアドレス範囲指定によって再割り当てされるサイズは， I64 システムでは VAX システムの場合より大きくなる可能性があります。これは，要求が CPU 固有のページ・サイズに合わせて切り上げられるためです。この結果，隣接データが破壊される可能性があります。これは 1 ページを割り当てる場合でも発生します (inadr引数に指定した先頭アドレスと末尾アドレスが一致する場合には，1 ページが割り当てられます)。

対処方法

アプリケーションを変更しなければならないかどうかを判断するには，次のことを行ってください。

- 可能性のあるすべてのページ・サイズに対して，仮想アドレス空間の中で呼び出しの影響を受ける領域が重要なデータを破壊しないことを確認してください。
- 可能性のあるすべてのページ・サイズに対して，割り当てが開始される先頭アドレスが常にページ境界にアラインされることを確認してください。
- 省略可能なretadr引数がアプリケーションで指定されていない場合には，この引数を指定して，$CRETVA システム・サービスに対する呼び出しで割り当てられた正確なメモリの範囲を判断してください。

例 5–2 は，バッファに割り当てたメモリを$CRETVA システム・サービスによって再割り当てする方法を示しています。

例 5–2 既存のアドレス空間でのメモリの割り当て

```
#include  <ssdef.h>
#include  <stdio.h>
#include  <stsdef.h>
#include  <descrip.h>
#include  <dvidef.h>

char _align(page) buffer[1024];

main( argc, argv )
int argc;
char *argv[];
{

     int      status = 0;
     long     inadr[2];
     long     retadr[2];

     inadr[0] = &buffer[0];
     inadr[1] = &buffer[1023];

     printf("inadr[0]=%u,inadr[1]=%u\n",inadr[0],inadr[1]);

     status = SYS$CRETVA(inadr, &retadr, 0);

     if( status & STS$M_SUCCESS )
     {
        printf("success\n");
        printf("retadr[0]=%u,retadr[1]=%u\n",retadr[0],retadr[1]);
     }
     else
     {
        printf("failure\n");
        exit(status);
     }
}
```

5.2.3 仮想メモリの削除

$EXPREG システム・サービスと$CRETVA システム・サービスによって割り当てたメモリを解除するために$DELTVA システム・サービスを呼び出す場合，retadr引数で返されたアドレス範囲 (メモリを割り当てるために使用したルーチンから返された値) を$DELTVA システム・サービスのinadr引数として使用しているときは，アプリケーションを変更する必要はありません。実際に割り当てられるサイズは各システムで異なるため，割り当ての範囲に関してアプリケーションで何らかの仮定を行うことは望ましくありません。

5.3 メモリ・マッピング・ルーチンの確認

アプリケーションで実行するメモリ・マッピングを変更しなければならないかどうかを判断するには，アプリケーションが仮想メモリのどの部分でマッピングを実行するかを確認しなければなりません。メモリ・マッピング・システム・サービス ($CRMPSC と$MGBLSC) を使用すると，次の方法でメモリをマッピングできます。

- アプリケーションの仮想アドレス空間の拡張領域にメモリをマッピングする方法
- 指定した位置からはじまるアプリケーションの仮想アドレス空間に，メモリの 1 ページをマッピングする方法 (この位置は既存の仮想アドレス空間に存在してもかまいません)
- 仮想アドレス空間の中の指定した先頭アドレスと末尾アドレスによって定義される既存の領域に，メモリをマッピングする方法

アプリケーションがセクションをマッピングする方法は，主に$CRMPSC システム・サービスと$MGBLSC システム・サービスに対する次の引数によって決まります。

- inadr引数は，セクションのサイズと位置を先頭アドレスと末尾アドレスによって指定します。$CRMPSC システム・サービスはこの引数を次の方法で解釈します。
  - inadr引数に指定した 2 つのアドレスがどちらも同じであり，flags引数の SEC$M_EXPREG ビットがオンの場合には，指定したアドレスが含まれるプログラム領域でメモリが割り当てられますが，指定した位置は使用されません。
  - inadr引数に指定されているアドレスがどちらも同じであり， SEC$M_EXPREG フラグがオフの場合には，指定した位置を先頭アドレスとして 1 ページがマッピングされます ($CRMPSC システム・サービスのこの動作モードは I64 システムではサポートされません。アプリケーションでこのモードを使用している場合には，ソース・コードの変更方法に関して第 5.3.2 項を参照してください)。
  - 2 つのアドレスが異なる場合には，指定した境界を使用して，セクションがメモリにマッピングされます。
- pagcnt (ページ・カウント) 引数は，セクション・ファイルからマッピングするブロック数を指定します。
- relpag (相対ページ番号) 引数は，セクション・ファイルの中でマッピングを開始する位置を指定します。

$CRMPSC システム・サービスと$MGBLSC システム・サービスは，少なくとも CPU 固有のページを 1 ページ分マッピングします。セクション・ファイルが 1 ページ未満の場合には，ページの残りの部分には 0 が格納されます。ページの残りの領域はアプリケーションで使用しないでください。なぜなら，セクション・ファイルに格納できるデータだけがディスクに書き戻されるためです。

5.3.1 拡張した仮想アドレス空間へのマッピング

アプリケーションの仮想アドレス空間の拡張領域にセクション・ファイルをマッピングしている場合には，ソース・コードを変更する必要はありません。これは，拡張された仮想アドレス空間にマッピングされるため，たとえ I64 システムで割り当てられるメモリのサイズが VAX システムより大きくても，既存のデータの上にマッピングされる危険性がないためです。そのため， VAX システムで$CRMPSC システム・サービスに対して引数として指定した値は， I64 システムでも正しく機能します。

対処方法

セクションを仮想メモリの拡張領域にマッピングするアプリケーションは，変更しなくても正しく動作しますが，アプリケーションでretadr引数を指定していない場合には，この引数を指定し，呼び出しによってマッピングされたメモリの正確な範囲を確認してください。

注意

アプリケーションでrelpag**引数を指定する場合には，**retadr**引数も指定しなければなりません。これは省略可能な引数ではありません。**relpag**引数の使用についての詳細は，第 5.3.4 項を参照してください。**

例 5–3 は，セクション・ファイルを拡張アドレス空間にマッピングする$CRMPSC システム・サービスの呼び出しを示しています。この例では，次に示すように DCL の CREATE コマンドを使用して作成された MAPTEST.DAT という名前のセクション・ファイルをマッピングします。

```
$  CREATE maptest.dat
test data test data test data test data test data
test data test data test data test data test data
test data test data test data test data test data
test data test data test data test data test data
test data test data test data test data test data
test data test data test data test data test data
test data test data test data test data test data
test data test data test data test data test data
Ctrl/Z
```

例 5–3 拡張された仮想アドレス空間へのセクションのマッピング

(次ページに続く)

例 5–3 (続き) 拡張された仮想アドレス空間へのセクションのマッピング

```
#include  <ssdef.h>
#include  <string.h>
#include  <stdlib.h>
#include  <stdio.h>
#include  <stsdef.h>
#include  <descrip.h>
#include  <dvidef.h>
#include  <rms.h>
#include  <secdef.h>

struct FAB fab;

char _align(page) buffer[1024];
char *filename = "maptest.dat";

main( argc, argv )
int argc;
char *argv[];
{
     int    status = 0;
     long   flags = SEC$M_EXPREG;
     long   inadr[2];
     long   retadr[2];
     int    fileChannel;

/********  create disk file to be mapped *************/

     fab = cc$rms_fab;
     fab.fab$l_fna = filename;
     fab.fab$b_fns = strlen( filename );
     fab.fab$l_fop = FAB$M_CIF | FAB$M_UFO;  /* must be UFO */

     status = sys$create( &fab );

     if( status & STS$M_SUCCESS )
        printf("%s opened\n",filename);
     else
     {
        exit( status );
     }

     fileChannel = fab.fab$l_stv;
```

(次ページに続く)

例 5–3 (続き) 拡張された仮想アドレス空間へのセクションのマッピング

```
/**********  create and map the section  ***************/

     inadr[0] = &buffer[0];
     inadr[1] = &buffer[0];

     status = SYS$CRMPSC( inadr, /* inadr=address target for map */
                        &retadr, /* retadr= what was actually mapped */
                              0, /* acmode  */
                          flags, /* flags, with SEC$M_EXPREG bit set */
                              0, /* gsdnam, only for global sections */
                              0, /* ident, only for global sections */
                              0, /* relpag, only for global sections */
                    fileChannel, /* returned by SYS$CREATE */
                              0, /* pagcnt = size of sect. file used */
                              0, /* vbn = first block of file used */
                              0, /* prot = default okay */
                              0); /* page fault cluster size */

     if( status & STS$M_SUCCESS )
     {
          printf("section mapped\n");
          printf("retadr[0]=%u,retadr[1]=%u\n",retadr[0],retadr[1]);
     }
     else
     {
          printf("map failed\n");
          exit( status );
     }

}
```

5.3.2 特定の位置への単一ページのマッピング

アプリケーションでセクション・ファイルを 1 ページのメモリにマッピングしていた場合には，I64 システムではこの操作モードがサポートされないため，ソース・コードを変更しなければなりません。I64 システムのページ・サイズは VAX システムのページ・サイズと異なっており，さらに Intel Itanium アーキテクチャの実装ごとにも異なるため，セクション・ファイルをマッピングするための正確なメモリ境界を指定しなければなりません。このような使い方をした場合，$CRMPSC システム・サービスは，引数が誤っていることを示すエラー (SS$_INVARG) を返します。

アプリケーションでこのモードを使用しているかどうかを判断するには，inadr引数に指定した先頭アドレスと末尾アドレスを確認します。両方のアドレスが同じであり，flags引数の SEC$M_EXPREG ビットがオフの場合には，アプリケーションはこのモードを使用しています。

対処方法

このモードの$CRMPSC システム・サービスの呼び出しを変更する場合には，次のガイドラインに従ってください。

- マッピングの宛先となる位置が重要でない場合には，flags引数の SEC$M_EXPREG ビットをオンにし，システム・サービスがアプリケーションの仮想アドレス空間の拡張領域にセクション・ファイルをマッピングするようにしてください。この操作モードについての詳細は，第 5.3.1 項を参照してください。
- マッピングの宛先となる位置が重要な場合には，inadr引数の先頭アドレスと末尾アドレスの両方を定義し，定義した領域にセクションをマッピングしてください。このモードについての詳細は，第 5.3.3 項参照してください。

5.3.3 定義されたアドレス範囲へのマッピング

アプリケーションで仮想アドレス空間の定義された領域にセクションをマッピングしている場合には，ソース・コードを変更しなければならない可能性があります。これは， I64 システムでは$CRMPSC システム・サービスおよび$MGBLSC システム・サービスが VAX システムと異なる方法で一部の引数を解釈するためです。相違点は次のとおりです。

- inadr引数に指定する先頭アドレスは， CPU 固有のページ境界にアラインされていなければならず，指定する末尾アドレスも CPU 固有のページの末尾にアラインされていなければなりません。 VAX システムでは，$CRMPSC システム・サービスおよび$MGBLSC システム・サービスは，これらのアドレスを調整して，ページ境界にアラインされるようにします。 I64 システムでは，このようなアドレスの調整は実行されません。これは，ページ・サイズがはるかに大きいため， CPU 固有のページ境界にアドレスを調整すると，メモリのより大きな部分に影響があるからです。したがって， I64 システムでは，仮想メモリ空間のどこにマッピングするかを明示的に指定しなければなりません。指定したアドレスが CPU 固有のページ境界にアラインされていない場合には，$CRMPSC システム・サービスは，引数が誤っていることを示すエラー (SS$_INVARG) を返します。
- retadr引数に返されるアドレスは，呼び出しで実際にマッピングされたメモリの中で使用可能な部分だけを反映し，マッピングされたメモリ全体を反映するわけではありません。使用可能なサイズとは，pagcnt引数に指定した値 (ページレット単位の値) と，セクション・ファイルのサイズのうち，どちらか小さい方の値です。実際にマッピングされるサイズは，セクション・ファイルをマッピングするために CPU 固有のページが何ページ必要であるかに応じて変わります。セクション・ファイルが CPU 固有のページより小さい場合には，ページの残りの部分に 0 が挿入されます。このページの残りの空間をアプリケーションで使用しないでください。retadr引数で指定する末尾アドレスは，アプリケーションで使用できる上限を指定します。また，relpag引数を指定するときはretadr引数も指定しなければなりません。 I64 システムでは，この引数は VAX システムでのように省略可能ではありません。詳細は，第 5.3.4 項を参照してください。

対処方法

可能な場合には，拡張された仮想アドレス空間にデータをマッピングするように，アプリケーションを変更してください。アプリケーションがデータをマッピングする方法を変更できない場合には，次のガイドラインに従ってください。

- オペレーティング・システムは少なくとも 1 物理ページをマップします。 I64 システム上の物理ページのサイズは VAX システムより大きいため，アプリケーションの中で定義したバッファにセクションをマップする場合，隣接するデータが上書きされないよう注意してください。多くの VAX システム上のアプリケーションでは，たとえマップされるセクション・ファイルのサイズが 512 バイト以下でも，セクションがマップされるバッファのサイズを VAX システムのページ・サイズである 512 バイト単位で定義しています。 I64 でこの方針に従うためには，メモリが無駄になりますが，アプリケーションでバッファのサイズを 64K バイト単位で宣言してください。

  セクションをマップするときに，隣接データが重ね書きされないことを確認するためのよりよい方法は，バッファを独立したイメージ・セクションに格納するようにリンカに指示することです。(リンカはイメージをイメージ・セクションから作成します。それぞれのイメージ・セクションは，イメージの各部分のメモリの必要量を定義しています。) リンカはイメージ・セクションをページ境界に配置し，隣接するデータは次のページ境界から配置します。このため，バッファを個別のイメージ・セクションに分離することで，マッピング操作によって隣接データが重ね書きされないことが保証されます。このように，隣接するデータを破壊したり，バッファのサイズを変更することなく， 1 ページ分のメモリをセクションにマップすることが可能です。

  リンカがセクション・ファイルを個別のイメージ・セクションに置いたことを確認するために，リンカの PSECT_ATTR=オプションを使用して，プログラム・セクションの SOLITARY 属性を設定しなければなりません (詳細は，『OpenVMS Linker Utility Manual』を参照してください)。また，使用している高級または中級のプログラミング言語の機能を使用して，定義したバッファが個別のプログラム・セクションに配置されていることを確認しなければならない場合があります。詳細は，コンパイラのマニュアルを参照してください。

- $CRMPSC システム・サービスと$MGBLSC システム・サービスの引数として指定するセクションの先頭アドレスおよび末尾アドレスが， CPU 固有のページの先頭アドレスおよび末尾アドレスにアラインされることを確認してください。 VAX システムでは，ページ境界に合わせてアドレスが調整されます。 I64 システムでは，ユーザが指定したアドレスはページ境界に合わせて調整されません。

  SOLITARY プログラム・セクション属性を使用して，セクションを個別のイメージ・セクションに分離すると，先頭アドレスはページ境界にアラインされます。これは，実行時のホスト・マシンのページ・サイズにかかわらず，リンカがデフォルトでイメージ・セクションをページ境界にアラインするためです。

セクションの末尾アドレスがCPU固有のページ境界にアラインされていることを確認するためには，アプリケーションを実行するマシンがサポートしているページ・サイズを知る必要があります。$GETSYIシステム・サービスやLIB$GETSYIランタイム・ライブラリ・ルーチンを呼ぶことにより，実行時にCPU固有のページ・サイズを得ることができます。得た値を使ってアラインされた末尾アドレスの値を計算し，システム・サービスのinadr引数に渡すことができます。

システムがマップした使用可能なメモリ量を確認するためには，retadr引数を指定してください。アプリケーションがページの一部分しか使用しない場合でも，オペレーティング・システムは最低でも1ページをマップします。retadr引数が示す末尾アドレスは，使用できるメモリの上限を示します。(I64システムでは，$CRMPSCシステム・サービスでrelpag引数を指定する場合，必ずretadr引数も指定しなければなりません。)

たとえば，例5–4に示すVAXプログラムは，第5.3.1項で作成したセクション・ファイルを既存の仮想アドレス空間にマッピングします。アプリケーションはbufferという名前のバッファを定義します。このバッファのサイズは512バイトであり，VAXのページ・サイズを反映しています。プログラムはバッファの1バイト目のアドレスを先頭アドレスとして，また，バッファの最終バイトのアドレスを末尾アドレスとしてinadr引数に渡すことにより，セクションの正確な境界を定義します。

例 5–4 仮想アドレス空間の定義された領域へのセクションのマッピング

```
#include <ssdef.h>
#include <stdio.h>
#include <stsdef.h>
#include <descrip.h>
#include <dvidef.h>
#include <rms.h>
#include <secdef.h>

struct FAB fab;

char *filename = "maptest.dat";

char _align(page) buffer[512];

main( argc, argv )
int argc;
char *argv[];
{
     int    status = 0;
     long   flags = 0;
     long   inadr[2];
     long   retadr[2];
     int    fileChannel;
```

(次ページに続く)

例 5–4 (続き) 仮想アドレス空間の定義された領域へのセクションのマッピング

```
/********  create disk file to be mapped *************/

     fab = cc$rms_fab;
     fab.fab$l_fna = filename;
     fab.fab$b_fns = strlen( filename );
     fab.fab$l_fop = FAB$M_CIF | FAB$M_UFO;  /* must be UFO */

     status = sys$create( &fab );

     if( status & STS$M_SUCCESS )
        printf("Opened mapfile %s\n",filename);
     else
     {
        printf("Cannot open mapfile %s\n",filename);
        exit( status );
     }

     fileChannel = fab.fab$l_stv;

/**********  create and map the section  ***************/

     inadr[0] = &buffer[0];
     inadr[1] = &buffer[511];

     printf("inadr[0]=%u,inadr[1]=%u\n",inadr[0],inadr[1]);

     status = SYS$CRMPSC(inadr, /* inadr=address target for map */
                         &retadr, /* retadr= what was actually mapped */
                               0, /* acmode  */
                               0, /* flags */
                               0, /* gsdnam, only for global sections */
                               0, /* ident, only for global sections */
                               0, /* relpag, only for global sections */
                      fileChannel, /* returned by SYS$CREATE */
                               0, /* pagcnt = size of sect. file used */
                               0, /* vbn = first block of file used */
                               0, /* prot = default okay */
                               0 ); /* page fault cluster size */

     if( status & STS$M_SUCCESS )
     {
           printf("Map succeeded\n");
           printf("retadr[0]=%u,retadr[1]=%u\n",retadr[0],retadr[1]);
     }
     else
     {
           printf("Map failed\n");
           exit( status );
     }

}
```

例 5–4 に示したプログラムを I64 システムで正しく実行するには，次の変更が必要です。

- inadr引数に指定するセクションの先頭アドレスがI64のページ境界にアラインされていることと，指定する末尾アドレスがI64のページの末尾にアラインされていることを確認しなければなりません。
- I64システムの大きいページをマッピングするときに，隣接データが上書きされないことを確認しなければなりません。

これらの目標を達成するための1つの方法として，SOLITARYプログラム・セクション属性を使用することにより，セクション・データを格納したプログラム・セクションを個別のイメージ・セクションに分離する方法があります。

この例では，bufferという名前のセクションが，bufferという名前のプログラム・セクション内に示されています。(プログラム・セクションの生成方法は，各プラットフォーム上のプログラミング言語によって異なります。セクションが個別のプログラム・セクションにあることを確認する方法については，コンパイラのマニュアルを参照してください。)次のリンク操作は，このプログラム・セクションのSOLITARY属性を設定する方法を示しています。

```
$  LINK MAPTEST, SYS$INPUT/OPT
PSECT_ATTR=BUFFER,SOLITARY
Ctrl/Z
```

CPU固有のページの末尾にアラインされる末尾アドレスをセクション・バッファに対して指定するには，実行時にCPU固有のページ・サイズを確認し，その値から1を減算し，その値を使用して配列の最終要素のアドレスを求めます。この値をinadr引数の2番目のロングワードとして渡します(実行時にページ・サイズを確認する方法については，第5.4節を参照してください)。セクションがマッピングされるバッファの割り当てを変更する必要はありません。

アプリケーションが任意のページ・サイズのI64システムで正しく実行されるようにするには，/BPAGE=16修飾子を指定することにより，リンカがイメージ・セクションを64KBの境界に強制的にアラインするようにします。実際にマッピングされるメモリの総量は，使用可能なメモリの合計よりはるかに大きくなる可能性があります。使用可能なメモリのサイズは，ページ・カウント引数(pagcnt)の値とセクション・ファイルのサイズのうち，どちらか小さい方の値によって決まります。セクションの範囲内に含まれないメモリを使用しないようにするには，retadr引数に返された値を使用します。

例5–5は，I64システムで正しく実行するために例5–4に対して必要なソースの変更を示しています。

例5–5　例5–4をI64システムで実行するのに必要なソース・コードの変更

(次ページに続く)

例 5–5 (続き) 例 5–4 を I64 システムで実行するのに必要なソース・コードの変更

```
#include  <ssdef.h>
#include  <stdio.h>
#include  <stsdef.h>
#include  <string.h>
#include  <stdlib.h>
#include  <descrip.h>
#include  <dvidef.h>
#include  <rms.h>
#include  <secdef.h>
#include  <syidef.h> 1

char buffer[512];  2
char *filename = "maptest.dat";
struct FAB fab;

long  cpu_pagesize; 3

struct itm {                      /* item list */
    short int     buflen;  /* length of buffer in bytes */
    short int  item_code;  /* symbolic item code */
    long          bufadr;  /* address of return value buffer */
    long       retlenadr;  /* address of return value buffer length */
  } itmlst[2]; 4

main( argc, argv )
int argc;
char *argv[];
{
     int    i;
     int    status = 0;
     long   flags = SEC$M_EXPREG;
     long   inadr[2];
     long   retadr[2];
     int    fileChannel;
     char  *mapped_section;

/********  create disk file to be mapped *************/

     fab = cc$rms_fab;
     fab.fab$l_fna = filename;
     fab.fab$b_fns = strlen( filename );
     fab.fab$l_fop = FAB$M_CIF | FAB$M_UFO;  /* must be UFO */

     status = sys$create( &fab );

     if( status & STS$M_SUCCESS )
        printf("%s opened\n",filename);
     else
     {
        exit( status );
     }

     fileChannel = fab.fab$l_stv;
```

(次ページに続く)

例 5–5 (続き) 例 5–4 を I64 システムで実行するのに必要なソース・コードの変更

```
/**********  obtain the page size at run time  ***************/

     itmlst[0].buflen =  4;
     itmlst[0].item_code = SYI$_PAGE_SIZE;
     itmlst[0].bufadr =  &cpu_pagesize;
     itmlst[0].retlenadr = &cpu_pagesize_len;
     itmlst[1].buflen = 0;
     itmlst[1].item_code = 0;

5    status = sys$getsyiw( 0, 0, 0, &itmlst, 0, 0, 0 );

     if( status & STS$M_SUCCESS )
     {
          printf("getsyi succeeds, page size = %d\n",cpu_pagesize);
     }
     else
     {
          printf("getsyi fails\n");
          exit( status );
     }

/**********  create and map the section  ***************/

     inadr[0] = &buffer[0];
     inadr[1] = &buffer[cpu_pagesize - 1]; 6

     printf("address of buffer = %u\n", inadr[0] );

     status = SYS$CRMPSC(&inadr, /* inadr=address target for map */
                         &retadr, /* retadr= what was actually mapped */
                               0, /* acmode  */
                               0, /* no flags to set  */
                               0, /* gsdnam, only for global sections */
                               0, /* ident, only for global sections */
                               0, /* relpag, only for global sections */
                      fileChannel, /* returned by SYS$CREATE */
                               0, /* pagcnt = size of sect. file used */
                               0, /* vbn = first block of file used */
                               0, /* prot = default okay */
                               0); /* page fault cluster size */

     if( status & STS$M_SUCCESS )
     {
          printf("section mapped\n");
          printf("start address returned =%u\n",retadr[0]);
     }
     else
     {
          printf("map failed\n");
          exit( status );
     }
}
```

以下の各項目は，例 5–5 の番号に対応しています。

1 ヘッダ・ファイル SYIDEF.H には，$GETSYI システム・サービスに対する OpenVMS 項目コードの定義が含まれています。

2 バッファは_align(page) ストレージ記述子を使用せずに定義されています。ページ・サイズは OpenVMS I64 システムで実行するまで決まらないため， HP C for OpenVMS I64 コンパイラは，_align(page) が指定されているときに，データを I64 の最大ページ・サイズ (64KB) にアラインします。

3 この構造体は，実行時にページ・サイズを取得するために使用する項目リストを定義しています。

4 この変数には，返されたページ・サイズ値が格納されます。

5 $GETSYI システム・サービスに対するこの呼び出しで，実行時にページ・サイズを取得します。

6 バッファの末尾アドレスは，返されたページ・サイズ値から 1 を減算することで指定します。

5.3.4 オフセットによるセクション・ファイルのマッピング

アプリケーションではセクション・ファイルの一部だけをマッピングできます。その場合には，マッピングを開始するアドレスをセクション・ファイルの先頭からのオフセットとして指定します。このオフセットを指定するには，$CRMPSC システム・サービスのrelpag引数に対して値を指定します。relpag引数の値は，ファイルの先頭を基準にしてマッピングを開始するページ番号を指定します。

$CRMPSC システム・サービスは，互換性を維持するために， VAX システムと I64 システムの両方のシステムにおいて，relpag引数の値を 512 バイト単位で解釈します。しかし， I64 システムの CPU 固有のページ・サイズは 512 バイトより大きいため，relpag引数にオフセットとして指定する値は，おそらく CPU 固有のページ境界にアラインされません。$CRMPSC システム・サービスは仮想メモリを CPU 固有のページ単位でのみマッピングできます。したがって， I64 システムでは，セクション・ファイルのマッピングは，オフセットによって指定されるアドレスからではなく，オフセット・アドレスを含む CPU 固有のページの先頭から開始されます。

注意

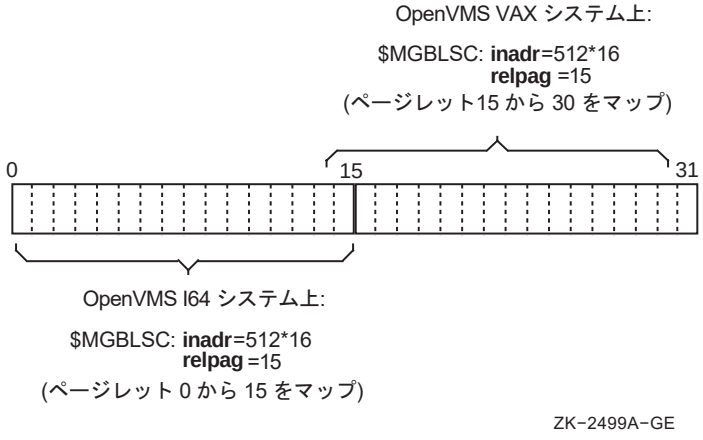
ルーチンは，オフセットによって指定されるアドレスを含む CPU 固有のページの先頭からマッピングを開始しますが，retadr引数に返される先頭アドレスはオフセットによって指定されたアドレスであり，実際にマッピングが開始されたアドレスではありません。

アプリケーションでオフセットからセクション・ファイルにマッピングする場合には， I64 システムでマッピングされる余分な仮想メモリ空間を格納できるように，inadr引数で指定するアドレス範囲のサイズを拡大する必要があります。指定した

アドレス範囲が小さすぎると，アプリケーションはセクション・ファイルの中で必要な部分全体をマッピングできない可能性があります。これは，マッピングがセクション・ファイルのより先頭に近い開始アドレスから開始されるためです。

たとえば，VAX システムでセクション・ファイルをマッピングするときに，ブロック番号 15 から始まる 16 ブロックをマッピングする場合には，アドレス範囲として 16*512 バイトのサイズをinadr引数に指定し，relpag引数に対して 15 を指定します。これと同じマッピングを I64 システムで実行するには，ページ・サイズの違いを考慮しなければなりません。たとえば，8K バイト・ページ・サイズの I64 システムでは，relpagオフセットによって指定されるアドレスは，図 5–2 に示すように，15 ページレットを CPU 固有の 1 ページに格納できます。I64 システムでは，$CRMPSC システム・サービスはセクション・ファイルのマッピングを CPU 固有のページ境界から開始するため，16 番目から 30 番目までのブロックを正しくマッピングできません。マッピングを正しく実行するためには，I64 システムで$CRMPSC システム・サービス (または$MGBLSC システム・サービス) がマッピングする追加の 15 ページレットを格納できるようにアドレス範囲のサイズを拡大しなければなりません。サイズを拡大しなかった場合には，指定したセクション・ファイルの中で 1 ブロックだけしかマッピングされません。

図 5–2 オフセットによるマッピングでアドレス範囲に与える影響



relpag引数に指定するアドレス範囲をどれだけ拡大するかを計算する場合には，次の式を使用すると便利です。この式は，特定の数のページレットをマップするのに充分な CPU 固有のページ数を計算します。

$$\frac{(\text{マップするページレットの数} + (2 * \text{ページ当たりのページレット}) - 2)}{\text{ページ当たりのページレット}}$$

たとえば，この式を使用すれば，前の例に指定したアドレス範囲をどれだけ拡大すればよいかを計算できます。次の式では，ページ・サイズは 8K であると仮定しています。したがって，**ページ当たりのページレット**は 16 になります。

```
16+((2x16)-2)/16=2.87...
```

結果をもっとも近い整数に切り捨てることにより，この計算はinadr引数に指定するアドレス範囲が， CPU 固有のページの 2 ページを含まなければならないことを示しています。

5.4 ページ・サイズの実行時確認

I64 システムでサポートされるページ・サイズを確認するには，$GETSYI システム・サービスを使用します。例 5–6 は，このシステム・サービスを使用して実行時にページ・サイズを取得する方法を示しています。

例 5–6 CPU 固有のページ・サイズを確認するための$GETSYI システム・サービスの使用

```
#include  <ssdef.h>
#include  <stdio.h>
#include  <stsdef.h>
#include  <descrip.h>
#include  <dvidef.h>
#include  <rms.h>
#include  <secdef.h>
#include  <syidef.h>  /* defines page size item code symbol */

struct itm {              /* define item list                       */
    short int     buflen;  /* length in bytes of return value buffer */
    short int  item_code;  /* item code                              */
    long          bufadr;  /* address of return value buffer         */
    long       retlenadr;  /* address of return value length buffer  */
  } itmlst[2];

long  cpu_pagesize;
long  cpu_pagesize_len;
```

(次ページに続く)

例 5–6 (続き) CPU 固有のページ・サイズを確認するための$GETSYI システム・サービスの使用

```
main( argc, argv )
int argc;
char *argv[];
{
     int    status = 0;

     itmlst[0].buflen =  4;               /*  page size requires 4 bytes  */
     itmlst[0].item_code = SYI$_PAGE_SIZE; /*  page size item code        */
     itmlst[0].bufadr =  &cpu_pagesize;    /*  address of ret_val buffer   */
     itmlst[0].retlenadr = &cpu_pagesize_len; /* addr of length of ret_val */
     itmlst[1].buflen = 0;
     itmlst[1].item_code = 0;   /* Terminate item list with longword of 0  */

     status = sys$getsyiw( 0, 0, 0, &itmlst, 0, 0, 0 );

     if( status & STS$M_SUCCESS )
     {
          printf("getsyi succeeds, page size = %d\n",cpu_pagesize);
          exit( status );
     }
     else
     {
          printf("getsyi fails\n");
          exit( status );
     }
}
```

5.5 メモリをワーキング・セット内にロックする操作

$LKWSET システム・サービスは，VAX システムでのアドレス範囲としてinadr引数で指定したページ範囲をワーキング・セット内にロックします。このシステム・サービスは，必要に応じて，アドレスを CPU 固有のページ境界に調整します。

OpenVMS Alpha および I64 では，新しい LIBRTL ルーチン LIB$LOCK_IMAGE が同等の機能を提供します。コードと関連するデータをワーキング・セット内にロックするには，LIB$LOCK_IMAGE および関連するルーチン LIB$UNLOCK_IMAGE を使用することをお勧めします。ワーキング・セット内にイメージをロックする方法についての詳細は，『OpenVMS RTL Library (LIB$) Manual』のこれらのルーチンの説明を参照してください。

6

共用データの整合性の維持

この章では，共用データの整合性を維持するために必要な同期メカニズムについて説明します。たとえば，ある種の VAX 命令で保証されている不可分性などについて説明します。

6.1 概要

アプリケーションで複数の実行スレッドを使用しており，これらのスレッドが同じデータをアクセスする場合には， I64 システムで共用データの整合性を保護するためには，アプリケーションに明示的な同期メカニズムを追加しなければなりません。正しく同期をとらなかった場合には， 1 つのアプリケーション・スレッドによって開始されたデータ・アクセスが，別のスレッドによって同時に開始されたアクセスに干渉する可能性があり，データは予測できない状態になります。

VAX システムでは，必要な同期のレベルは，以下のような実行スレッドの関係に応じて異なります。

- 1 つのプロセス内で実行される複数のスレッド。たとえば，非同期システム・トラップ (AST) スレッドによってメイン・スレッドが割り込まれる場合

  AST スレッドはアプリケーションによって開始されますが，オペレーティング・システムによって開始されることもあります。たとえば，オペレーティング・システムは， AST を使用して状態を入出力状態ブロックに書き込みます。また，オペレーティング・システムは， AST を使用して，指定したユーザ・バッファへのバッファード I/O の読み取り操作を終了します。
- 1 つのプロセッサ上で複数のプロセスが実行されており，各プロセス内の複数のスレッドが共通のグローバル・セクションにアクセスする場合
- 複数のプロセッサ上で複数のプロセスが**並列に**実行されており，各プロセス内のスレッドがグローバル・セクションにアクセスする場合

VAX システムでは，マルチプロセッサ・システムの並列処理機能を利用するアプリケーションは，必ずロック，セマフォ，インターロック命令などの明示的な同期メカニズムを準備することにより，共用データを保護しなければなりません。しかし，ユニプロセッサ・システムで複数のスレッドを使用するアプリケーションは，明示的に共用データを保護していない可能性があります。これらのアプリケーションは， VAX ユニプロセッサ・システムで実行されるアプリケーションのスレッド間の同期を保証する VAX アーキテクチャの機能によって提供される，暗黙の保護に依存している可能性があります (第 6.1.1 項を参照)。

6.1.1 不可分性を保証する VAX アーキテクチャの機能

VAX アーキテクチャの次の機能は，ユニプロセッサ・システムで実行される複数の実行スレッド間で同期を保証します (ただし VAX アーキテクチャでは，マルチプロセッサ・システムに対してはこのような不可分性は保証されません)。

- **命令の不可分性**—VAX アーキテクチャによって定義されている多くの命令は，単一プロセッサで実行される複数のアプリケーション・スレッドの観点から見ると，1 つの割り込み不可能なシーケンス (**不可分な**操作と呼ぶ) として読み取り/変更/書き込み操作を実行できます。 Intel Itanium アーキテクチャでは，このような命令がサポートされます。

  VAX システムとの互換性を維持するために， Intel Itanium アーキテクチャでは，読み取り/書き込み操作が不可分な方法で実行されることを保証する，いくつかの命令が定義されています。これらの命令についての説明と，高級言語で作成されたプログラムでこの機能を利用するために， I64 システムのコンパイラが提供している機能については，第 6.1.2 項を参照してください。

  しかし， VAX システムでも， VAX 命令の不可分性に暗黙に依存することは望ましくありません。 VAX システムのコンパイラは，最適化を実行するため，インクリメント操作 (x = x+ 1) のようなプログラム文に対して， VAX の不可分な命令が使用できる場合でも，それを使用するという保証はありません。

- **メモリ・アクセスの粒度**—VAX アーキテクチャは，バイト・サイズのデータとワード・サイズのデータを 1 つの割り込み不可能な操作で処理できる命令をサポートしています (VAX アーキテクチャは他のサイズのデータを処理する命令もサポートしています)。 Intel Itanium アーキテクチャでも，バイト・サイズのデータとワード・サイズのデータを操作する命令をサポートしています。

- **読み取り/書き込みの順序**—VAX ユニプロセッサ・システムおよびマルチプロセッサ・システムでは，順次書き込み操作と読み取り操作は，すべてのタイプの外部実行スレッドから見て，要求した順序と同じ順序で実行されます。 I64 ユニプロセッサ・システムでも，ユニプロセッサで実行される単一プロセス，または複数プロセス内で実行される複数の実行スレッドに対して，読み取り操作と書き込み操作の順序は同期がとられているように見えます。しかし， I64 マルチプロセッサ・システムで同時に実行されるスレッドからの書き込み操作では，明示的に同期をとることが必要です。

  VAX システムとの互換性を維持するために， Intel Itanium アーキテクチャでは，システム内のすべてのプロセッサから見て，読み取り/書き込み操作が指定した順に実行されるようにする命令をサポートしています。この命令についての説明と，高級言語でこの命令をどのように使用するかについての説明は，第 6.1.2 項を参照してください。この同期をとるために Intel Itanium アーキテクチャが提供する機能についての説明と，高級言語で作成されたプログラムでこの機能を利用するために， I64 システムのコンパイラが提供している機能については，第 6.3 節を参照してください。

6.1.2 Intel Itanium の互換性機能

Intel Itanium アーキテクチャには，VAX アーキテクチャの不可分性機能との互換性を維持するためのいつかのメカニズムがあります。

- Compare and Exchange **命令**—Intel Itanium 命令セットでは，Compare and Exchange 命令と呼ばれる 4 つの命令 (cmpxchg1，cmpxchg2，cmpxchg4，cmpxchg8) が定義されており，不可分な比較とメモリ入れ替え操作が可能になっています。
- Exchange **命令**—Intel Itanium 命令セットでは，Exchange 命令と呼ばれる 4 つの命令 (xchg1，xchg2，xchg4，xchg8) が定義されており，不可分なメモリの入れ替え操作が可能になっています。
- Fetchadd **命令**—Intel Itanium 命令セットでは，Fetch and Add Immediate と呼ばれる 2 つの命令 (fetchadd4，fetchadd8) が定義されており，メモリ位置の不可分なインクリメント操作およびデクリメント操作が可能になっています。
- **メモリ・フェンス**—I64 命令セットには，マルチプロセッサ・システムで複数のプロセッサで実行される複数のスレッドが要求した読み取り/書き込み操作が要求した順に実行されているかのように見えるようにするための命令が準備されています。この命令はメモリ・フェンス (MF) と呼ばれ，複数の実行スレッドから見て，前のすべてのロード/ストア命令がメモリ・アクセスを完了するまで，後続のロード/ストア命令がメモリをアクセスしないことを保証します。

  MF 命令以外にも，前述したすべての命令には，後続のすべてのデータ・メモリ・アクセスの前，または前のすべてのデータ・メモリ・アクセスの後にメモリの読み取り/書き込みが見えるようにすることを保証する形式があります。

6.2 アプリケーションにおける不可分性への依存の検出

アプリケーションで同期が保証されると仮定している部分を検出するための 1 つの方法として，複数の実行スレッド間で共用されるデータを識別し，各スレッドからのデータ・アクセスを確認する方法があります。共用データを検出する際には，意図的に共用されるデータだけでなく，暗黙のうちに共用されるデータも検出しなければなりません。暗黙のうちに共用されるデータとは，複数の実行スレッドによってアクセスされるデータに近接しているために共用されるデータです。たとえば，$QIO，$ENQ，$GETJPI などのシステム・サービスの結果としてオペレーティング・システムが生成した AST によって書き込まれるデータは，このような暗黙のうちに共用されるデータです。

I64 システムのコンパイラは，どのデータに対してもデフォルトでクォドワード命令を使用する場合があるため，共用データが格納されているクォドワード内のすべてのデータは暗黙のうちに共用されてしまう可能性があります。たとえば，コンパイラは，自然な境界にアラインされていないデータにアクセスするためにクォドワード命令を使用します。(アドレスがデータ・サイズで割り切れる場合には，データは自然

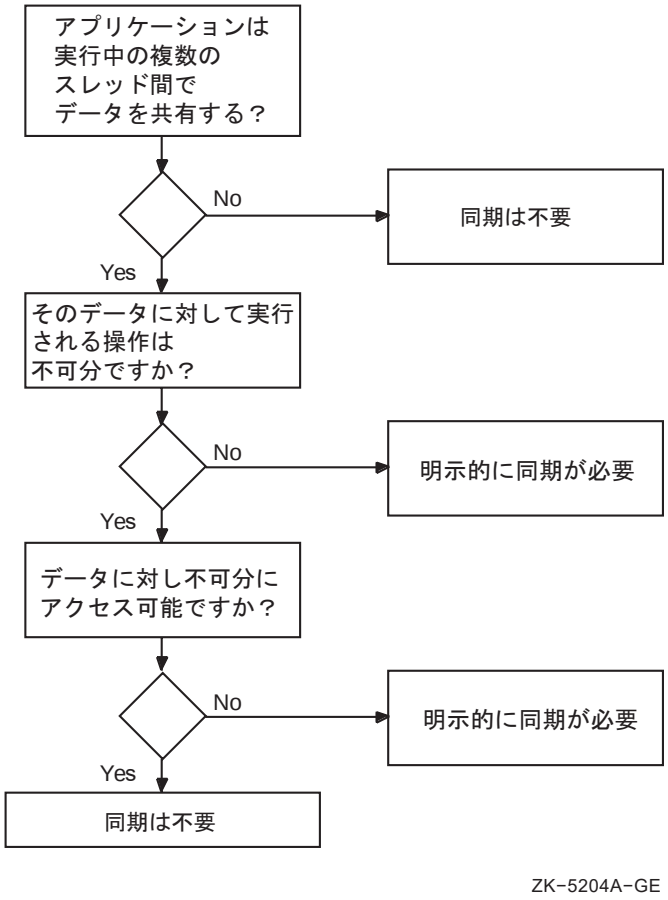
にアラインされています。詳細は，第 7 章を参照してください。コンパイラは，デフォルトで，明示的に宣言されたデータを自然な境界にアラインします。）

データ・アクセスを調べる場合には，処理途中の状態のデータを別のスレッドが参照する可能性がないかどうかを判断し，このような可能性がある場合には，それがアプリケーションにとって重要な問題であるかどうかを判断してください。場合によっては，共用データの値が正確であることがそれほど重要でない場合もあります。たとえば，アプリケーションが変数の相対値だけを必要とする場合には，正確な値は必要ありません。これらを調べるために，次の事項をチェックしてください。

- 共用データに対して実行される操作は，他の実行スレッドの観点から見たときに不可分ですか。
- 関係するデータ型に対する不可分な操作を実行できますか。

図 6–1 はこの判断を下す過程を示しています。

図 6–1 同期に関する判断



6.2.1 明示的に共用されるデータの保護

例 6–1 のプログラムは，VAX アプリケーションで不可分性が保証されると仮定した部分を簡単に示しています。このプログラムでは，flagという変数を使用しており，AST スレッドはこの変数を通じてメイン処理スレッドと通信します。この例では，カウンタ変数が前もって定義した値に到達するまで，メイン処理ループは処理を継続します。プログラムは AST 割り込みをキューに登録します。この割り込みにより，flag に最大値が設定され，処理ループを終了します。

例 6–1 AST スレッドを含むプログラムにおける不可分な処理への依存

```
#include <ssdef.h>
#include <descrip.h>

#define MAX_FLAG_VAL 1500

int    ast_rout();
long  time_val[2];
short int    flag;    /* accessed by main and AST threads */

main( )
{
     int      status = 0;
     static  $DESCRIPTOR(time_desc, "0 ::1");

     /*  changes ASCII time value to binary value  */

     status = SYS$BINTIM(&time_desc, &time_val);

     if ( status != SS$_NORMAL )
     {
        printf("bintim failure\n");
        exit( status );
     }

     /*  Set timer, queue ast */

     status = SYS$SETIMR( 0, &time_val, ast_rout, 0, 0 );

     if ( status != SS$_NORMAL )
     {
        printf("setimr failure\n");
        exit( status );
     }
```

(次ページに続く)

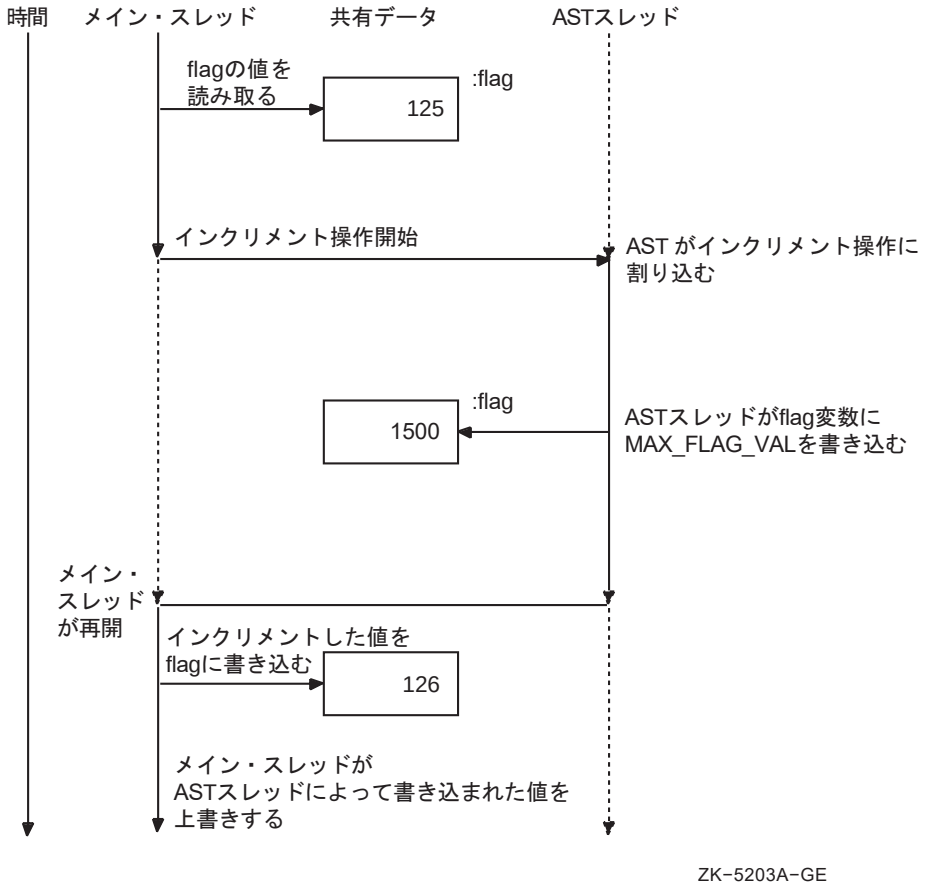
例 6–1 (続き) AST スレッドを含むプログラムにおける不可分な処理への依存

```
    flag = 0;   /* loop until flag = MAX_FLAG_VAL */
    while( flag < MAX_FLAG_VAL )
    {
          printf("main thread processing (flag = %d)\n",flag);
          flag++;
    }
    printf("Done\n");
}
ast_rout()     /*  sets flag to maximum value to stop processing */
{
     flag = MAX_FLAG_VAL;
}
```

例 6–1 では，flagという名前の変数がメイン実行スレッドと AST スレッドの間で明示的に共用されます。このプログラムでは，この変数の整合性を保護するために同期メカニズムを使用していません。つまり，インクリメント操作が不可分な方法で実行されることを暗黙のうちに仮定しています。

I64 システムでは，このプログラムは常に意図したとおりに動作するわけではありません。これは，図 6–2 に示すように，メイン実行スレッドのインクリメント操作の途中で，新しい値をメモリに書き戻す前に AST スレッドによって割り込まれる可能性があるからです (実際のアプリケーションでは，多くの AST スレッドがあるため，問題が発生する可能性がさらに高くなります)。この例では， AST スレッドはインクリメント操作が終了する前にこの操作に割り込みをかけ，変数の値として最大値を設定します。しかし，制御がメイン・スレッドに戻った後，インクリメント操作は完了し， AST スレッドの値が上書きされます。ループ・テストを実行すると，値は最大値でないため，処理ループは継続されます。

図 6–2 例 6–1 での不可分性の仮定



対処方法

このような不可分性への依存を修正するには，次の処理を実行してください。

- データにアクセスしている間，$SETAST システム・サービスを使用して AST の実行要求を禁止し，アクセスが終了した後で実行要求を可能にします。
- コンパイラのメカニズムを使用して，データを明示的に保護します。たとえば，HP C for OpenVMS I64 システムは，不可分性に関する組み込み関数をサポートしています。さらに，このデータへのアクセスの同期をとるために他のメカニズムを使用できます。たとえば，$ENQ システム・サービスを使用したり (マルチプロセッサ・システムで動作する複数のスレッドによってアクセスされるデータの場合)， LIB$BBCCI や LIB$BBSSI などのランタイム・ライブラリ・ルーチンやインターロック・キュー・ルーチンを使用することもできます。

  たとえば，例 6–1 では， C のインクリメント演算子 (flag++) によって実行されるインクリメント操作の代わりに， HP C for OpenVMS I64 システムがサポートする不可分性に関する組み込み関数 (__ADD_ATOMIC_LONG(&flag,1,0)) を使用してください。詳しい例については例 6–2 を参照してください。

共用変数を不可分性に関する組み込み関数によって保護するには，これらの変数はアラインされたロングワードまたはアラインされたクォドワードでなければなりません。

- バイト・サイズまたはワード・サイズのデータをロングワードまたはクォドワードに変更できない場合には，データをアクセスするときにコンパイラが使用する粒度を変更してください。I64 システムの多くのコンパイラでは，特定のデータをアクセスするときやモジュール全体を処理するときに使用する粒度を指定できます。ただし，バイト粒度やワード粒度を指定すると，アプリケーションの性能が低下するおそれがあります。

例 6–2 は，例 6–1 に示したプログラムに対してこれらの変更を行う方法を示しています。

例 6–2　例 6–1 の同期バージョン

```
#include <ssdef.h>
#include <descrip.h>
#include <builtins.h> 1

#define MAX_FLAG_VAL  1500
int    ast_rout();
long  time_val[2];
int 2        flag;    /* accessed by mainline and AST threads */

main( )
{
     int      status = 0;
     static  $DESCRIPTOR(time_desc, "0 ::1");

     /*  changes ASCII time value to binary value  */

     status = SYS$BINTIM(&time_desc, &time_val);

     if ( status != SS$_NORMAL )
     {
        printf("bintim failure\n");
        exit( status );
     }

     /*  Set timer, queue ast */

     status = SYS$SETIMR( 0, &time_val, ast_rout, 0, 0 );

     if ( status != SS$_NORMAL )
     {
        printf("setimr failure\n");
        exit( status );
     }
```

(次ページに続く)

例 6–2 (続き) 例 6–1 の同期バージョン

```
     flag = 0;
     while( flag < MAX_FLAG_VAL )   /* perform work until flag set to zero */
     {
           printf("mainline thread processing (flag = %d)\n",flag);
           __ADD_ATOMIC_LONG(&flag,1,0); 3
     }
     printf("Done\n");
}
ast_rout()     /*  sets flag to maximum value to stop processing */
{
      flag = MAX_FLAG_VAL;
}
```

以下の項目は例 6–2 に示した番号に対応しています。

1 HP C for OpenVMS I64 システムの不可分性に関する組み込み関数を使用するためには， builtins.h ヘッダ・ファイルをインクルードしなければなりません。

2 このバージョンでは，不可分なアクセスを可能にするために，変数flagはロングワードとして宣言されています (不可分性に関する組み込み関数を使用するために必要です)。

3 インクリメント操作は不可分性に関する組み込み関数を使用して実行されます。

6.2.2 無意識に共用されるデータの保護

例 6–1 では， 2 つのスレッドはどちらも同じ変数をアクセスします。これに対しては， I64 システムでは，暗黙のうちに共用される変数に対してアプリケーションで不可分性を持たせることが可能でした。次の例では， 2 つの変数はロングワードまたはクォドワードの範囲内で物理的に隣接しています。 VAX システムでは，各変数は個別に操作できます。 I64 システムでは，ロングワード・データとクォドワード・データのみ，不可分な読み取り操作と書き込み操作がサポートされるため，操作の対象となるバイトを変更する前に，ロングワード全体をフェッチしなければなりません (データ・アクセス粒度についての変更の詳細は，第 7 章を参照してください)。

この問題を示すために，例 6–1 のプログラムを変更したバージョンについて考えます。このバージョンでは，メイン・スレッドと AST スレッドはそれぞれ構造体で宣言された別々のカウンタ変数をインクリメントします。カウンタ変数は次の文によって宣言されます。

```
struct {
    short int     flag;
    short int ast_flag;
    };
```

メイン・スレッドと AST スレッドがどちらも，処理の対象となるワードを同時に変更しようとした場合には，2 つの操作が実行されるタイミングに応じて，結果は予想できなくなります。

対処方法

同期に関するこの問題を解決するには，次のことを行ってください。

- 共用変数のサイズをロングワードまたはクォドワードに変更します。しかし，I64 システムのコンパイラは状況によってはクォドワード命令を使用するため，データの整合性を確実に実現するためにはクォドワードを使用しなければなりません。たとえば，データが自然な境界にアラインされていない場合には，コンパイラはデータにアクセスするためにクォドワード命令を使用します。

  構造体の各要素が自然なクォドワード境界に強制的にアラインされるように，データの間にバイトを挿入することもできます。 OpenVMS I64 コンパイラは，デフォルトでデータを自然な境界にアラインします。

  たとえば，他の実行スレッドからの干渉を受けずに，構造体の各フラグ変数を確実に変更できるようにするには，64 ビットの変数となるように変数の宣言を変更します。 C の場合は`double`データ型を使用できます。次のコードを参照してください。

```
struct {
    double      flag;
    double  ast_flag;
    };
```

- 不可分性に関する組み込み関数や volatile 属性などのコンパイラ・メカニズムを使用して，データを明示的に保護してください。さらに，マルチプロセッサ・システムで実行される複数の実行スレッドによるデータ・アクセスは，$ENQ システム・サービスや， LIB$BBCCI や LIB$BBSSI などのランタイム・ライブラリ・ルーチンを使用するか，インターロック・キュー操作を使用することにより同期させることができます。

6.3 読み取り/書き込み操作の同期

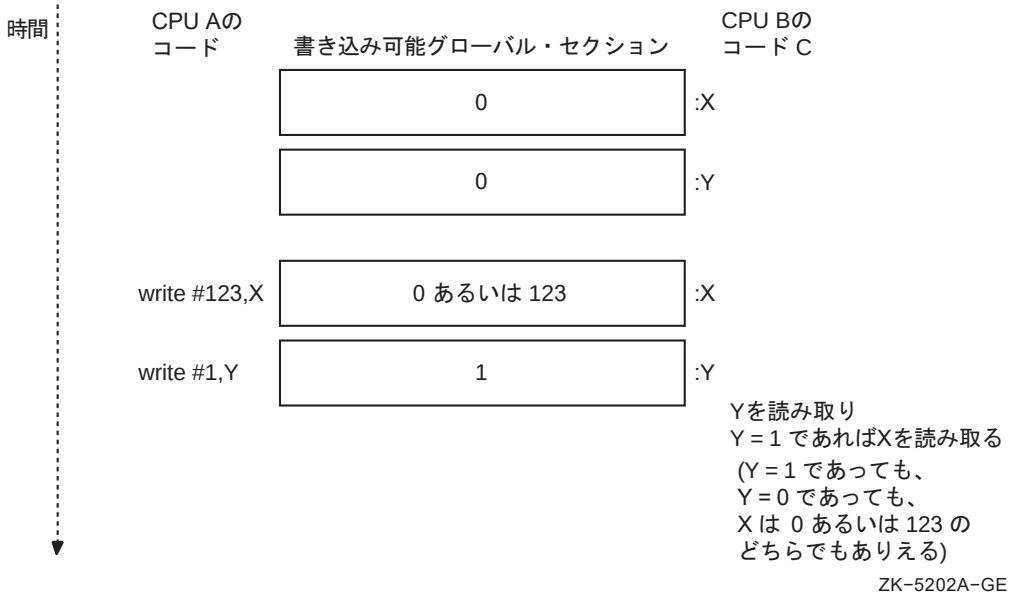
VAX マルチプロセシング・システムは従来，マルチプロセシング・システム内の 1 つのプロセッサが複数のデータを書き込むときに，これらのデータが書き込まれた順序と同じ順序で，他のすべてのプロセッサから確認できるように設計されていました。たとえば， CPU A がデータ・バッファを書き込み (図 6–3 の X)，その後でフラグを書き込む場合 (図 6–3 の Y)， CPU B はフラグの値を確認することにより，データ・バッファが変更されたことを判断できます。

I64 システムでは，メモリ・サブシステム全体の性能を向上させるために，メモリとの間の読み取り操作および書き込み操作の順序が変更される可能性があります。単一プロセッサで実行されるプロセスの場合には，そのプロセッサからの書き込み操作は要求された順に読み取り可能になることを仮定できます。しかし，マルチプロセッ

サ・アプリケーションの場合には，メモリに対する書き込み操作の結果がシステム全体から確認できるようになる順序に依存することはできません。つまり， CPU A によって実行される書き込み操作は，書き込まれた順序とは異なる順序で CPU B から見える場合もあります。

図 6–3 はこの問題を示しています。 CPU A は X に対する書き込み操作を要求し，その後， Y に対する書き込み操作を要求します。 CPU B は Y からの読み取り操作を要求し， Y の新しい値を確認した後に， X の読み取り操作を開始します。 X の新しい値がまだメモリに書き込まれていない場合には， CPU B は古い値を読み取ります。この結果， CPU A と CPU B で実行されるプロシージャが依存するトークン受け渡しプロトコルは正しく機能しなくなります。 CPU A はデータを書き込み，フラグ・ビットをセットできますが， CPU B は，データが実際に書き込まれる前にフラグ・ビットがセットされていることを検出する可能性があり，その結果，誤って古いメモリの内容を使用してしまいます。

図 6–3 I64 システムでの読み取り/書き込み操作の順序



対処方法

並列に実行され，読み取り/書き込みの順序に依存するプログラムは， I64 システムで正しく実行するために何らかの設計変更が必要です。アプリケーションに応じて，以下の方法を使用してください。

- 終了の順序が重要な，すべての読み取り命令と書き込み命令の前後では， I64 メモリ・フェンス (MF) 命令を使用してください。たとえば， C for OpenVMS I64 システムコンパイラは，__MB( )組み込み関数でメモリ・フェンス命令をサポートします。

- コンパイラで提供される組み込みのメモリ・インターロック操作を使用するか，LIB$ランタイム・ライブラリで提供されるVAXインターロック命令ルーチンを使用するように，アプリケーションの設計を変更してください。
- ロックによってデータを保護するために，$ENQシステム・サービスと$DEQシステム・サービスを使用するように，アプリケーションの設計を変更してください。

6.4 トランスレートされたイメージの不可分性の保証

VESTコマンドの/PRESERVE修飾子は，VAXシステムで提供されるのと同じ不可分性を保証して，トランスレートされたVAXイメージをAlphaシステムで実行できるようにするためのキーワードを受け付けます。/PRESERVE修飾子のキーワードは複数のタイプの不可分性保護機能を提供します。ただし，これらの/PRESERVE修飾子のキーワードを指定すると，アプリケーションの性能が低下するおそれがあります(/PRESERVE修飾子の指定についての詳細は，『DECmigrate for OpenVMS AXP Systems Translating Images』を参照してください)。

VAXシステムでVAX命令が不可分な方法で実行できる操作を，トランスレートされたイメージでも不可分な方法で実行できるようにするには，/PRESERVE修飾子に対してINSTRUCTION_ATOMICITYキーワードを指定します。

ロングワードまたはクォドワードに格納された隣接バイトを同時に更新し，これらの各バイトが相互に干渉しないようにするには，/PRESERVE修飾子に対してMEMORY_ATOMICITYキーワードを指定します。

読み取り/書き込み操作が実行される順序が要求した順序と同じ順序で実行されるように見えるようにするには，/PRESERVE修飾子に対してREAD_WRITE_ORDERINGキーワードを指定します。

7

アプリケーション・データ宣言の移植性の確認

この章では，アプリケーションで使用しているデータがVAXアーキテクチャに依存しているかどうかを確認する方法について説明します。また，データ型の選択がI64システムでアプリケーションのサイズと性能にどのような影響を与えるかについても説明します。

7.1 概要

Cのintや， FortranのINTEGER*4など，高級プログラミング言語でサポートされるデータ型は，アプリケーションに対して高度なデータの移植性を保証します。なぜなら，これらのデータ型を用いていれば，マシン内部のデータ型を意識しなくてもよいからです。各言語が持つデータ型は，ターゲット・プラットフォームでサポートされるデータ型にマッピングされます。この理由から， VAXシステムのアプリケーションは，データ宣言を変更せずにI64システムで正しく再コンパイルし，実行することが可能です。

しかし，アプリケーションでデータ型に関して次のような仮定を設定している場合には，ソース・コードを変更しなければなりません。

- **データ型のマッピングに関する仮定**— アプリケーションによっては，高級言語によってマッピングされるVAXデータ型に依存している可能性があります。 Intel Itaniumアーキテクチャは大部分のVAXデータ型をサポートしていますが，サポートしていないデータ型もあります。アプリケーションでは， I64システムでサポートされないデータ型のサイズやビット形式に関して仮定を設定している可能性があります。第7.2節では，この問題について詳しく説明します。
- **データ型の選択に関する仮定**— データ型の選択が与える影響は， I64システムでは異なる可能性があります。たとえばVAXシステムで，メモリを節約してデータを表現するために，最小のデータ型を選択したとします。しかし， I64システムでは，逆にメモリ使用量が増加する可能性があります。第7.3節では，この問題について詳しく説明します。

7.2 VAXデータ型への依存の確認

データの互換性を維持するために， Intel Itaniumアーキテクチャは VAXアーキテクチャと同じデータ型の多くをサポートするように設計されています。表 7–1は， 2つのアーキテクチャがどちらもサポートするネイティブ・データ型を示しています(デ

ータ型の形式についての詳細は，『OpenVMS Programming Concepts Manual』の付録 B を参照してください)。

表 7–1　VAX と I64 のネイティブ・データ型の比較

VAX のデータ型	I64 のデータ型
バイト	バイト
ワード	ワード
ロングワード	ロングワード
クォドワード	クォドワード
オクタワード	–
F 浮動小数点	F 浮動小数点[1]
D 浮動小数点 (56 ビットの精度)	D 浮動小数点 (56 ビットの精度)[1]
G 浮動小数点	G 浮動小数点[1]
H 浮動小数点	–
–	S 浮動小数点 (IEEE)
–	T 浮動小数点 (IEEE)
–	X 浮動小数点 (IEEE)
可変長ビット・フィールド	–
絶対キュー	絶対ロングワード・キュー
–	絶対クォドワード・キュー
自己相対キュー	自己相対ロングワード・キュー
–	自己相対クォドワード・キュー
文字列	–
トレーリング数字列	–
リーディング・セパレート数字列	–
パック 10 進数字列	–

[1]ハードウェアでサポートされないデータ型。サポートはコンパイラで提供される。

対処方法

アプリケーションが VAX データ型の形式やサイズに依存していないかぎり，データ型のマッピングは自動的に変更されるため，アプリケーションを変更する必要はありません。可能な場合，I64 システムのコンパイラは，そのデータ型を VAX システムと同じネイティブ・データ型にマッピングします。VAX データ型が Intel Itanium アーキテクチャでサポートされない場合には，コンパイラはそれらのデータ型を，最も近いネイティブ I64 データ型にマッピングします (I64 システムのコンパイラが，サポートするデータ型をネイティブ I64 データ型にマッピングする方法についての詳細は，第 9 章とコンパイラのマニュアルを参照してください)。

次のリストは，データ型宣言に有効なガイドラインを示しています。

- VAX **浮動小数点データ型**—Integrity サーバでの VAX 浮動小数点データ型のサポートについては，ホワイト・ペーパー『Intel® Itanium®アーキテクチャにおける OpenVMS 浮動小数点について』を参照してください。このホワイト・ペーパーがある Web 上の場所については，本書の「まえがき」を参照してください。

- **ポインタ・データ**— アドレス (ポインタ)・データ型が整数データ型と同じサイズであると仮定している部分がないか確認してください。 I64 システムでは， 64 ビットのアドレスを使用することが可能です。 64 ビットのポインタを使用するようにアプリケーションを変換する方法については，『OpenVMS Programming Concepts Manual』を参照してください。

7.3 データ型の選択に関する仮定の確認

アプリケーションを I64 システムで再コンパイルし，正しく実行できる場合でも，Intel Itanium アーキテクチャの利点を活用したデータ型の選択を行っていない可能性があります。特に，データ型の選択は I64 システムでのアプリケーションの最終的なサイズと性能に影響を与える可能性があります。

7.3.1 データ型の選択がコード・サイズに与える影響

VAX システムでは，アプリケーションは通常，データにとって適切な最小サイズのデータ型を使用します。たとえば， -32,768 ～ 32,767 の範囲の値を表現する場合， C で作成したアプリケーションでは short 型の変数を宣言します。 VAX システムでは，このようにすることで必要な記憶空間を節約できます。また， VAX アーキテクチャはすべてのサイズのデータ型に対して動作する命令をサポートしているので，効率を損ないません。

I64 でもバイト命令およびワード命令がサポートされているため，バイト・フィールドやワード・フィールドをロングワード・フィールドやクォドワード・フィールドに変更する必要はありません。フィールドのサイズ拡大が必要な 1 つの理由は，大きな値を格納できるようにすることですが，すべてのバイト・フィールドやワード・フィールドを変更する必要はありません。

7.3.2 データ型の選択が性能に与える影響

データ型の選択が与えるもう 1 つの影響としてデータ・アラインメントがあります。アラインメントとは，メモリ内の位置についてのデータの属性です。 VAX システムのアプリケーションでは多くの場合，データ構造体定義や静的データ領域でバイト・サイズ，ワード・サイズ，およびそれ以上のサイズのデータ型が混在していますが，この結果，自然な境界にアラインされないデータが発生します (アドレスがサイズ (バイト数) の整数倍である場合には，データは自然にアラインされています)。

VAX システムでも I64 システムでも，アラインされていないデータにアクセスすると，アラインされているデータにアクセスする場合より多くのオーバヘッドが発生します。しかし， VAX システムでは，アラインされていないデータに対処するマイクロコードを使用しています。 I64 システムではこのようなハードウェアの支援はありません。したがって，アラインされていないデータを参照するとフォルトが発生し，オペレーティング・システムによって処理しなければならなくなります。したがっ

て，アラインされていないデータを参照したときの性能の低下は，I64 システムではきわめて大きくなります。

I64 システムのコンパイラが，アラインされていないデータに対する参照をコンパイル時に認識できる場合には，特殊な命令シーケンスを生成することにより，性能の低下を最低限に抑えようとします。この結果，実行時にアラインメント・フォルトが発生するのを防止できます。実行時にアラインされていないデータに対する参照が発生した場合には，アラインメント・フォルトとして処理しなければなりません。

対処方法

データ型の選択がコード・サイズと性能に影響を与える可能性があると知って，バイトとワードのアクセスで必要になる余分な命令を排除し，アラインメントを改善するために，バイト・サイズとワード・サイズのすべてのデータ宣言をロングワードに変更する必要があると思うかもしれません。しかし，データ宣言を全面的に変更する前に，以下の要素を考慮してください。

- **アクセスの頻度とデータの繰り返し回数**— バイト・サイズまたはワード・サイズのデータが何度も参照される場合には，それをロングワードに変更することにより，参照のたびに必要となる余分な命令を排除し，アプリケーション・サイズを大幅に削減できます。しかし，バイト・サイズまたはワード・サイズのある特定のデータが何度も参照されるわけではなく，何度も複製される場合 (たとえば，構造体のインスタンスが何度も生成される場合)，このようなデータのデータ型をロングワードに変更すると，大量のメモリが必要となり，ロングワードへの変更によるコストが，参照のたびに必要な追加命令のコストを上回る場合があります。ロングワードに変更した結果，3 バイトが余分に必要になり，そのデータを何千回も繰り返す場合は，必要な仮想メモリが大幅に増えます。したがって，データ宣言を変更する前に，データの使用方法を考慮し，性能を向上するためにどれだけの仮想メモリ (および物理メモリ) を使用できるかを判断しなければなりません。このようなサイズと性能のどちらを重視するかの判断は，設計段階で何度も考慮しなければなりません。
- **相互操作性の必要性**— データ・オブジェクトをトランスレートされた構成要素またはネイティブな VAX 構成要素と共用する場合には，他の構成要素がデータのバイナリ・レイアウトに依存しているため，レイアウトを改善するような変更は不可能です。この場合，コンパイラ (および VEST ユーティリティ) は，アラインされていないデータに対する参照の命令シーケンスを，生成するコードに含めることにより，性能に与える影響を最低限に抑えようとします。

データ型の選択を確認する場合には，これらの要因を考慮した上で，以下のガイドラインに従ってください。

- 頻繁に参照されるが，何度も複製されることのないデータに対しては，バイト・サイズとワード・サイズのフィールドをロングワードに変更してください。特に，性能が重要なフィールドに対しては，このような変更が必要です。
- 頻繁に参照されないが，何度も複製されるデータの場合には，変更は望ましくありまん。

- 頻繁に参照され，何度も複製されるデータの場合には，変更によるコード・サイズと性能への影響を注意深く調べた後で判断を下さなければなりません。
- I64 システムのコンパイラの機能を使用して，自然な境界にアラインされていないデータを検出してください。 I64 システムの多くのコンパイラ (HP Fortran など) は，/WARNING=ALIGNMENT 修飾子をサポートしています。この修飾子は，自然な境界にアラインされていないデータがないかチェックします。
- 実行時分析ツールである Program Coverage and Analyzer (PCA) と OpenVMS Debugger の機能を利用して，自然な境界にアラインされていないデータを実行時に検出してください。詳細は，『Guide to Performance and Coverage Analyzer for VMS Systems』と『OpenVMS デバッガ説明書』を参照してください。
- 相互操作性の問題がない場合には， I64 システムのコンパイラが提供している自然なアラインメントを利用してください。 I64 システムでは，コンパイラはデフォルトで可能なかぎりデータを自然な境界にアラインします。 VAX システムでは，コンパイラはバイト・アラインメントを使用します。

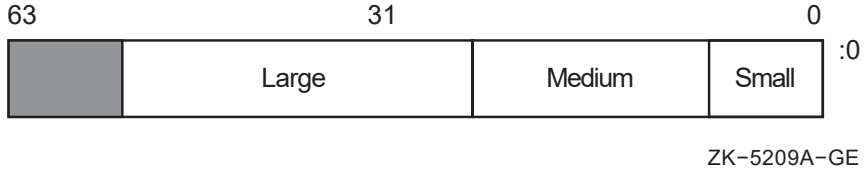
  I64 システムのコンパイラがサポートしている修飾子や言語プラグマを使用すると， VAX システムと同じバイト・アラインメントの使用を要求することができます。たとえば， OpenVMS I64 システム用の C コンパイラは/NOMEMBER_ALIGNMENT 修飾子と，それに対応したプラグマもサポートしています。これを使用することで，データ・アラインメントを制御することが可能となります。詳細は， C コンパイラのマニュアルを参照してください。

次の mystruct という構造体の例は，バイト・サイズ，ワード・サイズ，およびロングワード・サイズのデータで構成されます。

```
struct{
      char     small;
      short   medium;
      long     large;
      } mystruct ;
```

VAX Cを使用してコンパイルした場合には，この構造体は図 7–1 に示すようにメモリに配置されます。

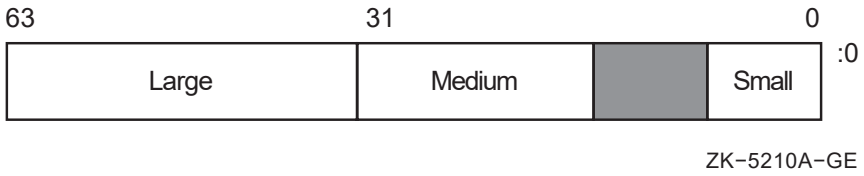
図 7–1 VAX Cによる mystruct のアラインメント



OpenVMS I64 システム用の C コンパイラを使用してコンパイルした場合には，図 7–2 に示すように，自然なアラインメントを実現するために構造体にパディングが

挿入されます。最初のフィールド (small) の後に 1 バイトのパディングを追加することにより，その後の構造体メンバはどちらもアラインされます。

図 7–2 C for OpenVMS I64 システムによる mystruct のアラインメント



構造体中のバイト・サイズとワード・サイズのフィールドは，アクセスのために複数の命令のシーケンスを必要とします。 small フィールドと medium フィールドが頻繁に参照され，構造体全体が何度も複製されることがない場合には，ロングワード・データ型を使用するように構造体を再定義することを検討してください。しかし，フィールドが頻繁に参照されない場合や，構造体が何度も複製される場合には，バイト参照やワード参照によって発生する性能の低下とメモリ・サイズの拡大のどちらが重要かを判断しなければなりません。

8

アプリケーション内の条件処理コードの確認

この章では，アプリケーション内の条件処理コードに対して， VAX アーキテクチャと Intel Itanium アーキテクチャの違いがどのような影響を与えるかについて説明します。

8.1 概要

ほとんどの場合，アプリケーションの条件処理コードは I64 システムでも正しく機能します。特に， Fortran の END， ERR， IOSTAT 指定子など，アプリケーションが高級言語で提供される条件処理機能を使用している場合には，問題はありません。これらの言語機能はアーキテクチャ固有の条件処理機能からアプリケーションを分離します。

しかし， I64 の条件処理機能と VAX の条件処理機能の間にはいくつかの違いがあり，場合によってはソース・コードを変更しなければなりません。次の場合には，ソース・コードの変更が必要です。

- メカニズム・アレイの形式への変更
- システムから返される条件コードの変更
- アプリケーション内での条件処理に関連する他の作業を実現する方法の変更。たとえば，例外通知を許可し，実行時に条件処理ルーチンを動的に指定するなど

この後の節では，これらの変更について詳しく説明し，ソース・コードの変更が必要かどうかを判断するのに役立つガイドラインも示します。

8.2 動的条件ハンドラの設定

OpenVMS I64 のランタイム・ライブラリ (RTL) には LIB$ESTABLISH ルーチンがありませんが， OpenVMS VAX の RTL にはこのルーチンがあります。 OpenVMS I64 の呼び出し規則により，条件ハンドラの設定はコンパイラによって行われます。

条件ハンドラを動的に設定しなければならないプログラムのために，一部の I64 言語では， LIB$ESTABLISH の呼び出しが特別な方法で取り扱われ，実際に RTL ルーチンを呼び出さずに適切なコードが生成されます。次の言語は，対応する VAX 言語と互換性のある方法で LIB$ESTABLISH をサポートします。

- HP C と HP C++

OpenVMS I64 システム用の HP C と HP C++ は， LIB$ESTABLISH を組み込み関数として取り扱いますが， OpenVMS VAX システムや OpenVMS Alpha システムで LIB$ESTABLISH を使用することは望ましくありません。 C および C++ のプログラマは， LIB$ESTABLISH の代わりに VAXC$ESTABLISH を呼び出すようにしてください (VAXC$ESTABLISH は， OpenVMS I64 システム用の HP C と HP C++ で提供される組み込み関数です)。

- HP Fortran

  HP Fortran では，内部関数 LIB$ESTABLISH および LIB$REVERT に対する宣言が可能であり，これらは HP Fortran RTL 固有のエントリ・ポイントに変換されます。

- HP Pascal

  HP Pascal では， ESTABLISH と REVERT という組み込みルーチンが提供され， LIB$ESTABLISH および LIB$REVERT の代わりに使用できます。 LIB$ESTABLISH を宣言して使用しようとすると，コンパイル時に警告が出力されます。

- MACRO–32

  MACRO–32 コンパイラは， LIB$ESTABLISH の呼び出しがソース・コードに指定されている場合，これを呼び出そうとします。

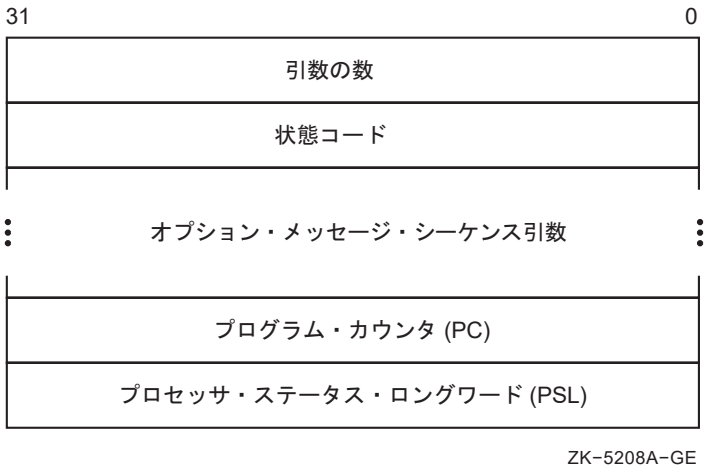
  MACRO–32 プログラムが 0(FP) にルーチン・アドレスを格納することにより，動的ハンドラを設定している場合には，これらのプログラムは， OpenVMS I64 システムでコンパイルした場合も正しく動作します。しかし， JSB (Jump to Subroutine) ルーチンの内部から条件ハンドラ・アドレスを設定することはできません。必ず CALL_ENTRY ルーチンの内部から設定する必要があります。

8.3 依存している条件処理ルーチンの確認

ユーザ作成条件処理ルーチンの呼び出しシーケンスは， I64 システムでも VAX システムのときと同じです。条件処理ルーチンは，例外条件を通知するときにシステムが返すデータにアクセスするために， 2 つの引数を宣言します。システムは，シグナル・アレイとメカニズム・アレイという 2 つの配列を使用して，どの例外条件がシグナルを起動したかを識別する情報を伝え，例外が発生したときのプロセッサの状態を報告します。

シグナル・アレイとメカニズム・アレイの形式はシステムで定義され，『OpenVMS Programming Concepts Manual』で説明されています。 I64 システムでは，シグナル・アレイに返されるデータとその形式は， VAX システムの場合と同じです。図 8–1 を参照してください。

図 8–1 VAX システムと I64 システム上の 32 ビット・シグナル・アレイ



次の表では，シグナル・アレイ内の各引数について説明しています。

引数	説明
引数の数	I64 システムでも VAX システムでも，この引数には正の整数が格納され，配列内でこの後に続くロングワードの数を示す。
状態コード	I64 システムでも VAX システムでも，この引数は 32 ビットのコードであり，ハードウェアまたはソフトウェアの例外条件を一意に識別する。条件コードの形式は I64 システムでも変更されておらず，『OpenVMS Programming Interfaces: Calling a System Routine』に説明されているとおりである。しかし，I64 システムでは，VAX システムで返されるすべての条件コードをサポートするわけではなく，さらに VAX システムでは返されない条件コードを定義している。I64 システムで返されない VAX 条件コードについては，第 8.4 節を参照。
オプション・メッセージ・シーケンス	これらの引数は返される例外に関する追加情報を提供し，例外によって異なる。これらの引数については，『OpenVMS Programming Concepts Manual』を参照。
プログラム・カウンタ (PC)	例外がトラップである場合には，例外が発生したときに次に実行される命令のアドレス。例外がフォルトの場合には，例外の原因となった命令のアドレス。I64 システムでは，この引数には PC の下位 32 ビットが格納される (I64 システムでは，PC の長さは 64 ビットである)。
プロセッサ・ステータス・ロングワード (PSL)	フォーマッティングした 32 ビットの引数であり，例外が発生したときのプロセッサの状態を示す。I64 システムでは，この引数には Alpha と同等のプロセッサ・ステータス (PS)・クォドワードの下位 32 ビットが格納される。フィールド IPL，CM，CSW，および IP が有効である。

I64 システムでは，メカニズム・アレイには，VAX の場合と同様のデータが返されます。しかし，その形式は異なります。I64 システムで返されるメカニズム・アレイには，浮動小数点スクラッチ・レジスタだけでなく，整数スクラッチ・レジスタの内容も保存されます。さらに，これらのレジスタの長さは 64 ビットであるため，メカニズム・アレイは，VAX システムのようにロングワード (32 ビット) ではなく，I64 システムではクォドワード (64 ビット) で構成されます。図 8–2 は，VAX システムでの

メカニズム・アレイの形式を示します。図 8–3 は， I64 システムでのメカニズム・アレイの形式を示します。

図 8–2 VAX システムでのメカニズム・アレイ

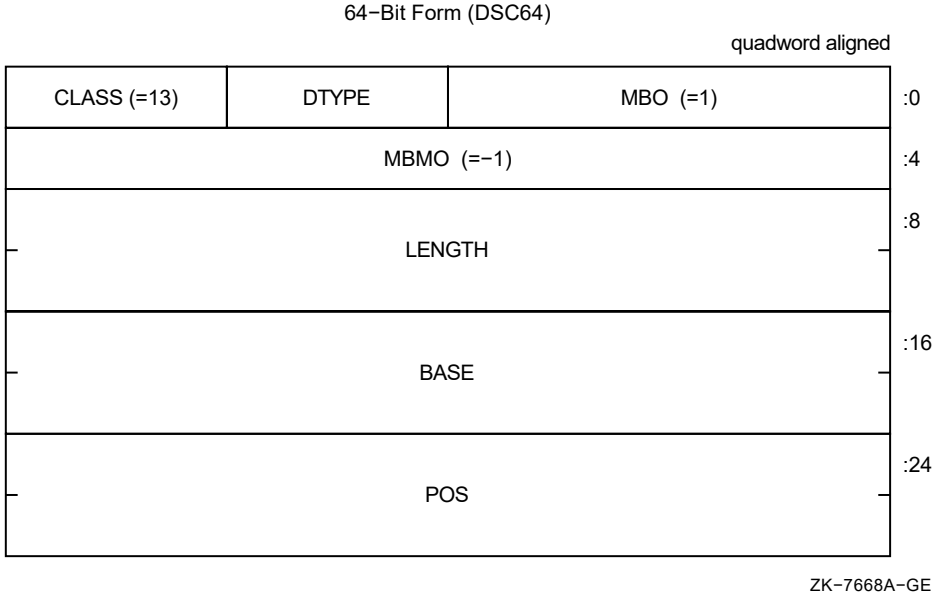
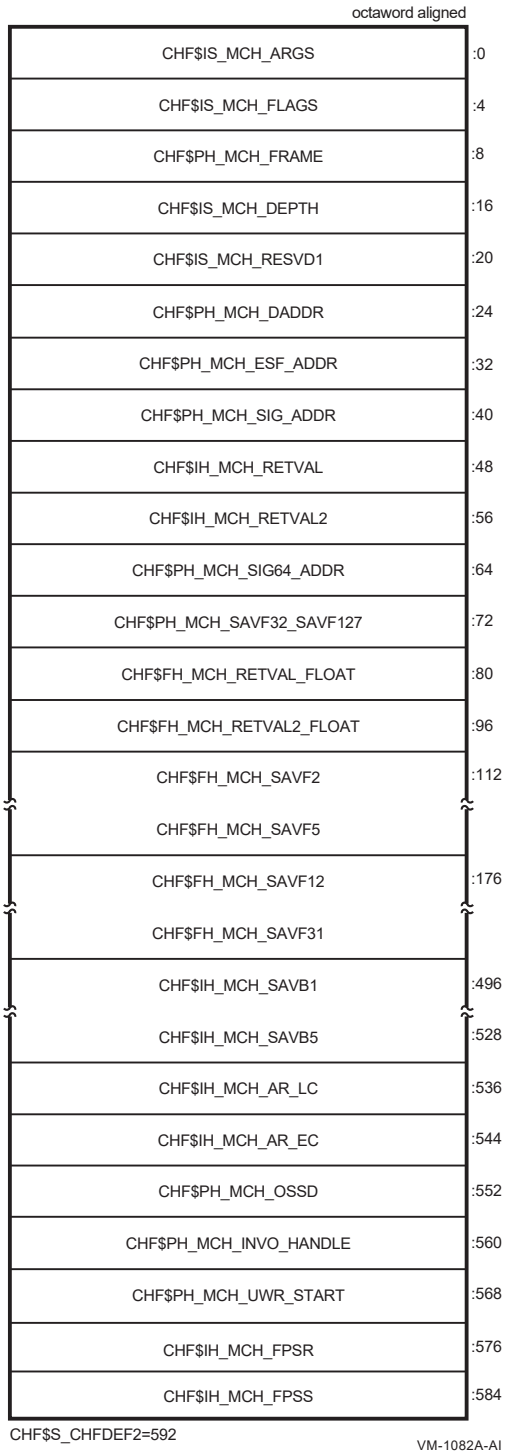


図 8–3 I64 システムでのメカニズム・アレイ



次の表では，メカニズム・アレイ内の各引数について説明しています。

引数	説明
引数の数	VAX システムでは，この引数には正の整数が格納され，配列内でその後に続くロングワードの数を示す。 I64 システムでは，この引数はメカニズム・アレイ内の**クォドワード**の数を示し，「引数の数」のクォドワードは含まない。 I64 システムでは， CHF$V_FPREGS_VALID がオフのときは引数の数は 71 で，このビットがオンのときは 263 である。
フラグ	I64 システムでは，この引数には追加情報を伝達するためのさまざまなフラグが格納される。ビット 0 は，プロセスがすでにレジスタ F2-F31 内で浮動小数点演算を実行し，配列内の浮動小数点レジスタが正しいことを示す (VAX システムのメカニズム・アレイにはこれに対応する引数はない)。ビット 1 は，プロセスがすでにレジスタ F32-F127 内で浮動小数点演算を実行し，拡張領域内の浮動小数点レジスタが正しいことを示す (VAX システムのメカニズム・アレイにはこれに対応する引数はない)。
フレーム・ポインタ (FP)	VAX システムでは， FP の内容が格納される。 I64 では，プロシージャのエントリ時点での SP の値である PSP (Previous Stack Pointer) が格納される。
深さ	VAX システムでも I64 システムでも，この引数には，例外を発生させたフレームを基準にして，条件処理ルーチンを設定したプロシージャのフレーム番号を表す整数が格納される。
リザーブ	予約されている。
ハンドラ・データ・アドレス	I64 システムでは，この引数には，ハンドラが存在する場合はハンドラ・データ・クォドワードのアドレスが格納される (VAX システムのメカニズム・アレイにはこれに対応する引数はない)。
例外スタック・フレーム・アドレス	I64 システムでは，この引数には例外スタック・フレームのアドレスが格納される (VAX システムのメカニズム・アレイにはこれに対応する引数はない)。
シグナル・アレイのアドレス	I64 システムでは，この引数には 32 ビットのシグナル・アレイのアドレスが格納される (VAX システムのメカニズム・アレイにはこれに対応する引数はない)。
関数の戻り値	I64 システムでは，これら 2 つのクォドワードには，例外発生時の R8 と R9 の内容が格納される (VAX システムのメカニズム・アレイにはこれに対応する引数はない)。
シグナル・アレイのアドレス	I64 システムでは，この引数には 64 ビットのシグナル・アレイのアドレスが格納される (VAX システムのメカニズム・アレイにはこれに対応する引数はない)。
浮動小数点例外アドレス	I64 システムでは，例外発生時点の浮動小数点レジスタ F32-F127 の内容が格納された配列のアドレスが格納される (VAX システムのメカニズム・アレイにはこれに対応する引数はない)。
浮動小数点の戻り値	I64 システムでは，これら 2 つのクォドワードは，例外発生時点の F8 と F9 の内容が格納される (VAX システムのメカニズム・アレイにはこれに対応する引数はない)。
レジスタ	VAX システムでも I64 システムでも，メカニズム・アレイにはスクラッチ・レジスタの内容が格納される。 I64 システムでは，この引数にはるかに大きなレジスタ・セットが格納され，浮動小数点レジスタ，分岐レジスタ，一部のアプリケーション・レジスタも格納される。

引数	説明
オペレーティング・システム固有のデータ領域	I64 システムでは，条件ハンドラのオペレーティング・システム固有のデータ領域のアドレスが格納される (VAX システムのメカニズム・アレイにはこれに対応する引数はない)。
起動ハンドル	I64 システムでは，条件ハンドラを設定したプロシージャの起動ハンドルが格納される (VAX システムのメカニズム・アレイにはこれに対応する引数はない)。
アンワインド領域	アンワインド領域のアドレス。

詳細は，『OpenVMS Calling Standard』を参照してください。

対処方法

32 ビット・シグナル・アレイは I64 システムと VAX システムとで同じであるため，条件処理ルーチンのソース・コードを変更する必要はありません。しかし，メカニズム・アレイは変更されているため，ソース・コードを変更しなければならない可能性があります。特に，以下のことを確認してください。

- 条件処理ルーチンのソース・コードを調べ，メカニズム・アレイ内の配列要素のサイズや配列要素の順序に関して何らかの仮定を設定していないかどうかを確認してください。
- アプリケーションの条件処理ルーチンで特定の数のスタック・フレームをアンワインドするために depth 引数を使用している場合には，ソース・コードを変更しなければならない可能性があります。アーキテクチャが変更されたため， I64 システムで返される depth 引数は VAX システムで返される depth 引数と異なる可能性があります (メカニズム・アレイの depth 引数は，例外が発生したフレームを基準にして，ハンドラを設定したプロシージャとの間のフレームの数を示します)。

  SYS$UNWIND システム・サービスに対して depth 引数のアドレスを指定することにより，例外処理ハンドラを設定したフレームまでアンワインドするアプリケーションや， SYS$UNWIND システム・サービスのデフォルトの depth 引数を使用することにより，例外処理ハンドラを設定したフレームの呼び出し側までアンワインドするアプリケーションは， I64 でも正しく動作します。 depth を負の値として指定した場合には，例外ベクタを示します (VAX システムの場合と同じ)。

例 8–1 は C で作成した条件処理ルーチンを示しています。

例 8–1 C で作成した条件処理ルーチン

```
#include  <ssdef.h>
#include  <chfdef.h>
   .
   .
   .
1  int cond_handler( sigs, mechs )
   struct  chf$signal_array  *sigs;
   struct  chf$mech_array   *mechs;
{
     int status;

2    status = LIB$MATCH_COND(sigs->chf$l_sig_name, /* returned code */
                             SS$_INTOVF);          /* test against  */

3    if(status != 0)
     {
         /*  ...Condition matched.  Perform processing.  */
         return SS$_CONTINUE;
     }
     else
     {
         /*  ...Condition does not match. Resignal exception. */
         return SS$_RESIGNAL;
     }
}
```

以下の項目は例 8–1 に示した番号に対応しています。

1 このルーチンでは，システムがシグナル・アレイとメカニズム・アレイに返すデータにアクセスするために，sigsとmechsという 2 つの引数を定義します。このルーチンは，前もって定義されている 2 つの構造体を使用して引数を宣言します。2 つのデータ構造とは chf$signal_array と chf$mech_array であり，システムによって CHFDEF.H ヘッダ・ファイルで定義されています。

2 この条件処理ルーチンは， LIB$MATCH_COND ランタイム・ライブラリ・ルーチンを使用することにより，返された条件コードと，整数オーバフローを示す条件コード (SSDEF.H に定義されているコード) を比較します。条件コードはシステム定義のシグナル・データ構造のフィールドとして参照されます (CHFDEF.H で定義されています)。

3 LIB$MATCH_COND ルーチンは，一致する条件コードを検出したときにゼロ以外の結果を返します。条件処理ルーチンはこの結果をもとに，異なるコード・パスを実行します。

8.4 例外条件の識別

アプリケーションの条件処理ルーチンは，シグナル・アレイに返された条件コードを確認することにより，どの例外が通知されているかを識別します。次のプログラムは例 8–1 から抜粋したものであり，ランタイム・ライブラリ・ルーチン LIB$MATCH_COND を使用することにより，条件処理ルーチン内でこの作業を実現する方法を示しています。

```
status = LIB$MATCH_COND( sigs->chf$l_sig_name,  /* returned code */
                                  SS$_INTOVF);  /* test against  */
```

I64 システムは， 32 ビットの条件コードの形式とシグナル・アレイ内での位置は，VAX システムの場合と同じです。しかし，条件処理ルーチンが VAX システムで受け取っていた条件コードの中には， I64 システムでは意味がないものがあります。アーキテクチャが異なるため， VAX システムで返されていた一部の例外条件は， I64 システムではサポートされません。

ソフトウェア例外の場合には， I64 システムは VAX システムの場合と同じ例外をサポートします。このことについては，オンライン・ヘルプ・メッセージ・ユーティリティまたは『OpenVMS system messages documentation』に示されています。しかし，ハードウェア例外はソフトウェア例外よりアーキテクチャに依存する部分が多く，特に算術演算例外はアーキテクチャに依存しています。 VAX システムでサポートされていたハードウェア例外の一部 (『OpenVMS Programming Concepts Manual』を参照) だけが I64 システムでもサポートされています。さらに， Intel Itanium アーキテクチャでは， VAX アーキテクチャでサポートされていない例外がいくつか追加定義されています。

表 8–1 は， I64 システムでサポートされない VAX ハードウェア例外と， VAX システムでサポートされない I64 ハードウェア例外を示しています。アプリケーションの例外処理ルーチンがこれらの VAX 固有の例外をテストしている場合には，対応する I64 の例外をテストするためのコードを追加する必要があります (I64 システムでの算術演算例外のテストについての詳細は，第 8.4.1 項を参照してください)。

注意

I64 システムで実行されるトランスレートされた VAX イメージも，これらの VAX 例外を返すことがあります。

表 8–1 アーキテクチャ固有のハードウェア例外

例外条件コード	コメント
I64 システム固有の例外	
SS$_FLTINV– 無効な浮動小数点オペランド値 (トラップ)	VAX システムには対応する例外はない
SS$_FLTINV_F– 無効な浮動小数点オペランド値 (フォルト)	VAX システムには対応する例外はない
SS$_FLTINE– 浮動小数点の結果が不正確 (トラップ)	VAX システムには対応する例外はない
SS$_FLTINE_F– 浮動小数点の結果が不正確 (フォルト)	VAX システムには対応する例外はない
SS$_FLTDENORMAL– 正規化されていない浮動小数点結果	VAX システムには対応する例外はない
SS$_ALIGN– データ・アラインメント・トラップ	VAX システムには対応する例外はない
VAX システム固有の例外	
SS$_ARTRES– 予備の算術演算トラップ	I64 システムには対応する例外はない
SS$_COMPAT– 互換性フォルト	I64 システムには対応する例外はない
SS$_DECOVF–10 進オーバフロー[1]	I64 ハードウェアでは生成されない
SS$_INTDIV–0 による整数除算[1]	I64 ハードウェアでは生成されない
SS$_INTOVF– 整数オーバフロー[1]	I64 ハードウェアでは生成されない
SS$_TBIT– トレース・ペンディング	I64 システムには対応する例外はない
SS$_OPCCUS– ユーザ用に確保されているオペコード	I64 システムには対応する例外はない
SS$_RADMOD– 予備のアドレッシング・モード	I64 システムには対応する例外はない
SS$_SUBRNG–INDEX 添字範囲チェック	I64 システムには対応する例外はない

[1]I64 システムではソフトウェアによって生成される可能性があります。

8.4.1 I64 システムでの算術演算例外のテスト

OpenVMS I64 では， OpenVMS VAX とほぼ同じ浮動小数点条件コードを使用しますが，新しい条件コードもいくつか使用します。また， VAX ではハードウェアで条件が通知されていた場合でも，いくつかの条件はライブラリ・ソフトウェアで通知されます。さらに，同じ状況でもアーキテクチャによって例外が異なる場合があります。

Intel Itanium アーキテクチャの算術演算例外は， VAX と同様に厳密です。つまり，例外 PC は，失敗した命令を正確に示します (これに対し， Alpha アーキテクチャでは，浮動小数点例外は非同期に検出され，厳密ではありません)。しかし， I64 の命令形式により，例外 PC は単なる失敗した命令のアドレスではありません。 I64 の命令は，それぞれ 3 つの命令がまとめられた 16 バイトのバンドルで構成されます。例外 PC は，命令バンドルのアドレスを指し，下位 2 ビットで命令スロット番号を表します。

命令アドレッシングと命令形式の違いにより，例外の原因となった命令を解釈する条件ハンドラコードを書き換える必要があります。

対処方法

算術演算例外に応答して処理を実行する条件処理ルーチンを I64 システムで実行するために変更しなければならないかどうかを判断する場合には，次のガイドラインに従ってください。

- アプリケーション内の条件処理ルーチンが，発生した算術演算例外の数だけを数える場合や，算術演算例外が発生したときに強制終了する場合には，最低限の修正だけで I64 システムで正しく動作するようになります。これらの条件処理ルーチンでは， I64 で追加された条件コードのテストを追加するだけで十分です。
- アプリケーションで例外の原因となった演算を再実行しようとする場合には，条件ハンドラ中の，失敗した命令を見つけて解釈する部分のコードを書き直す必要があります。

注意

I64 システムで実行されるトランスレートされた VAX イメージは，算術演算例外条件も含めて，VAX 例外条件を返します。VEST コマンドの利用についての詳細は，『DECmigrate for OpenVMS AXP Systems Translating Images』を参照してください。

8.4.2 データ・アラインメント・トラップのテスト

I64 システムでは，自然なアラインメントになっていないアドレスを使用して，レジスタとの間でロングワードまたはクォドワードをロード/ストアしようとする操作を実行すると，データ・アラインメント・トラップが発生します (データ・アラインメントについての詳細は，第 7 章を参照してください)。

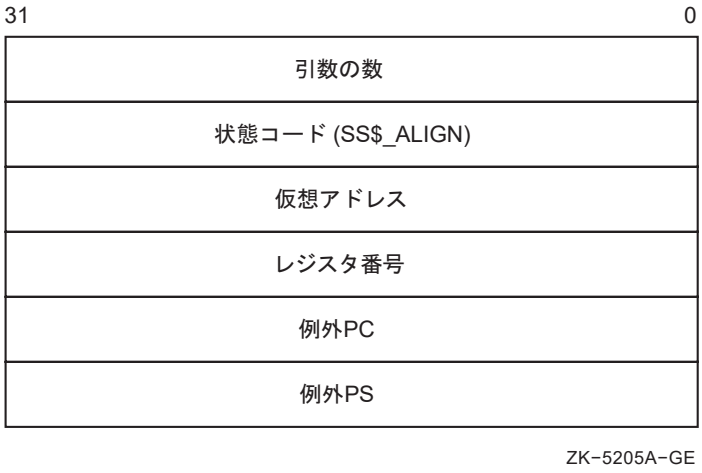
I64 システムのコンパイラは通常，次の操作を実行することにより，アラインメント・フォルトの発生を防止します。

- デフォルトで，静的データを自然な境界にアラインします (このデフォルトの動作は，コンパイラ修飾子を使用することにより変更できます)。
- コンパイル時に正しくアラインされていないことがわかっているデータに対して，特殊なインライン・コード・シーケンスを生成します。

しかし，動的に定義されるデータをコンパイラがアラインすることはできません。したがって，このような場合はアラインメント・フォルトが発生する可能性があります。

アラインメント例外は条件コード SS$_ALIGN によって示されます。図 8–4 は，SS$_ALIGN 例外によって返されるシグナル・アレイの要素を示しています。

図 8–4 SS$_ALIGN 例外のシグナル・アレイ



このシグナル・アレイには，SS$_ALIGN 例外固有の 2 つの引数が格納されます。それは**仮想アドレス**と**レジスタ番号**です。仮想アドレスには，アクセスしているアラインされていないデータのアドレスが格納されます。**レジスタ番号**は操作の対象となるレジスタを示します。

対処方法

- アプリケーションの開発中にアラインメント・フォルトを検出するには，この例外条件を使用します。このようにすれば，アプリケーションの性能に影響するデータ・アラインメントの問題をこの段階で修正することができます。この例外が報告されているということは，アプリケーションはデータ・アラインメントの問題によって性能に影響を受けているということになります。

8.5 条件処理に関連する他の作業の実行

いままでに述べてきた条件処理ルーチンの問題に加えて，条件処理を含むアプリケーションは，システムに対して条件処理ルーチンを設定するなどの処理を実行しなければなりません。ランタイム・ライブラリには，アプリケーションでこれらの処理を実行するためのルーチンが用意されています。たとえば，アプリケーションでランタイム・ライブラリ・ルーチン LIB$ESTABLISH を呼び出すことにより，例外が発生したときに実行する条件処理ルーチンを識別 (または設定) することができます。

VAX アーキテクチャと Intel Itanium アーキテクチャには相違点があり，両方のアーキテクチャの呼び出し規則にも違いがあるため，これらの多くの処理の実現方法は同じではありません。表 8–2 は，VAX システムで提供されるランタイム・ライブラリ条件処理サポート・ルーチンと，I64 システムではどのルーチンがサポートされるかを示しています。

表 8–2 ランタイム・ライブラリ条件処理サポート・ルーチン

ルーチン	I64 システムでのサポート
算術演算例外サポート・ルーチン	
LIB$DEC_OVER–10 進オーバフローの通知を許可または禁止する	サポートされない
LIB$FIXUP_FLT– 予備の浮動小数点オペランドを指定された値に変更する	サポートされない
LIB$FLT_UNDER– 浮動小数点アンダフローの通知を許可または禁止する	サポートされない
LIB$INT_OVER– 整数オーバフローの通知を許可または禁止する	サポートされない
一般的な条件処理サポート・ルーチン	
LIB$DECODE_FAULT– フォルトに対して命令コンテキストを解析する	サポートされない
LIB$ESTABLISH– 条件ハンドラを設定する	RTL ではサポートされないが，互換性を維持するためにコンパイラによってサポートされる
LIB$MATCH_COND– 条件値を照合する	サポートされる
LIB$REVERT– 条件ハンドラを削除する	RTL ではサポートされないが，互換性を維持するためにコンパイラによってサポートされる
LIB$SIG_TO_STOP– 通知された条件を継続できない条件値に変換する	サポートされる
LIB$SIG_TO_RET– シグナルをリターン・ステータスに変換する	サポートされる
LIB$SIM_TRAP– 浮動小数点トラップをシミュレートする	サポートされない
LIB$SIGNAL– 例外条件を通知する	サポートされる
LIB$STOP– シグナルを使用して実行を停止する	サポートされる

対処方法

次のリストは，ランタイム・ライブラリ・ルーチンを使用するアプリケーションにおけるガイドラインを示しています。

- アプリケーションで例外報告を可能にするランタイム・ライブラリ・ルーチンのいずれかを呼び出すことにより，例外の通知を許可している場合には，ソース・コードを変更しなければなりません。これらのルーチンは I64 システムではサポートされません。しかし，特定のタイプの算術演算例外は I64 システムで常に通知されるように設定されています。次のタイプの算術演算例外は常に通知されます。
  - 無効な浮動小数点演算
  - ゼロによる浮動小数点除算
  - 浮動小数点オーバフロー

デフォルトで通知されないように設定されている例外は，コンパイル時に通知されるように設定しなければなりません。

- アプリケーションで，ランタイム・ライブラリ・ルーチン LIB$ESTABLISH を呼び出すことにより条件処理ルーチンを指定している場合には，ソース・コードを変更する必要はありません。 I64 システムの大部分のコンパイラは，互換性を維持するために， LIB$ESTABLISH ルーチンへの呼び出しを受け付けます。コンパイラは「現在の」条件ハンドラを指す変数を，スタック上に作成します。 LIB$ESTABLISH はこの変数を設定し， LIB$REVERT はこの変数を消去します。これらの言語に対して静的に設定されたハンドラは，この変数の値を読み取り，どのルーチンを呼び出すかを判断します。特定の言語での詳細は，第 9 章を参照してください。

たとえば，例 8–2 に示した Fortran プログラムは， RTL ルーチン LIB$ESTABLISH を使用して，条件コード SS$_INTOVF を指定することで整数オーバフローをテストする条件処理ルーチンを指定しています。 VAX システムでは，整数オーバフローの検出を可能にするために，プログラムをコンパイルする際に/CHECK=OVERFLOW 修飾子を指定しなければなりません。

このプログラムを I64 システムで実行するには， VAX システムの場合と同様に，オーバフロー検出を可能にするためにコンパイル・コマンド行に /CHECK=OVERFLOW 修飾子を指定なければなりません。 HP Fortran は LIB$ESTABLISH ルーチンを組み込み関数として受け付けるため，このルーチンの呼び出しを変更する必要はありません。

例 8–2 条件処理プログラムの例

```
C       This program types a maximum value of integers
C       Compile with /CHECK=OVERFLOW and the /EXTEND_SOURCE qualifiers

        INTEGER*4 int4
        EXTERNAL HANDLER
        CALL LIB$ESTABLISH (HANDLER)  1

        int4=2147483645
        WRITE (6,*) ' Beginning DO LOOP, adding 1 to ', int4
        DO I=1,10
          int4=int4+1
          WRITE (6,*) ' INT*4 NUMBER IS  ', int4
        END DO
        WRITE (6,*) ' The end ...'
        END
```

（次ページに続く）

例 8–2 (続き) 条件処理プログラムの例

```
C       This is the condition-handling routine

        INTEGER*4 FUNCTION HANDLER (SIGARGS, MECHARGS)
        INTEGER*4 SIGARGS(*),MECHARGS(*)
        INCLUDE '($FORDEF)'
        INCLUDE '($SSDEF)'
        INTEGER INDEX
        INTEGER LIB$MATCH_COND

        INDEX = LIB$MATCH_COND (SIGARGS(2), SS$_INTOVF)
        IF (INDEX .EQ. 0 ) THEN
           HANDLER = SS$_RESIGNAL
        ELSE IF (INDEX .GT. 0) THEN
           WRITE (6,*) 'Arithmetic exception detected...'
           CALL LIB$STOP(SIGARGS(1))
        END IF
        END
```

次のリストの各項目は例 8–2 に示されている番号に対応しています。

1 この例では，条件処理ルーチンを指定するために LIB$ESTABLISH を呼び出します。

次の例は，例 8–2 に示したプログラムをコンパイル，リンク，および実行する方法を示しています。

```
$ FORTRAN/EXTEND_SOURCE/CHECK=OVERFLOW  EXCEPTIION
$ LINK  EXCEPTION
$ RUN EXCEPTION
 Beginning DO LOOP, adding 1 to 2147483645
 INT*4 NUMBER IS    2147483646
 INT*4 NUMBER IS    2147483647
Arithmetic exception detected...
%TRACE-F-TRACEBACK, symbolic stack dump follows
image     module    routine              line      rel PC           abs PC
exception  exception$MAIN  HANDLER       2662 0000000000000440 0000000000010440
DEC$FORRTL                                  0 00000000000DAC00 FFFFFFFF85440C00
                                            0 FFFFFFFF8039FE70 FFFFFFFF8039FE70
image     module    routine              line      rel PC           abs PC
exception  exception$MAIN  EXCEPTION$MAIN
                                           10 00000000000001C0 00000000000101C0
                                            0 FFFFFFFF80B356B0 FFFFFFFF80B356B0
DCL                                         0 000000000006AE90 000000007AE26E90
%TRACE-I-END, end of TRACE stack dump
```

9

OpenVMS I64 コンパイラ

この章では，ネイティブ OpenVMS I64 コンパイラ固有の機能について説明します。さらに， OpenVMS VAX コンパイラの機能のうち， OpenVMS I64 コンパイラではサポートされない機能と，動作が変更された機能についても示します。

以下にこの章で説明するコンパイラを示します。

- Ada (第 9.1 節)
- BASIC (第 9.2 節)
- C (第 9.3 節)
- COBOL (第 9.4 節)
- Fortran (第 9.5 節)
- Pascal (第 9.6 節)

コンパイラの相違点は， OpenVMS VAX で動作する以前のバージョンのコンパイラと最新版のコンパイラの違いと， VAX， Alpha，および I64 コンピュータで動作するバージョンの違いの 2 つに大きく分けることができます。 OpenVMS I64 コンパイラは， OpenVMS VAX および OpenVMS Alpha の対応するコンパイラと互換性を維持するように設計されています。この後の節に示すように，互換性を維持するためにいくつかの修飾子が提供されます。

各言語は言語標準規格に準拠し， OpenVMS VAX の大部分の言語拡張機能がサポートされています。コンパイラは OpenVMS VAX システムの場合と同じデフォルトのファイル・タイプを使用して出力ファイルを作成します。たとえば，オブジェクト・モジュールのファイル・タイプは.OBJ です。しかし， OpenVMS VAX システムのコンパイラでサポートされていた機能のうち，一部の機能は OpenVMS I64 システムでは利用できません。

各言語のコンパイラの相違点についての詳細は，その言語のマニュアル，特にユーザーズ・ガイドとリリース・ノートを参照してください。

9.1 I64 システムと VAX システム間の Ada の互換性

OpenVMS システムでは，以下の 2 つの Ada コンパイラが利用できます。

- HP Ada は弊社より提供され， OpenVMS VAX および OpenVMS Alpha で利用できます。 Ada 83 言語標準規格に準拠しており，以前はVAX Ada， DEC Ada，および Compaq Ada と呼ばれていました。 HP Ada は OpenVMS I64 では利用できません。
- GNAT Pro は， Ada Core Technologies 社より提供されます。これは，OpenVMS Alpha および OpenVMS I64 で利用でき， Ada 95 言語標準規格に準拠しています。 GNAT Pro は， VAX 以外の他の多数のプラットフォームでも利用できます。

表 9–1 に Ada コンパイラの比較を示します。

表 9–1 OpenVMS での Ada 言語のサポート

会社名	製品名	言語標準規格
HP	HP Ada	Ada 83
Ada Core	GNAT Pro	Ada 95

GNAT Pro は Ada 95 コンパイラで， HP Ada は Ada 83 コンパイラです。一般に，Ada 95 は Ada 83 との上位互換性が高いため， Ada 83 プログラムは変更なしまたは軽微な変更で Ada 95 で動作します。

VAX で Ada を使用しており，アプリケーションを I64 に移植する場合は，コンパイラを HP Ada から GNAT Pro に変更する必要があります。

Ada アプリケーションを VAX から I64 に移植する際には，以下の方法が利用できます。

- GNAT Pro を使用して直接 I64 に移植する
- HP Ada を使用して Alpha に移植し，その後 I64 に移植する
- GNAT Pro を使用して Alpha に移植し，その後 I64 に移植する

移植するプログラムは再コンパイルしなければなりません。 Ada イメージは VAXから I64 にトランスレートすることはできません。

詳細は，『Ada Technical Overview and Comparison』を参照してください。このドキュメントでは， HP Ada と GNAT Pro の互換性を詳細に比較しており，次の場所から入手できます。

h71000.www7.hp.com/commercial/ada/ada_index.html

このドキュメントは，Ada がインストールされているシステムの ADA$EXAMPLE:DEC_ADA_OVERVIEW_AND_COMPARISON.*ディレクトリにもあります。

GNAT Pro for I64 をインストールすると，『GNAT Pro User's Guide』がマニュアル領域にインストールされます。このマニュアルでは，HP Ada との互換性と，Ada 83 および Ada 95 の互換性について説明しています。

GNAT Pro についての詳細は，次に示す Ada Core 社の Web ページを参照してください。

http://www.gnat.com

問い合わせ先はsales@adacore.comです。

9.1.1 タスクに関する相違点

VAX 上のタスクの実装は，Alpha および OpenVMS I64 とは異なっています。Alpha と I64 はどちらも DECthreads を使用しますが，DECthreads は VAX では使用できません。

9.1.2 Ada を使用したイメージのトランスレート

Ada イメージは，別のマシン・ターゲットにトランスレートすることはできません。HP Ada イメージも GNAT Pro イメージも，Alpha や I64 にトランスレートすることはできません。

9.2 VAX BASICと HP BASIC の互換性

HP BASIC は，VAX システムで動作するVAX BASICを元にしており，高い互換性があります。しかし，以降の項で説明するように，いくつかの相違点もあります。ここで説明する情報は，両方の BASIC 製品と互換性のある BASIC アプリケーションを開発する場合や，VAX BASICアプリケーションを OpenVMS I64 および OpenVMS Alpha オペレーティング・システム上の HP BASIC に移行する場合に役立ちます。

9.2.1 HP BASIC では利用できないVAX BASICの機能

VAX BASICの以下の機能は，I64 システムおよび Alpha システムでは利用できません。

- プログラム開発のための，BASIC 固有の機能を提供するVAX BASIC環境，RUN コマンド，イミディエイト・モード

- 各行を入力した後に構文チェックを行うことを指定する/SYNTAX_CHECK 修飾子
- ANSI Minimal BASIC 標準規格を強制する/ANSI_STANDARD 修飾子
- PDP–11 BASIC/PLUS2 と互換性のないVAX BASICプログラムの機能を有効にする/FLAG=BP2COMPATIBILITY 修飾子
- I64 BASIC/Alpha BASIC でサポートされていないVAX BASICプログラムの機能を有効にする/FLAG=AXPCOMPATIBILITY 修飾子
- Program Design Facility (PDF) のサポートを有効にする/DESIGN 修飾子。/DESIGN を指定すると，コンパイラはプログラムのコンパイルを行わない。
- 『Programming with VAX BASIC Graphics』に記載されている， Graphics 文，変換機能，その他の機能
- 浮動小数点データ用の HFLOAT VAX 浮動小数点形式。また， DECLARE 文や REAL 関数などのさまざまな組み込み関数で， HFLOAT キーワードは指定できない。
- サイズが明示的に宣言されていない浮動小数点データ変数を HFLOAT と見なす/REAL_SIZE=HFLOAT 修飾子

9.2.2 VAX BASICでは利用できない HP BASIC の機能

HP BASIC の以下の機能は，VAX BASICでは利用できません。

- IEEE 浮動小数点データ型 SFLOAT (32 ビット)， TFLOAT (64 ビット)，および XFLOAT (128 ビット) と，整数データ型 QUAD (64 ビット) のサポート
- デフォルトの整数データ型のサイズをクォドワード (64 ビット) に設定する /INTEGER_SIZE=QUAD 修飾子
- デフォルトの浮動小数点データ型のサイズを IEEE 浮動小数点データ型のいずれかに設定する/REAL_SIZE={ SFLOAT | TFLOAT | XFLOAT }修飾子
- 個々のコンパイル単位をオブジェクト・ファイル中で個別のモジュールにするかどうかを制御する/SEPARATE_COMPILATION 修飾子
- コンパイラがVAX BASICの例外動作をエミュレートするための追加コードを生成するかどうかを制御する/SYNCHRONOUS_EXCEPTIONS 修飾子
- 自然にアラインされていない RECORD フィールド， COMMON および MAP 内の変数， RECORD 配列が見つかったら警告するようにコンパイラに指示する/WARNINGS=ALIGNMENT 修飾子
- ターゲット・マシン・アーキテクチャに応じて，より効率の良いコードを生成することをコンパイラに許可する/ARCHITECTURE 修飾子
- コンパイラが行う最適化のレベルを制御する/OPTIMIZE=(LEVEL=, TUNE=) 修飾子

9.2.3 VAX BASICと HP BASIC の動作の相違点

ここでは，VAX BASICと HP BASIC の動作の相違点について説明します。

9.2.3.1 浮動小数点データ型の演算

HP BASIC では， 3 種類の IEEE 浮動小数点データ型 SFLOAT， TFLOAT，および XFLOAT がサポートされています。これは，ベース・ハードウェアでもサポートされています。 Alpha アーキテクチャでは， VAX の DOUBLE (D 浮動小数点) データ型をサポートしていません。また， Intel Itanium アーキテクチャでは， VAX の浮動小数点データ型 (SINGLE (F 浮動小数点)， DOUBLE (D 浮動小数点)， GFLOAT, HFLOAT) のハードウェアによるサポートは行っていません。

表 9–2 浮動小数点データ型の対応

VAX BASIC	HP BASIC (Alpha)	HP BASIC (I64)	IEEE
SINGLE (F 浮動小数点)	F 浮動小数点	SFLOAT	SFLOAT
DOUBLE (D 浮動小数点)	GFLOAT	TFLOAT	TFLOAT
GFLOAT	GFLOAT	TFLOAT	TFLOAT
HFLOAT	XFLOAT	XFLOAT	XFLOAT

浮動小数点データは，宣言されたデータ型でメモリに格納されますが，データの演算を行う前にアーキテクチャでサポートされている型に変換されます。結果は，メモリに格納する前に必要なデータ型に変換されます。この変換処理によって精度が低下する可能性があります。

9.2.3.1.1 HP BASIC での (DOUBLE) D 浮動小数点データ型の使用 Alpha ハードウェアは D 浮動小数点データ型を完全にサポートしていないため， HP BASIC では BASIC DOUBLE 演算 (+， - など) を GFLOAT で行います。その結果，約 3 ビットの精度が失われます。

9.2.3.1.2 HP BASIC での VAX 浮動小数点データ型の使用 Intel Itanium ハードウェアは VAX 浮動小数点データ型をサポートしていないため， VAX 浮動小数点データは，演算の前にすべて対応する IEEE 浮動小数点データ型に変換されます。関係するデータ型によっては，精度が失われます。

9.2.3.1.3 HFLOAT データ型の暗黙的な使用 VAX BASICでは，ソース・コードで明示的な使用が指定されていない場合でも，中間的な計算を HFLOAT データ型で行う場合があります。一般に，値が大きな DECIMAL データと浮動小数点データの間で，データ型が混在する演算を実行する場合にこの処理が行われます。

Alpha プラットフォーム用の HP BASIC は，これらの操作を GFLOAT で実行し，I64 プラットフォーム用の HP BASIC は，これらの操作を TFLOAT で実行します。その結果，精度がいくらか失われる可能性があります。この相違点の原因となるソース・コードが見つかると，コンパイル時に次の警告メッセージが出力されます。

```
OPEPERGFL, operation performed in GFLOAT, loss of precision possible
```

9.2.3.1.4 CDD レコード中の HFLOAT データ

HP BASIC は，CDD レコード中の HFLOAT データを，CDD レコード中に見つかったその他のサポートされていないデータ型と同様に，16 バイトの文字列を含むグループにマッピングします。

9.2.3.2 浮動小数点データ型のデフォルトのサイズ

VAX BASICおよび HP BASIC for Alpha では，浮動小数点のデフォルトのサイズは SINGLE (VAX F 浮動小数点) であり，HP BASIC for I64 でのデフォルトは SFLOAT です。

9.2.3.3 パラメータの値渡し

HP BASIC およびVAX BASICでは，実パラメータを値で渡すことができますが，値渡しの仮パラメータを使用できるのは HP BASIC だけです。

9.2.3.4 配列パラメータ

HP BASIC とVAX BASICの配列パラメータの処理には，以下の相違点があります。

- サブプログラムまたは関数に配列全体が渡された場合，HP BASIC もVAX BASICもパラメータ・チェックを行います。渡された配列が，サブプログラムまたは関数が期待するものに一致しない場合は，コンパイラはエラー・メッセージ "Arguments don’t match"を出力します。VAX BASICでは，配列が参照されるたびにこのチェックが行われます。HP BASIC では，サブプログラムまたは関数の先頭で一度だけこのチェックが行われます。

  HP BASIC では，配列パラメータはより効率的に処理されます。HP BASIC とVAX BASICの配列パラメータの処理方法には，以下の違いがあります。

  - HP BASIC では，サブプログラムまたは関数がパラメータ・リスト中で配列を宣言している場合には，そのサブプログラムまたは関数を呼び出す際には，呼び出し元のプログラムは配列を渡さなければなりません。配列以外を渡すと，予期しないエラーが発生します。たとえば，配列の代わりにヌル・パラメータを渡すと，メモリ管理違反となり，プログラムは異常終了します。VAX BASICでは，サブプログラムまたは関数内で配列にアクセスしていない場合は，ヌルを渡してもかまいません。
  - HP BASIC では，サブプログラムは "Arguments don’t match"エラーをトラップすることはできません。通知されたエラーは，呼び出し元のプログラムでだけトラップできます。

- 配列全体を記述子によって渡す場合，VAX BASICは DSC$K_CLASS_A 記述子を作成しますが，HP BASIC は DSC$K_CLASS_NCA 記述子を作成します。

  呼び出し元のプログラムと呼び出されるサブプログラムは NCA 記述子を使用するため，ほとんどの BASIC アプリケーションではこれに気づきません。しかし，記述子の各フィールドを使用しているプログラムでは，HP BASIC で生成される記述子を扱うための変更が必要になります。

  DSC$K_CLASS_A 記述子と DSC$K_CLASS_NCA 記述子についての詳細は，『OpenVMS Calling Standard』を参照してください。

- VAX BASICでは， 10 進配列全体をサブプログラムまたは関数に渡す場合，スケール・チェックや精度チェックが実行されません。

  HP BASIC のサブプログラムと関数は，記述子によって受け取ったすべての 10 進配列をチェックし，その精度，位取り因数，境界情報が，呼び出し元プログラムのパラメータと一致していることを確認します。

  たとえば，次のプログラムでは，サブプログラム test_func が実行を開始すると，エラー“Arguments don’t match”が発生します。

```
DECLARE DECIMAL(5,2) a(10)
CALL test_func (a())
PRINT a(1)
END

SUB test_func (DECIMAL(10,4) b())
b(1) = 12.12
END SUB
```

- VAX BASICでは，呼び出し元からレコードの配列を受け取ると，最低限のチェックが実行されます。たとえば，次のプログラムでは，渡された配列のサイズがサブプログラムで宣言されたサイズに等しいことはチェックされません。
- HP BASIC では，配列要素のサイズが等しいことと，次元数が一致することがチェックされます。次のプログラムでは，サブプログラム test_func が実行を開始すると，エラー“Arguments don’t match”が発生します。

```
RECORD rec1
   LONG a
   LONG b
END RECORD
DECLARE rec1 a(10)
CALL test_func (a())
END

SUB test_func (rec2 a())
RECORD rec2
   LONG x
   LONG y
   LONG z
END RECORD
a(2)::x = 1
END SUB
```

- VAX BASICでは，記述子パラメータとして受け取った配列に対する境界チェックが必ず実行されます。

  HP BASIC では，/CHECK=NOBOUNDS 修飾子が指定されていると，記述子パラメータとして受け取った配列に対するチェックを実行しません。このようにすると，パラメータとして受け取った配列は他のすべての配列と整合性があります。

9.2.3.5 DEF*ルーチン

HP BASIC では， DEF ルーチンまたは WHEN ハンドラ内から DEF*ルーチンを呼び出すことはできません。呼び出そうとすると，次のエラー・メッセージが表示されます。

```
BASIC-E-DEFSNOTALL, DEF* reference not allowed in DEF or handler
```

HP BASIC では，式の中から呼び出された DEF*ルーチンに最も高い優先順位が与えられます。そのため， DEF*ルーチンの呼び出しが最初に評価されます。同じ式の中で使用されている変数の値を DEF*ルーチンが直接変更すると，式の結果が影響を受けます。式の中の DEF*呼び出しの順序がコンパイラによって変わる場合は，次の警告が表示されます。

```
BASIC-W-DEFEXPCOM, expression with DEF* too complex, moving <name>
invocation
```

このエラーを避けるには，式を単純化します。

9.2.3.6 /LINES 修飾子

HP BASIC では，/LINES 修飾子は ERL 関数だけに影響を与え，実行時エラー・メッセージ中に BASIC の行番号を表示するかどうかを決定します。 HP BASIC とVAX BASICには以下の違いがあります。

- /NOLINES 修飾子がデフォルトです。
- 対象のない RESUME 文を使用するために/LINES を使用する必要はありません。
- ほとんどの行に行番号があるプログラムで/LINES を使用すると，実行時の性能が低下します。

9.2.3.7 DCL コマンド行でのファイルの追加

VAX BASICでは， DCL コマンド行で追加演算子 (+) で指定したソース・ファイルには，ファイルに行番号が含まれている必要があります。行番号が含まれていないと，エラー・メッセージが表示されます。

HP BASIC では，ソース・ファイル中に行番号が含まれている必要はありません。追加演算子は OpenVMS の追加演算子として扱われます。ソース・ファイルは単一のソース・ファイルであるかのように追加されてコンパイルされます。

9.2.3.8 制御が到達しないコードに関するエラー

HP BASIC では，制御が到達しないコードを検索する際に広範囲にわたる分析が行われるため，VAX BASICよりも報告されるエラーの数が多くなります。

HP BASIC では，制御が到達しないコードに関するコンパイル時のエラー・メッセージ UNREACH は，情報メッセージです。VAX BASICでは，制御が到達しないコードに関するコンパイル時のエラー・メッセージ INACOFOL は警告です。

HP BASIC では，参照されない DEF 関数をチェックし，情報メッセージ“UNCALLED, routine xxxx can never be called.”が表示されます。

9.2.3.9 行番号

HP BASIC では，VAX BASICと異なり，重複した行番号や昇順に並んでいない行番号は許可されません。この制限は，単一のソース・ファイルと， DCL コマンド行で"+"で連結されたソース・ファイルに適用されます。重複した行番号や昇順に並んでいない行番号があると， E レベルのコンパイル・エラーが発生します。

HP BASIC では，この相違点に対処するための TPU コマンド・プロシージャの例が提供されています。このプロシージャを使用すると，ソース・ファイルを追加したり， 1 つ以上のソース・ファイルに対して BASIC 行番号を昇順にソートすることができます。

HP BASIC をインストールすると， TPU コマンド・プロシージャが SYS$COMMON:[SYSHLP.EXAMPLES.BASIC]BASIC$ENV.TPU に格納されます。使用方法はファイル中に記載されています。 TPU コマンド・プロシージャで判明している問題はありませんが，完全なテストは行われていないため，弊社によるサポートは提供されません。

9.2.3.10 エラー処理セマンティック

最高の性能を得るため， HP BASIC コンパイラは算術演算命令の順序を変更します。その結果，エラー処理セマンティックがVAX BASICと互換性がなくなる場合がまれにありますが，ほとんどのプログラムではこの変更による影響はありません。

VAX BASICと完全に同じ動作が必要なプログラムでは， HP BASIC の修飾子 /SYNCHRONOUS_EXCEPTIONS を使用してください。

9.2.3.11 オブジェクト・モジュールの生成

HP BASIC では，単一のソース・プログラム内のすべてのルーチン (SUB, FUNCTION，メイン・プログラム) は，デフォルトでオブジェクト・ファイル内の単一のモジュールとしてコンパイルされます。VAX BASICでは，各ルーチンは個別のモジュールとして生成されます。VAX BASICと同じ動作をさせるには， HP BASIC の/SEPARATE_COMPILATION 修飾子を使用してください。

9.2.3.12 RESUME と DEF

DEF の外にある，対象が指定されていない RESUME 文は， DEF 文内でプログラムの実行を再開することができないという文書化された制限があります。VAX BASICではこの制限は強制されませんが， HP BASIC では実行時にこの制限が強制されます。

9.2.3.13 例外

HP BASIC コンパイラは，式の結果が使用されないと判断すると，その式を評価するためのコードを生成しません。これにより，削除された式で例外が発生する場合は，VAX BASICとの互換性がなくなります。次のプログラム例をVAX BASICで実行すると，ゼロによる除算エラーが発生します。 HP BASIC は変数 A が使用されないことを認識するため， A に代入される式を評価するためのコードが生成されず，エラーは発生しません。

```
B = 5
A = B / 0
END
```

9.2.3.14 コンパイラ・メッセージの相違点

HP BASIC とVAX BASICのコンパイラ・メッセージの出力方法には若干の違いがあります。VAX BASICでは，ソース情報はメッセージ文の前に出力され，ソースとソース行番号がどちらも出力されます。 HP BASIC ではソース情報はメッセージ文の後に出力され，ソース行番号だけが出力されます。

HP BASIC コンパイラがソース行情報を報告する際のメッセージは，次のようになります。

```
%BASIC-E-xxxxxxxxx, xxxxxxxxxxxxx at line number YY in file xxxxxxxxxxxxx
```

HP BASIC でもVAX BASICでも，報告される行番号はファイル中の物理的なソース行の番号です。ソース・プログラム内に現れる BASIC 行番号ではありません。

9.2.3.15 DCL に返されるエラー状態

エラーが発生すると， HP BASIC コンパイラとVAX BASICコンパイラは，場合によって異なる状態を DCL に返します。たとえば， DCL コマンド行で指定されたファイルが見つからない場合， HP BASIC は BASIC-F-ABORT を返しますが，VAX BASICは BASIC-F-OPENIN を返します。

9.2.3.16 SYS$INPUT

DCL コマンド行の入力ファイル指定で SYS$INPUT を指定した場合，オブジェクト・ファイルとリスト・ファイルの名前は， HP BASIC とVAX BASICで異なります。 HP BASIC では，ファイルの名前は，ファイル・タイプ.OBJ および.LIS だけになります (区切り文字の前にファイル名がない)。VAX BASICでは， NONAME.OBJ および NONAME.LIS になります。

9.2.3.17 FSS$関数

VAX BASICコンパイラでは， FSS$関数を使用したプログラムをコンパイルすることができますが，実行時に FSS$関数が起動されると，次の実行時エラーが表示されます。

```
%BAS-F-NOTIMP, Not implemented
```

HP BASIC コンパイラでは， FSS$関数を使用していると，コンパイル時にすべての使用箇所で次のエラーが表示されます。

```
%BAS-E-BLTFUNNOT, built-in function not supported
```

9.2.3.18 BAS$K_FAC_NO 定数

BAS$K_FAC_NO 定数は I64 システムと Alpha システムでは定義されていません。EXTERNAL LONG CONSTANT BAS$K_FAC_NO は，すべて EXTERNAL LONG CONSTANT BAS$_FACILITY で置き換える必要があります。 OpenVMS VAX システムでは，定数 BAS$K_FAC_NO を使用して， SYS$LIBRARY:BASRTL.EXE と SYS$LIBRARY:BASRTL2.EXE の間で機能番号をやり取りします。この番号は I64 システムと Alpha システムでは必要ありません。

9.2.3.19 結果が異なる算術関数

HP BASIC とVAX BASICでは，いくつかの算術関数で結果が異なります。これは，元となる I64 と Alpha のシステム・ルーチンが，改良されたアルゴリズムを使用してこれらの演算を行うためです。

9.2.3.20 浮動小数点エラー

VAX システムで正常に実行できるプログラムの中には， Alpha システムと I64 システムではゼロによる除算などの浮動小数点エラーで異常終了するものがあります。エラーになったプログラムに汚れた浮動小数点ゼロがないか調べてください。「汚れた浮動小数点ゼロ」とは，指数部がゼロで仮数部がゼロでない数です。ほとんどの OpenVMS VAX システム命令は，不正な浮動小数点数をゼロとして扱いますが， I64 および Alpha の命令の中には，例外が発生するものがあります。

BASIC の算術演算式を使用して汚れたゼロを作ることはできませんが，ファイルから読み取ることで作ることができます。 GET や MOVE FROM などの BASIC の入出力文は，データが変数に対して有効かどうかをチェックせずにデータの各バイトを変数に格納します。

以下のいずれかの方法で問題に対処してください。

- 汚れたゼロがどのようにして作成されたかを調べて対処します。これが推奨される方法です。
- 汚れたゼロの値を受け取り，浮動小数点数を正常な値にするルーチンを作成します。

次の例は，単精度浮動小数点数を正常な値にするルーチンの例です (倍精度や G 浮動小数点についても同様のルーチンを作成できます)。

```
SUB clean_single (SINGLE a)
MAP (over) SINGLE b
MAP (over) WORD w1, w2
b = a
IF (w1 AND 32640%) = 0% THEN
   a = 0
END IF
END SUB
```

このルーチンは浮動小数点数を受け取り，指数部がゼロかチェックして，仮数部をクリアします。正しいビットがテストされるように，浮動小数点数を整数として再定義します。

浮動小数点形式と汚れたゼロについての詳細は，『Alpha Architecture Reference Manual』を参照してください。

9.2.3.21 不正な MAT 演算に関するエラーの検出

HP BASIC とVAX BASICの間には，不正な MAT 演算に関して，以下の 2 つの違いがあります。

- マトリックスの乗算を行う際に，演算対象のマトリックスのどちらかが代入先のマトリックスと同じ場合は， HP BASIC では正しく ILLOPE (Error 141 - "Illegal operation") が報告されます。VAX BASICでは，次のマトリックス乗算を行おうとしてもこの問題が正しく検出されず， ILLOPE メッセージが報告されません。ここで， B は仮想配列であり， A は仮想配列またはインメモリ配列です。

  ```
  MAT B = A * B
  ```

- VAX BASICでは，ある条件の下で， MAT 演算で使用する配列の下限がゼロでなければならないという，文書化された制限が強制されません。 HP BASIC では，下限がゼロでない配列に対して MAT 演算を実行しようとすると，コンパイル時に LOWNOTZER エラーとなるか，実行時に MATDIMERR エラーが報告されます。

9.2.3.22 デバッグの相違点

VAX BASICと HP BASIC には，デバッグの相違点があります。特に，例外ハンドラ， DEF 関数，外部サブプログラム， GOSUB ルーチンの近辺でデバッガの STEP コマンドを使用した場合の動作が異なります。これらの相違点についてここで説明しますが，HP BASIC for OpenVMS Systems User Manualも参照してください。

エラーのあるソース・コードでデバッガの STEP コマンドを使用すると， OpenVMS VAX と OpenVMS BASIC/Alpha の間で，デバッガの動作が異なります。これらの違いは， 2 つのシステムのハードウェアとソフトウェアのアーキテクチャの違いによるものです。

HP BASIC では，例外となる文に対して STEP コマンドを実行すると，デバッガに制御が戻らなくなります。デバッガは，例外が発生した後で BASIC ソース・コード中のどの文を実行すべきか判断することができません。そのため，例外が発生する文に対して STEP コマンドを使用する場合は，明示的なブレークを設定してください。

エラーを処理するプログラムをデバッグするために STEP コマンドを使用する際には，以下のヒントを参考にしてください。

- エラーを捕捉する文で STEP コマンドを使用すると，プログラムが明示的なブレークポイント，またはエラーが発生しなかった場合に実行される次の文に到達しないかぎり，デバッガに制御が戻りません。プログラムを他の場所で停止させたい場合は，明示的なブレークを設定します。

- エラーを捕捉する文で STEP コマンドを使用すると，プログラムがエラー・ハンドラ・コードに到達したときに，デバッガに制御が戻りません。プログラムの実行がエラー・ハンドラに渡った時点でプログラムをブレークさせたい場合は，エラー・ハンドラに明示的にブレークポイントを設定します。これは， ON ERROR ハンドラと WHEN ハンドラの両方に当てはまります。
- WHEN ハンドラの中では， WHEN ハンドラ内で実行を停止する文 (CONTINUE， RETRY， END WHEN， END HANDLER， EXIT HANDLER) で STEP コマンドを実行しても，明示的なブレークポイントが設定されている箇所にプログラム・フローが達するまで，実行は停止しません。
- ON ERROR ハンドラ内の RESUME ステートメントで STEP コマンドを実行すると，エラー・ハンドラでないコードの最初の行でプログラムの実行が停止します。
- 予期しないエラーが発生するのを防ぐために，デバッグ・セッションの始めで SET BREAK/EXCEPTION を実行します。すべてのエラー・ハンドラで明示的なブレークポイントを設定してある場合は，このブレークポイントは必要ありません。しかし，このコマンドを使用することで，すべての例外でブレークし，例外の後にプログラムの実行を停止させるための適切なブレークポイントを設定してあるかどうかを確認することができます。

9.2.3.23 リスト・ファイルの相違点

以下に HP BASIC とVAX BASICのリスト・ファイルの相違点を示します。

- /MACHINE/LIST—VAX BASICでは， BASIC/MACHINE を指定すると，マシン言語リストが含まれ，ソース・コード・リストが含まれないリスト・ファイルが出力されます。 HP BASIC では， BASIC/MACHINE を指定してもリストは出力されません。リスト・ファイルを出力するには，/LIST を指定しなければなりません。 HP BASIC では，/MACHINE/LIST を指定すると，マシン言語とソース・コードがリスト・ファイルに出力されます。

  VAX BASICでは， 1 つ以上のルーチンがあるプログラムのリスト・ファイルを作成する際，そのルーチンのソース・コードの後に各ルーチンの機械語コードが出力されます。 HP BASIC コンパイラで生成されるリスト・ファイルでは，/SEPARATE_COMPILATION 修飾子を指定しないかぎり，全ルーチンのソース・リストの後にすべてのルーチンの機械語コードのリストが出力されます。
- %PAGE—HP BASIC では，改ページ後に%PAGE 指示文が出力されます。VAX BASICでは，%PAGE 指示文は改ページ前に出力されます。
- %TITLE 文字列と%SBTTL 文字列 — これらの文字列は， HP BASIC では 31 文字，VAX BASICでは 45 文字に切り捨てられます。
- 用紙送り —VAX BASICでは用紙送りが%PAGE 指示文として扱われます。 HP BASIC では，用紙送りで特別な処理は実行されません。ソース・ファイル中に用紙送りがあると，リスト・ファイル中にもその用紙送りが出力されますが，それに対するリスト・ヘッダ情報は出力されません。

- /SHOW=MAP 修飾子 —/SHOW=MAP 修飾子を指定した場合， I64 BASIC /Alpha BASIC では以下の違いがあります。
  - HP BASIC では，割り当てマップ中の，値が該当しないか，リスト・フェーズで分からないオフセット・フィールドは空白のままになります。
  - 配列の動的なマップで，VAX BASICでは配列指示子のサイズが報告されますが， HP BASIC では配列のサイズが報告されます。
- メッセージの配置 — リスト・ファイル中のエラー・メッセージの配置は，VAX BASICと HP BASIC で異なります。たとえば， HP BASIC では，「制御が到達しないコード」や「呼び出されないルーチン」などのフロー分析が必要なエラーは，ソース・コード・リストと割り当てマップ・リストの後に出力されます。複数のルーチンが含まれているソース・ファイルのリストでは，/SEPARATE_COMPILATION 修飾子が指定されていないかぎり，これらのエラーは，コンパイル対象のすべてのルーチンのソース・リストと割り当てリストの後に出力されます。

9.2.4 共通言語環境の相違点

ここでは，共通言語環境内の HP BASIC，VAX BASIC，およびその他の言語の相違点について説明します。

9.2.4.1 COMMON 文と MAP 文による PSECT の作成

次の表に示すように， HP BASIC での PSECT 属性はVAX BASICと異なります。

HP BASIC	VAX BASIC
NOPIC	PIC
NOSHR	SHR
OCTAWORD アラインメント	LONG アラインメント

HP BASIC では， COMMON 文と MAP 文が作成する PSECT の長さは， 16 の倍数に切り上げられます。 COMMON や MAP のサイズは変わらず， PSECT のサイズが変わります。この変化が影響するのは，複数言語環境内の共有イメージを使用しているアプリケーションだけです。

HP BASIC もVAX BASICも，同じプラットフォーム上の他の言語と互換性がある PSECT を作成します (MACRO を除く)。 MACRO 以外の言語で作成されたモジュールと，コードを変更せずにリンクすることができます。これらの PSECT を参照する MACRO モジュールに対してリンクする場合は， MACRO コードに対して該当する変更を行う必要があります。

9.2.4.2 64 ビットの浮動小数点データ

他のほとんどの HP 言語では，デフォルトの 64 ビット浮動小数点データ型は，OpenVMS VAX システムの D 浮動小数点から， OpenVMS Alpha システムでは G 浮動小数点に， OpenVMS BASIC システムでは T 浮動小数点に変更されました。 BASIC と，この変更が行われている他の言語との間で， BASIC DOUBLE (OpenVMS D 浮動小数点) をやり取りする場合は，以下のいずれかを実行する必要があります。

- 他の言語のコンパイラのコマンド行で， 64 ビットの浮動小数点データ型を， D 浮動小数点に変更して Alpha BASIC の動作に合わせるか， T 浮動小数点に変更して I64 BASIC の動作に合わせます。
- BASIC プログラムで， 64 ビット浮動小数点データのデータ型を DOUBLE から GFLOAT または TFLOAT に変更し，他の言語に合わせます。

9.3 HP C とVAX Cの互換性

VAX CはもともとVAX/VMSシステムで提供されていた C コンパイラです。VAX Cはその後 DEC C で置き換えられています。 DEC C は Compaq C に名称が変更され，さらに最近その名称が HP C に変更されています。VAX Cで作成されたプログラムがある場合，まず OpenVMS VAX 上の HP C へのポーティングを行うことをお勧めします。詳細については，『Compaq C Migration Guide for OpenVMS VAX Systems』を参照してください。このマニュアルは以下の URL で公開されています。

~~h71000.www7.hp.com/commercial/c/docs/~~

ご使用のアプリケーションが HP C で作成されている場合，もしくはVAX Cから HP C へのポーティングが完了している場合， OpenVMS I64 用にポーティングする際にさらにいくつかの問題が発生することがあります。しかしこれらの問題は，どのプログラミング言語でも共通に発生する浮動小数点，ページ・サイズ，粒度，アライメント，アトミシティなどの問題です。この種の問題については本書の他の箇所で説明しています。

9.4 VAX COBOL と HP COBOL の互換性と移行

HP COBOL は， OpenVMS VAX システム上で動作するVAX COBOL に基づいており，高い互換性がありますが，いくつかの重要な違いがあります。

詳細は，『HP COBOL User Manual』を参照してください。

9.5 OpenVMS VAX システムと OpenVMS I64 システムの HP Fortran の互換性

ここでは， HP Fortran for OpenVMS I64 システムと HP Fortran 77 for OpenVMS VAX システム (HP Fortran 77，以前の名前はVAX FORTRAN) の互換性について，以下の分野に分けて説明します。

- 言語機能 (第 9.5.1 項)
- コマンド行修飾子 (第 9.5.2 項)
- トランスレートされた共有イメージとの相互操作性 (第 9.5.3 項)
- HP Fortran 77 データの移植(第 9.5.4 項)

9.5.1 言語機能

HP Fortran には， ANSI FORTRAN–77と ISO/ANSI Fortran 9x の標準機能が含まれており，さらにこれらの Fortran 標準機能に対して， HP Fortran 77 の拡張機能も含まれています。たとえば，以下の拡張機能が含まれています。

- RECORD 文と STRUCTURE 文
- CDEC$指示文と OPTIONS 文
- BYTE， INTEGER*1， INTEGER*2， INTEGER*4， LOGICAL*1， LOGICAL*2， LOGICAL*4
- REAL*4， REAL*8， REAL*16， COMPLEX*8， COMPLEX*16
- IMPLICIT NONE 文
- INCLUDE 文
- NAMELIST 入出力
- ドル記号($)とアンダスコア(_)を含む最大 31 文字の名前
- DO WHILE 文と END DO 文
- 行末コメントに対する感嘆符(!)の使用
- 組み込み関数%DESCR，%LOC，%REF，および%VAL
- VOLATILE 文
- DICTIONARY 文 (FORTRAN–77コンパイラのみ)
- POINTER 文データ型
- 再帰
- ディスクとメモリの間のフォーマットされていないデータの変換
- インデックス付きファイル

- PRINT, ACCEPT, TYPE, DELETE, UNLOCK などの入出力文
- CARRIAGECONTROL, CONVERT, ORGANIZATION, RECORDTYPE などの, OPEN 文と INQUIRE 文の指定子
- 適切な Fortran 言語リファレンス・マニュアルで示されている他の言語要素

拡張機能と言語機能についての詳細は, Fortran 言語のリファレンス・マニュアルを参照してください。このマニュアルには，FORTRAN–77標準の拡張機能が示されています。

ここでは, HP Fortran 77 の言語機能と，各言語で共用されるものの，異なる方法で解釈される HP Fortran の言語機能, HP Fortran 77 には適用されない HP Fortran の制限事項，データを移植する際の検討事項について説明します。

9.5.1.1 HP Fortran 固有の言語機能

以下の言語機能は HP Fortran では提供されますが, HP Fortran 77 バージョン 6.4 ではサポートされません。

- 文字定数の区切り文字としての二重引用符(" ")。これは/VMS 修飾子を指定することにより無効にできます。
- COMMON ブロックの項目およびレコードのフィールドに対する自然にアラインされた境界またはパックされた境界
- INTEGER*1, INTEGER*8，および LOGICAL*8 データ型
- IEEE の S 浮動小数点, T 浮動小数点および X 浮動小数点データ型のサポートと，ネイティブでなく，フォーマッティングもされていないデータ・ファイル形式のサポート。ビッグ・エンディアン数値形式もサポートされます。 Alpha システムのネイティブ浮動小数点データ型については，『OpenVMS Calling Standard』を参照してください。
- LIB$ESTABLISH と LIB$REVERT は, HP Fortran 77 の条件処理との互換性を維持するために，組み込み関数として提供されます。

  HP Fortran では, LIB$ESTABLISH の宣言は HP Fortran RTL 固有のエントリ・ポイントに変換されます。
- 倍精度の複素数組み込み関数に対する代替の“Z”綴り (たとえば，倍精度平方根組み込み関数は CDSQRT または ZSQRT と指定できます)
- 以下の組み込み関数

  IMAG
  AND
  OR
  XOR
  LSHIFT
  RSHIFT
- いくつかの実行時エラーは HP Fortran 固有のエラーです。

- 大文字と小文字を区別する名前
- HP Fortran では，入出力ユニット番号は 0 または正の整数として指定できます。HP Fortran 77 では，入出力ユニット番号の値は 0 ～ 99 の範囲です。

注意

HP Fortran 90 コンパイラを使用しているユーザに対する注意事項ですが，ANSI/ISO Fortran 90 標準に準拠したいくつかの機能は HP Fortran 77 では使用できません。

HP Fortran の言語機能についての説明は， Fortran 言語のリファレンス・マニュアルを参照してください。

9.5.1.2 HP Fortran 77 固有の言語機能

以下の言語機能は HP Fortran 77 では使用できますが， HP Fortran ではサポートされません。

- FORTRAN/PARALLEL=(AUTOMATIC) の自動分析機能
- CPAR$指示文を使用した FORTRAN/PARALLEL=(MANUAL) の手動分析機能 (CPAR$ DO_PARALLEL など)
- PDP–11との互換性を維持するための次の入出力およびエラー・サブルーチン

ASSIGN	ERRTST	RAD50
CLOSE	FDBSET	R50ASC
ERRSET	IRAD50	USEREX

既存のプログラムを移植する場合には， ASSIGN， CLOSE，および FBDSET の呼び出しは適切な OPEN 文に変更しなければなりません (HP Fortran for OpenVMS Alpha では DEFINE FILE 文はサポートされていますが， DEFINE FILE 文の変更についても同時に検討すべきです)。

ERRSET および ERRTST の代わりに OpenVMS の条件処理を使用できます。HP Fortran for OpenVMS Alpha は ERRSNS サブルーチンをサポートします。

- *n*R*xxx*という形式のRadix–50定数

既存のプログラムを移植する場合には， radix-50 定数と IRAD50， RAD50，および R50ASC ルーチンは， CHARACTER として宣言したデータを使用して，ASCII でエンコーディングしたデータに変更しなければなりません。

HP Fortran 77 の以下の機能は， HP Fortran では使用が制限されているか，使用できません。

- 数値のローカル変数は，使用した最適化のレベルに応じて， 0 に初期化されることがありますが，常に初期化されるわけではありません。どのような状況でも値が 0 に初期化されるようにするには，明示的な代入文または DATA 文を使用してください。

- 文字定数には，数値の仮引数ではなく，文字の仮引数を割り当てなければなりません (HP Fortran 77 for OpenVMS VAX Systems では，仮引数が数値の場合，'A'を参照によって渡します)。このような引数に対しては，/BY_REF_CALL 修飾子を使用することを検討してください。
- 保存された仮配列は機能しません。

```
SUBROUTINE F_INIT (A, N)
REAL A(N)
RETURN
ENTRY F_DO_IT (X, I)
A (I) = X  ! No: A no longer visible
RETURN
END
```

- ホレリス実引数には，文字仮引数ではなく，数値仮引数を割り当てなければなりません。

以下の言語機能は HP Fortran 77 では使用できますが， Itanium アーキテクチャと VAX アーキテクチャとの違いにより， HP Fortran ではサポートされません。

- いくつかの FORSYSDEF シンボル定義モジュールは， VAX アーキテクチャまたは Itanium アーキテクチャ固有のモジュールです。
- 正確な例外制御

  特定の例外の処理方法は， VAX システムと I64 システムとで異なります。
- REAL*16 データは VAX システムでは H 浮動小数点データ形式を使用し， I64 システムではX 浮動小数点を使用します。
- D 浮動小数点に対する VAX のサポート

  I64 の命令セットでは， D 浮動小数点 REAL*8 形式がサポートされないため， D 浮動小数点データは演算中にソフトウェアによって T 浮動小数点に変換され，その後， D 浮動小数点形式に戻されます。したがって， VAX システムと I64 システムの間には， D 浮動小数点の演算に違いがあります。

  I64 システムで最適な性能を実現するためには， IEEE T 浮動小数点形式での REAL*8 データの使用を考慮する必要があり，多くの場合は形式を指定するために/FLOAT 修飾子を使用します。 D 浮動小数点データを T 浮動小数点形式に変換するための HP Fortran for OpenVMS I64 アプリケーション・プログラムを作成するには， Fortran 言語のリファレンス・マニュアルで説明されているファイル変換方式を使用します。
- ベクタ化機能

  ベクタ化は，/VECTOR 修飾子とそれに関連する修飾子，および CDEC$ INIT_DEP_FWD 指示文も含めてサポートされません。

9.5.1.3 解釈方法の相違

以下の言語機能は，HP Fortran 77 と HP Fortran とで解釈方法が異なります。

- 乱数ジェネレータ (RAN)

  HP Fortran の RAN 関数は，同じランダム・シードに対して，HP Fortran 77 と異なる数値パターンを生成します (RAN 関数と RANDU 関数は，HP Fortran 77 との互換性を維持するために提供されます)。

- フォーマット付き入出力文でのホレリス定数

  次のいずれかの場合には，HP Fortran 77 と HP Fortran は異なる動作をします。

  – 2 つの異なる入出力文が，フォーマット指定子として同じ CHARACTER PARAMETER 定数を参照する場合。次の例を参照してください。

```
CHARACTER*(*) FMT2
PARAMETER (FMT2='(10Habcdefghij)')
READ (5, FMT2)
WRITE (6, FMT2)
```

  – 2 つの異なる入出力文が，それぞれのフォーマット指定子として同じ文字定数を使用する場合。次の例を参照してください。

```
READ (5, '(10Habcdefghij)')
WRITE (6, '(10Habcdefghij)')
```

  HP Fortran 77 では，READ 文によって読み取られた値が WRITE 文の出力になります (FMT2 は無視されます)。一方，HP Fortran では，WRITE 文の出力はabcdefghijになります (つまり，READ 文によって読み取られた値は WRITE 文によって書き込まれる値に影響を与えません)。

9.5.2 コマンド行修飾子

HP Fortran と HP Fortran 77 では大部分の修飾子が共通ですが，一部の修飾子はどちらか一方のプラットフォームでしか使用できません。ここでは，HP Fortran と HP Fortran 77 のコマンド行の修飾子の相違点を要約します。

HP Fortran のコンパイル・コマンドとオプションについての詳細は，『DEC Fortran User Manual for OpenVMS AXP Systems』を参照してください。HP Fortran 77 のコンパイル・コマンドとオプションについての詳細は，『DEC Fortran User Manual for OpenVMS VAX Systems』を参照してください。

VAX システムまたは I64 システムでコンパイルを開始するには，FORTRAN コマンドを使用します。I64 システムでは，F90 コマンドを使用して HP Fortran 90 コンパイラによるコンパイルを開始します。

9.5.2.1 HP Fortran for OpenVMS I64 固有の修飾子

表 9–3 は，HP Fortran コンパイラの修飾子のうち，HP Fortran 77 ではオプションがなく，HP Fortran 77 バージョン 6.4 でサポートされていない修飾子を示しています。

表 9–3 HP Fortran 77 にない HP Fortran の修飾子

修飾子	説明
/BY_REF_CALL	文字定数の実引数に数値仮引数を関連付けることができる (HP Fortran for OpenVMS VAX Systems で認められている)。
/CHECK=FP_EXCEPTIONS	IEEE 浮動小数点例外値に関するメッセージが実行時に報告されるかどうかを制御する。
/DOUBLE_SIZE	DOUBLE PRECISION 宣言を REAL*8 ではなく REAL*16 にする。
/FAST	実行時の性能を向上する複数の修飾子を設定する。
/FLOAT	メモリ内で浮動小数点データに対して使用する形式 (REAL または COMPLEX) を制御する。たとえば，KIND=4 データに対して VAX の F 浮動小数点と IEEE S 浮動小数点のどちらを使用するのかや，KIND=8 データに対して VAX G 浮動小数点，VAX D 浮動小数点，IEEE T 浮動小数点のどれを使用するのかを選択する。HP Fortran 77 for OpenVMS VAX Systems では/[NO]G_FLOATING 修飾子を使用できる。
/GRANULARITY	共用データのデータ・アクセスの粒度を制御する。
/IEEE_MODE	IEEE データに対して浮動小数点例外の処理方法を制御する。
/INTEGER_SIZE	INTEGER 宣言と LOGICAL 宣言のサイズを制御する。
/NAMES	外部名を大文字に変換するのか，小文字に変換するのか，または元のまま保存するのかを制御する。
/OPTIMIZE	/OPTIMIZE 修飾子は，INLINE キーワード，LOOPS キーワード，TUNE キーワード，UNROLL キーワード，およびソフトウェア・パイプラインをサポートする。
/REAL_SIZE	REAL 宣言と COMPLEX 宣言のサイズを制御する。
/ROUNDING_MODE	IEEE データに対して浮動小数点演算を丸める方法を制御する。
/SEPARATE_COMPILATION	HP Fortran コンパイラが次の処理を行うかどうかを制御する。 • HP Fortran 77 と同様に，個々のコンパイル単位を独立したモジュールとしてオブジェクト・ファイルに配置する (/SEPARATE_COMPILATION)。 • コンパイル単位を 1 つのモジュールとしてオブジェクト・ファイルにまとめる (/NOSEPARATE_COMPILATION，デフォルトの設定)。この結果，プロシージャ間の最適化が促進される。

(次ページに続く)

表 9–3 (続き) HP Fortran 77 にない HP Fortran の修飾子

修飾子	説明
/SYNTAX_ONLY	構文チェックだけを行い，オブジェクト・ファイルを作成しないことを要求する。
/WARNINGS	いくつかのキーワードは HP Fortran 77 で使用できない。
/VMS	HP Fortran が特定の HP Fortran 77 表記法を使用することを要求する。

9.5.2.2 HP Fortran 77 固有の修飾子

ここでは， HP Fortran 77 コンパイラの修飾子のうち， HP Fortran には対応する修飾子がないものをまとめます。

表 9–4 は， HP Fortran 77 バージョン 6.4 固有のコンパイル修飾子を示しています。

表 9–4 HP Fortran で使用できない HP Fortran 77 の修飾子

HP Fortran 77 の修飾子	説明
/BLAS=(INLINE,MAPPED)	HP Fortran 77 が Basic Linear Algebra Subroutines (BLAS) を認識し，これらをインラインまたはマッピングするかどうかを指定する。 HP Fortran 77 でのみ使用できる。
/CHECK=ASSERTIONS	アサーション・チェックの有効と無効を切り替える。 HP Fortran 77 でのみ使用できる。
/DESIGN=[NO]COMMENTS /DESIGN=[NO]PLACEHOLDERS	設計情報を確認するためにプログラムを解析する。
/DIRECTIVES=DEPENDENCE	指定されたコンパイラ指示文をコンパイル時に使用するかどうかを指定する。 HP Fortran 77 でのみ使用できる。
/PARALLEL=(MANUAL または AUTOMATIC)	並列処理をサポートする。
/SHOW=(DATA_ DEPENDENCIES,LOOPS)	リスト・ファイルに次の情報を出力するかどうかを制御する。 • 依存解析の対象とならないループと，ベクタ化または自動分解を禁止するデータ依存性に関する診断情報 (DATA_ DEPENDENCIES)。 • コンパイル後のループ構造に関するレポート (LOOPS)。 キーワード DATA_DEPENDENCIES および LOOPS は， HP Fortran 77 for OpenVMS VAX Systems でのみ使用できる。
/VECTOR	ベクタ処理を要求する。 HP Fortran 77 でのみ使用できる。
/WARNINGS=INLINE	コンパイラが組み込みルーチンに対する参照のインライン・コードを生成できないときに，情報診断メッセージを出力するかどうかを制御する。 HP Fortran 77 でのみ使用できる。

要求された (手動の) 分解に関連するすべての CPAR$指示文と特定の CDEC$指示文，およびそれに関連する修飾子またはキーワードは HP Fortran 77 固有です。『*DEC Fortran Language Reference Manual*』を参照してください。

HP Fortran 77 のコンパイル・コマンドとオプションについての詳細は，『DEC Fortran User Manual for OpenVMS VAX Systems』を参照してください。

9.5.3 トランスレートされた共有イメージとの相互操作性

HP Fortran を使用して，イメージ起動時 (実行時) にトランスレートされたイメージと相互操作可能なイメージを作成できます。

トランスレートされた共有イメージの使用を可能にするには，次の操作を実行します。

- FORTRAN または F90 コマンド行に/TIE 修飾子を指定します。
- LINK コマンド行で/NONATIVE_ONLY 修飾子を指定します。

作成された実行イメージには，最終的な実行イメージが共有イメージと相互操作できるようにするコードが含まれます。たとえば， HP Fortran 77 RTL (FORRTL) が HP Fortran RTL (DEC$FORTRTL) と協調動作できるようにするためのコードが含まれます。ネイティブ・プログラム (HP Fortran RTL) とトランスレートされたプログラム (HP Fortran 77 RTL) は，ファイルをオープンした RTL がそのファイルのクローズも実行するのであれば，同じユニット番号に対して入出力を実行できます。

プログラムは，完全な (fac$xxxx) 名前によってルーチンを呼び出すのではなく，内部名 (接頭辞を除く名前) を使用しなければなりません。ただし，*fac$xxxx*という名前を使用できる 1 つの例外があります。トランスレートされたイメージ・プログラムでは， FOR$RAB システム関数を EXTERNAL として宣言できます。ネイティブ I64 プログラムでは， FOR$RAB を内部関数として使用しなければなりません。

9.5.4 HP Fortran 77 データの移植

レコード・タイプは， HP Fortran 77 でも HP Fortran でも同じです。
必要な場合には， EXCHANGE コマンドと/NETWORK 修飾子および
/TRANSFER=BLOCK 修飾子を使用してデータを移植してください。コピー操作でファイルを Stream_LF 形式に変換するには，/TRANSFER=BLOCK の代わりに/TRANSFER=(BLOCK,RECORD_SEPARATOR=LF) を使用するか，または EXCHANGE コマンドで/FDL 修飾子を使用して，レコード・タイプや他のファイル属性を変更します。

フォーマッティングされていない浮動小数点データを変換しなければならない場合には， HP Fortran 77 プログラム (VAX ハードウェア) が REAL*4 または COMPLEX*8 データを F 浮動小数点形式で格納し， REAL*8， REAL*16，または COMPLEX*16 データを D 浮動小数点または G 浮動小数点形式で格納し， REAL*16 データを H 浮動小数点形式で格納することを念頭に置いてください。 HP Fortran for OpenVMS Alpha プログラム (Alpha ハードウェアで実行されるプログラム) は， REAL*4， REAL*8， REAL*16， COMPLEX*8，および COMPLEX*16 データをそれぞれ，表 9–5 に示す形式のいずれかで格納します。

表 9–5 VAX システムと I64 システムでの浮動小数点データ

データ宣言	VAX 形式	I64 形式
REAL*4 と COMPLEX*8	VAX F 浮動小数点形式	IEEE S 浮動小数点または VAX F 浮動小数点[1]形式
REAL*8 と COMPLEX*16	VAX D 浮動小数点または G 浮動小数点形式	IEEE T 浮動小数点，VAX D 浮動小数点[1]，または VAX G 浮動小数点形式
REAL*16	VAX H 浮動小数点	X 浮動小数点。変換が必要。一般には，/CONVERT 修飾子または関連する OPTION 文，論理名，OPEN 文の /CONVERT キーワードのいずれかを使用する。RTL ルーチン CVT$CONVERT_FLOAT を使用することもできる。

[1]I64 システムでは，多数の演算を実行するときに VAX の F 浮動小数点，D 浮動小数点，または G 浮動小数点形式を使用することは望ましくない。このような場合には，HP Fortran for OpenVMS I64 の変換ルーチンを使用する変換プログラムで，F 浮動小数点形式を IEEE S 浮動小数点形式に変換し，D 浮動小数点形式と G 浮動小数点形式を IEEE T 浮動小数点形式に変換することを考慮しなければならない。

9.6 HP Pascal for I64 Systems と HP Pascal for VAX Systems の互換性

ここでは，HP Pascal と他の HP Pascal コンパイラを比較し，VAX システムと I64 システムの HP Pascal の違いを示します。これらの機能の完全な説明については，『DEC Pascal Language Reference Manual』を参照してください。

9.6.1 使用されない外部シンボル

OpenVMS VAX では，HP Pascal コンパイラは，最終的な命令ストリーム中にあるシンボルに対してだけグローバル・シンボル定義 (GSD) を生成します。OpenVMS I64 システムでは，HP Pascal は，プログラム中で使用されているかどうかにかかわらず，すべての外部定義に対して GSD を生成します。これにより，リンク時に追加のオブジェクトが必要になったり，一部のプログラムを実行する際に追加の共有イメージが必要になる場合があります。

このようなプログラムは，実際にプログラム中で使用している場合にだけ該当する外部定義を宣言するように変更してください。

9.6.2 プラットフォームにまたがった環境ファイル

コンパイラは，同じターゲット・プラットフォームのコンパイラが作成した環境ファイルだけを継承します。たとえば，HP Pascal for OpenVMS VAX が生成した環境ファイルを HP Pascal for OpenVMS I64 コンパイラで継承することはできません。

9.6.3 列挙型と論理型のデフォルトのサイズ

パックされていない構造体での列挙型と論理型のデフォルトのサイズは，I64 ではロングワードです。VAX システムでは，論理型と小さな列挙型のデフォルトはバイトで，大きな列挙型はワードです。I64 システムで VAX と同じ動作をさせる必要がある場合は，/ENUMERATION_SIZE=BYTE 修飾子または[ENUMERATION_SIZE(BYTE)]属性を使用するか，該当するフィールドや構成要素に対して，個別に[BYTE]属性または[WORD]属性を記述します。OpenVMS VAX コンパイラでのデフォルトは，互換性を維持するために/ENUMERATION_SIZE=BYTE のままです。

9.6.4 パックされていない配列とレコードのデフォルトのデータ・レイアウト

I64 システムでは，デフォルトのデータ・レイアウトは自然なアラインメントです。そのため，レコード・フィールドと配列の構成要素は，サイズに応じた境界にアラインされます。VAX システムのデフォルトのアラインメント規則では，このようなフィールドは次のバイト境界に割り当てられます。I64 システムで VAX と同じ動作をさせる必要がある場合は，/ALIGN=VAX 修飾子または[ALIGN(VAX)]属性を使用します。

9.6.5 デフォルトの浮動小数点形式

I64 システムのデフォルトの浮動小数点形式は IEEE 形式です。コンパイラは，REAL データ型に対しては IEEE S 浮動小数点を使用し，DOUBLE データ型に対しては IEEE T 浮動小数点，QUADRUPLE データ型に対しては X 浮動小数点を使用します。VAX システムでは，デフォルトの浮動小数点形式は VAX 形式です。OpenVMS I64 システムで VAX の浮動小数点形式を使用する必要がある場合は，/FLOAT=G_FLOAT 修飾子または/FLOAT=D_FLOAT 修飾子を使用するか，モジュール・レベル属性[FLOAT( )]を使用します。I64 システムで VAX 形式が要求されると，コンパイラは演算を行う前に VAX 形式の値を IEEE 形式に変換します。その後，コンパイラはメモリに格納する前に値を元の VAX 形式に変換します。VAX 形式と IEEE 形式の違いにより，有効桁数と精度に若干違いが生じます。

9.6.6 IADDRESS と VOLATILE

IADDRESS 組み込み機能では，パラメータが VOLATILE 変数であるか，VOLATILE パラメータであるか，ルーチン・エントリ・ポイントであることを仮定しています。ADDRESS 組み込み機能と異なり，IADDRESS 組み込み機能は，パラメータに VOLATILE 属性がなくても警告を生成しません。

VAX システムでは，HP Pascal コンパイラは，変数が宣言されているルーチン全体に対して変数が存在するように変数を割り当てることがあります。このような状況では，組み込み機能 IADDRESS を使用して変数のアドレスを取得する処理は，期待ど

おりに動作します。通常は，アドレスは項目リストまたは同等の方法でシステム・サービスに渡されます。

I64 システムでは， HP Pascal コンパイラがスタックにデータを配置する際，より最適化されるように配置します。 VOLATILE 属性がないと，コンパイラは最低限の期間だけ変数を割り当てます。 IADDRESS を使用してアドレスを取得すると，システム・サービスがそのアドレスにデータを書き込む時点では，その変数はすでに存在せず，メモリに書き込むと別の変数が壊れる可能性があります。

要約すると，自動変数またはパラメータに対して IADDRESS 組み込み機能を使用する場合は， VOLATILE 属性を使用して正しい動作を保証しなければなりません。

9.6.7 値が大きな符号なし数値での INT のオーバフロー

VAX システムでは， INT 組み込み関数は，値が大きな符号なし数値を受け取ると，そのまま負の整数に変換します。しかし， HP Pascal に 64 ビット整数型を追加するにあたり，この動作は誤りであることが明らかとなりました。そのため，オーバフロー・チェックが有効な場合，実パラメータを INTEGER32 で表すことができない場合は， INT 組み込み関数は実行時エラーを通知するようになりました。

負の整数に変換したい大きな符号なし数値がある場合は，型キャストを使用して演算を実行しなければなりません。

9.6.8 バウンド・プロシージャ値

VAX システムでは，バウンド・プロシージャ値は 2 つのロングワードからなる構造体であり，エントリ・ポイントのアドレスとネストした環境を定義するためのフレーム・ポインタが格納されます。 HP Pascal では， PROCEDURE または FUNCTION パラメータとして，これら 2 ロングワードの構造体のいずれかを期待します。さらに，呼び出されたルーチンは，バウンド・プロシージャ値を受け取ったのか，単純なルーチン・アドレスを受け取ったのかを識別できる必要があります。仮ルーチン・パラメータ%IMMED にルーチンを渡す場合， HP Pascal はエントリ・ポイントのアドレスを渡します。

I64 システムでは，バウンド・プロシージャ値は単に特別な種類の関数記述子であり，隠れたジャケット・ルーチンを呼び出します。ジャケット・ルーチンがさらにフレーム・ポインタを初期化して，実際のルーチンを呼び出します。この構造により，別のルーチンを間接的に呼び出すルーチンは，バウンド・プロシージャ値に対して何も特別なことを実行する必要はありません。同様に，%IMMED によってルーチンを渡す場合 (またはルーチンの IADDRESS を要求する場合) は，%IMMED がないかのように関数記述子のアドレスが渡されます。 HP Pascal には，ルーチンの実際のコード・アドレスを取得する直接的な方法はありません。これは，対応する関数記述子がないと，一般にはルーチンの使い道がないためです。

9.6.9 整合配列パラメータ用のさまざまな記述子クラス

整合パラメータについては，HP Pascal は「記述子渡し」メカニズムを使用してルーチン間で受け渡しを行います。整合配列パラメータに関して，HP Pascal は，VAX システムでは CLASS_A 記述子を使用し，I64 システムでは CLASS_NCA 記述子を使用します。CLASS_NCA 記述子では，配列の構成要素にアクセスする際に，より効率の良いコードが生成されます。Pascal 用の CLASS_A 記述子を構成するフォーリン・ルーチンがある場合は，コードを調べて，変更が必要かどうかを確認する必要があります。いくつかの実パラメータでは，CLASS_A 記述子と CLASS_NCA 記述子は，DSC$B_CLASS フィールド (HP Pascal は参照しません) を除いて同じです。それ以外のパラメータでは，CLASS_NCA 記述子を生成するか，Pascal ルーチンの仮整合パラメータに明示的な CLASS_A 属性を追加しなければなりません。

9.6.10 OpenVMS I64 では利用できない Pascal の機能

OpenVMS VAX システム用の HP Pascal の以下の機能は，OpenVMS I64 用の HP Pascal では利用できません。

- 組み込みルーチン MFPR および MTPR
- DCL 修飾子/DESIGN=COMMENTS
- DCL 修飾子/SHOW=TABLE_OF_CONTENTS およびリスト・セクション
- DCL 修飾子/SHOW=INLINE およびリスト・セクション

これらの機能に代わる機能や回避策はありません。

OpenVMS VAX から OpenVMS I64 に移行する前に，/PLATFORM=OPENVMS_AXP 修飾子付きでプログラムをコンパイルすることができます。OPENVMS_AXP キーワードを使用しなければならない理由は，現在出荷中の OpenVMS VAX システム用の HP Pascal コンパイラは，/PLATFORM DCL 修飾子の OPENVMS_I64 キーワードを認識しないためです。OpenVMS VAX から OpenVMS I64 への移植に関する問題は，OpenVMS VAX から OpenVMS Alpha に移植する際の問題と基本的に同じです。唯一の違いは，デフォルトの浮動小数点形式です。

9.6.11 Pascal レコード・レイアウト・ガイド

HP Pascal キットには，OpenVMS VAX から OpenVMS I64 にプログラムを移植する際のデータ・レイアウトとデータ変換の問題について，詳しく説明したドキュメントが付属しています。ファイルは，SYS$HELP:PASCAL_RECORD_LAYOUT_GUIDE.MEMにあります。

A

アプリケーション評価チェックリスト

開発の履歴と計画

1. アプリケーションは現在，他のオペレーティング・システムまたはハードウェア・アーキテクチャで実行されていますか? ☐ YES ☐ NO

   その場合には，アプリケーションは現在， RISC システムで実行されていますか? ☐ YES ☐ NO

   [その場合には， OpenVMS I64 への移行は容易に実行できます。]

2. 移行後のアプリケーションの開発/保守計画はどのようになっていますか?

   a. 開発は行わない ☐ YES ☐ NO

   b. 保守リリースのみ ☐ YES ☐ NO

   c. 機能を追加または変更する ☐ YES ☐ NO

   d. VAX ソースと I64 ソースを個別に保守する ☐ YES ☐ NO

   [a に対する回答が "YES"の場合には，アプリケーションのトランスレートを検討してください。 b または c に対する回答が "YES"の場合には，アプリケーションの再コンパイルと再リンクによって得られる利点を評価してください。ただし，トランスレーションも可能です。 VAX ソースと I64 ソースを個別に保守する場合，つまり， d に対する回答が "YES"の場合には，相互操作性と整合性に関する問題を考慮しなければなりません。特に，異なるバージョンのアプリケーションが，同じデータベースにアクセスする場合には，このことを考慮しなければなりません。]

外部的な依存性

3. アプリケーションの開発環境を準備するために，どのようなシステム構成 (CPU，メモリ，ディスク) が必要ですか? _____

   [この質問は移行に必要な資源の計画を立てるのに役立ちます。]

4. アプリケーションの典型的なユーザ環境を準備するために，どのようなシステム構成 (CPU，メモリ，ディスク) が必要ですか? インストール検証プロシージャ，回帰テスト，ベンチマーク，作業負荷も含めて考慮してください。 _____

   [この質問は，ユーザ環境全体を OpenVMS I64 で実現できるかどうかを判断するのに役立ちます。]

5. アプリケーションで特殊なハードウェアが必要ですか? ☐ YES ☐ NO

[この質問は，必要なハードウェアを OpenVMS I64 で使用できるかどうかと，アプリケーションにハードウェア固有のコードが含まれているかどうかを判断するのに役立ちます。]

6. a. アプリケーションは現在， OpenVMS のどのバージョンで実行されていますか? _____

   b. アプリケーションは OpenVMS VAX バージョン 6.1 で実行されていますか? ☐ YES ☐ NO

   c. アプリケーションは OpenVMS I64 で利用できない機能を使用していますか? ☐ YES ☐ NO

   [OpenVMS I64 への移行の基礎となるのは， OpenVMS VAX バージョン 6.1 です。質問 c に対する回答が "YES"の場合には，アプリケーションは OpenVMS I64 でまだサポートされていない機能を使用している可能性があり，また現在のバージョンの OpenVMS I64 と互換性のない OpenVMS RTL や他の共有イメージに対してリンクされている可能性があります。]

7. アプリケーションを実行するために，レイヤード・プロダクトが必要ですか?

   a. 弊社が提供するプロダクト (コンパイラ RTL 以外のプロダクト): ☐ YES ☐ NO

   b. 他社が提供するプロダクト: ☐ YES ☐ NO

   [a に対する回答が "YES"のときに， OpenVMS I64 で弊社のレイヤード・プロダクトが提供されているかどうかがよくわからない場合には，弊社の担当者に質問してください。 b に対する回答が "YES"の場合には，他社製品の提供ベンダーに問い合わせてください。]

アプリケーションの構造

8. アプリケーションのサイズは?

   a. モジュール数は? _____

   b. コードの行数または K バイト数は? _____

   c. 必要なディスク領域は? _____

   [この質問は，移行のために必要な作業量と資源の "サイズ "を判断するのに役立ちます。]

9. a. アプリケーションを構成する，すべてのソース・ファイルを入手できますか? ☐ YES ☐ NO

   b. 弊社のサービスの使用を検討している場合には，弊社にソース・ファイルとビルド・プロシージャを提供することは可能ですか? ☐ YES ☐ NO

   [質問 a に対する回答が "YES"の場合，ソース・ファイルを入手できない部分は，トランスレーションが唯一の移行手段となります。質問 b に対する回答が "YES"の場合には，広範囲にわたって弊社の移行サービスを利用できます。]

10. a. アプリケーションはどの言語で作成されていますか? (複数の言語が使用されている場合には，各言語の割合を示してください。) _____

    [コンパイラがまだ提供されていない場合には，アプリケーションをトランスレートするか，別の言語で再作成しなければなりません。]

    b. VAX MACROを使用している場合には，その理由は何ですか? _____

	c. VAX MACROコードの機能を，高級言語コンパイラ，またはシステム・サービス (プロセス名を検索するための$GETJPI など) で実行できますか?	☐ YES	☐ NO
11.	a. アプリケーションをテストするために必要な回帰テストが準備されていますか?	☐ YES	☐ NO
	b. 準備されている場合には， DEC Test Manager が必要ですか?	☐ YES	☐ NO
	[a に対する回答が "YES"の場合には，これらの回帰テストの移行を考慮しなければなりません。 OpenVMS I64 の初期リリースでは，DEC Test Manager は提供されません。回帰テストでこのプロダクトが必要な場合には，弊社の担当者にご連絡ください。]		

VAX アーキテクチャへの依存

12.	a. アプリケーションで H 浮動小数点データ型を使用しますか?	☐ YES	☐ NO
	b. アプリケーションで D 浮動小数点データ型を使用しますか?	☐ YES	☐ NO
	c. アプリケーションで D 浮動小数点を使用する場合， 56 ビットの精度 (16 桁の有効桁数) が必要ですか，または 53 ビットの精度 (15 桁の有効桁数) で十分ですか?	☐ 56 ビット	☐ 53 ビット
	[a に対する回答が "YES"の場合には， H 浮動小数点の互換性を維持するためにアプリケーションをトランスレートするか，またはデータを G 浮動小数点， S 浮動小数点，または T 浮動小数点形式に変換しなければなりません。 b に対する回答が "YES"の場合には，アプリケーションをトランスレートして， VAX での D 浮動小数点の完全な 56 ビットの精度の互換性を維持するか， I64 システムが提供している 53 ビットの精度の D 浮動小数点形式を使用するか，またはデータを G 浮動小数点， S 浮動小数点，または T 浮動小数点のいずれかの形式に変換しなければなりません。]		
13.	a. アプリケーションで大量のデータ，またはデータ構造を使用しますか?	☐ YES	☐ NO
	b. データがバイト，ワード，またはロングワードでアラインされていますか?	☐ YES	☐ NO
	[a に対する回答が "YES"であり， b に対する回答が "NO"の場合には， I64 の最適な性能を実現するために，データを自然なアラインメントにすることを考慮しなければなりません。多くのプロセスで共用されるグローバル・セクションにデータが格納されている場合や，メイン・プログラムと AST との間でデータが共用される場合には，データを自然なアラインメントにしなければなりません。]		
14.	コンパイラがデータをアラインする方法に関して，アプリケーションで何らかの仮定を設定していますか (つまり，構造体がパックされていること，自然にアラインされていること，ロングワードでアラインされていることなどをアプリケーションで仮定していますか)?	☐ YES	☐ NO
	[回答が "YES"である場合には， I64 プラットフォームでのコンパイラの動作と， VAX プラットフォームでのコンパイラの動作の違いから発生する，移植性と相互操作性の問題を考慮しなければなりません。データ・アラインメントに関するコンパイラのデフォルトの設定はさまざまで，アラインメントを強制的に設定するためのコンパイラ・スイッチもさまざまです。通常， VAX システムでは，デフォルトのデータ・アラインメントはパック形式のアラインメントですが，I64 コンパイラのデフォルトは，可能なかぎり自然なアラインメントです。]		

15. a. アプリケーションで，ページ・サイズが 512 バイトであると仮定していますか? ☐ YES ☐ NO

b. アプリケーションで，メモリ・ページがディスク・ブロックと同じサイズであると仮定していますか? ☐ YES ☐ NO

[a に対する回答が "YES"の場合には， I64 のページ・サイズに対応できるように，アプリケーションを準備しなければなりません。 I64 のページ・サイズは 512 バイトよりはるかに大きく，システムごとに異なります。したがって，ページ・サイズを固定値として参照することは避け，可能なかぎり，メモリ管理システム・サービスと RTL ルーチンを使用してください。 b に対する回答が "YES"の場合には，ディスク・セクションをメモリにマッピングする$CRMPSC システム・サービスと，$MGBLSC システム・サービスに対するすべての呼び出しを確認し，これらの仮定を削除しなければなりません。]

16. アプリケーションで， OpenVMS システム・サービスを呼び出しますか? ☐ YES ☐ NO

特に，次の操作を実行するサービスを呼び出していますか?

a. グローバル・セクションを作成，またはマッピングするシステム・サービス (たとえば$CRMPSC，$MGBLSC，$UPDSEC) ☐ YES ☐ NO

b. ワーキング・セットを変更するシステム・サービス (たとえば$LCKPAG，$LKWSET) ☐ YES ☐ NO

c. 仮想アドレスを操作するシステム・サービス (たとえば$CRETVA，$DELTVA) ☐ YES ☐ NO

[これらの質問に対する回答が "YES"の場合には，コードを調べ，必要な入力パラメータが正しく指定されているかどうかを判断しなければなりません。]

17. a. アプリケーションで複数の協調動作プロセスを使用しますか? ☐ YES ☐ NO

使用する場合:

b. プロセスの数? _____

c. 使用するプロセス間通信方式? _____

☐ $CRMPSC ☐ メールボックス ☐ SCS ☐ その他

☐ DLM ☐ SHM， IPC ☐ SMG$ ☐ STR$

d. 他のプロセスとの間でデータを共用するために，グローバル・セクション ($CRMPSC) を使用する場合には，どのような方法でデータ・アクセスの同期をとりますか? _____

[この質問は，明示的に同期をとらなければならないかどうかを判断し，アプリケーションの各要素間で，同期を保証するのに必要な作業レベルを判断するのに役立ちます。一般に，高いレベルの同期方式を使用すれば，アプリケーションをもっとも容易に移行できます。]

18. アプリケーションは現在，マルチプロセッサ (SMP) 環境で実行されていますか? ☐ YES ☐ NO

[回答が "YES"の場合には，アプリケーションはすでに適切なプロセス間同期方式を採用している可能性が高いと言えます。]

19. アプリケーションで， AST (非同期システム・トラップ) メカニズムを使用していますか? ☐ YES ☐ NO

[回答が "YES"の場合には， AST とメイン・プロセスが，プロセス空間のデータを共用しているかどうかを判断しなければなりません。共用している場合には，明示的にこのようなアクセスの同期をとらなければなりません。]

20. a. アプリケーションに条件ハンドラが含まれていますか? ☐ YES ☐ NO

b. アプリケーションで，算術演算例外の即時報告が必要ですか? ☐ YES ☐ NO

[Intel Itanium アーキテクチャでは，算術演算例外はただちに報告されません。条件ハンドラが条件を修正しようとし，例外の原因となった命令シーケンスを再起動する場合には，ハンドラを変更しなければなりません。]

21. アプリケーションは特権モードで実行されますか，または SYS.STB に対してリンクされますか? ☐ YES ☐ NO

その場合は理由を示してください。 _____

[アプリケーションが OpenVMS エグゼクティブに対してリンクされるか，または特権モードで実行される場合には，ネイティブ I64 イメージとして実行されるように，アプリケーションを変更しなければなりません。]

22. 独自のデバイス・ドライバを作成しますか? ☐ YES ☐ NO

23. アプリケーションで，接続/割り込みメカニズムを使用しますか? ☐ YES ☐ NO

使用する場合には，どのような機能を使用しますか? _____

24. アプリケーションで，機械語を作成または変更しますか? ☐ YES ☐ NO

[OpenVMS I64 では，命令ストリームに書き込まれた命令が正しく実行されることを保証するには，多大な注意が必要です。]

25. アプリケーションのどの部分が性能に最も大きな影響を与えますか? それは入出力ですか，浮動小数点ですか，メモリですか，リアルタイム性ですか (つまり，割り込み待ち時間)。 _____

[この質問は，アプリケーションの各部分に対する作業に優先順位をつけるのに役立ち，お客様にとって最も意味のある性能向上を，弊社が計画するのに役立ちます。]

用語集

アラインメント
自然なアラインメントを参照。

不可分な命令 (atomic instruction)
1 つ以上の分割不能な操作で構成される命令であり，これらの操作は中断せずに 1 つの操作としてハードウェアで処理される。

不可分な操作
AST (非同期システム・トラップ) サービス・ルーチンなど，他のシステム・イベントによって中断することができない操作。不可分な操作は他のプロセスにとって 1 つの操作であるかのように見える。不可分な操作が開始されると，その操作は中断されずに必ず最後まで実行される。

読み取り/変更/書き込み (リード・モディファイ・ライト) 操作は通常， RISC マシンの命令レベルでは不可分な操作ではない。

バイト粒度
メモリ・システムの特性であり，隣接するバイトを異なるプロセスまたはプロセッサが同時に独立して書き込むことができる特性。

CISC
複雑命令セット・コンピュータを参照。

互換性
あるコンピュータ・システム (たとえば OpenVMS VAX) のために作成されたプログラムを，別のシステム (たとえば OpenVMS I64) で実行できる能力。

複雑命令セット・コンピュータ (CISC)
メモリ内の位置に対して直接実行される複雑な操作も含めて，個別の命令で複雑な操作を実行するコンピュータ。このような操作の例としては，複数バイトのデータ移動や部分文字列検索を実行する命令がある。 CISC コンピュータは通常，*RISC*コンピュータ (**縮小命令セット・コンピュータ**) の反対語である。

同時実行/並列処理
複数のエージェントが共用オブジェクトに対して操作を同時に実行すること。

粒度
1 つの命令によって読み書きできるデータ・サイズ，つまり，個別に読み書きを実行できるデータ・サイズを定義する記憶システムの特性。 VAX システムの粒度はバイト粒度またはマルチバイト粒度であるが，ディスク・システムの粒度は，通常 512 バイト以上である。

イメージ・セクション

イメージを仮想メモリに割り当てるときの単位となる，同じ属性を持つプログラム・セクションの集合。この場合属性とは，たとえば読み取り専用アクセス属性，読み取り/書き込みアクセス属性，固定アドレス属性，再配置可能属性などである。

インターロック命令

インターロック命令は，マルチプロセシング環境で 1 つの中断されない操作として完全な結果を保証できる方法で動作を実行する。インターロック命令が終了するまでの間，他の衝突する可能性のある操作はブロックされるため，インターロック命令は性能を低下させる可能性がある。

ロード/ストア・アーキテクチャ

データが最初にプロセッサ・レジスタにロードされ，操作された後，最終的にメモリに格納されるようなマシン・アーキテクチャ。ロード/ストア以外のメモリ操作は，このような命令セットには準備されていない。

ロングワード

任意のアドレッシング可能なバイト境界から始まる連続した 4 バイト (32 ビット)。各ビットには右から左に 0 〜 31 の番号が付けられる。ロングワードのアドレスは最下位ビット (ビット 0) を含むバイトのアドレスである。アドレスが 4 で割り切れる場合には，ロングワードは**自然にアラインされる**という。

複数命令発行

1 つのクロック・サイクルで複数の命令を出すこと。

自然なアラインメント

データ・アドレスがデータ・サイズ (バイト数) で割り切れるメモリ内のデータ。たとえば，自然にアラインされたロングワードは 4 で割り切れるアドレスを持ち，自然にアラインされたクォドワードは 8 で割り切れるアドレスを持つ。構造体のすべてのメンバが自然にアラインされている場合には，その構造体も自然にアラインされていると言う。

ページ・サイズ

システムのハードウェアが補助記憶との間でアドレス・マッピング，共用，保護，および移動のための単位として取り扱うバイト数。

ページレット

Itanium 環境で， 512 バイト単位のメモリを指す表現。 OpenVMS I64 システムでは，いくつかの DCL コマンドとユーティリティ・コマンド，システム・サービス，およびシステム・ルーチンは，必要なメモリとクォータをページレット単位で入力として受け付け，出力として提供する。この結果，これらの構成要素の外部インタフェースは VAX システムの外部インタフェースと互換性を持つが， OpenVMS I64 は，内部的には必ず CPU メモリのページ・サイズの整数倍でメモリを管理する。

PALcode

特権付きアーキテクチャ・ライブラリを参照。

特権付きアーキテクチャ・ライブラリ (PAL)
特定のオペレーティング・システム固有の命令を実行するための呼び出し可能ルーチンを登録したライブラリ。特殊な命令でルーチンを呼び出し，これらは中断せずに実行される。

プロセッサ・ステータ (PS)
Alpha システムでは，**クォドワード**の情報で構成される特権付きプロセッサ・レジスタであり，現在のアクセス・モード，現在の割り込み優先順位レベル (IPL)，スタック・アラインメント，複数の予約フィールドなどを含む。

プロセッサ・ステータス・ロングワード (PSL)
VAX システムで，1 ワードの特権付きプロセッサ・ステータスと，**プロセッサ・ステータス・ワード**自体で構成される特権付きプロセッサ・レジスタ。特権付きプロセッサ・ステータス情報には，現在の割り込み優先順位レベル (IPL)，前のアクセス・モード，現在のアクセス・モード，割り込みスタック・ビット，トレース・トラップ・ペンディング・ビット，互換モード・ビットなどが含まれる。

プロセッサ・ステータス・ワード (PSW)
VAX システムで，**プロセッサ・ステータス・ロングワード**の下位ワード。プロセッサ・ステータス情報には条件コード (キャリ，オーバフロー， 0，負)，算術演算トラップ・イネーブル・ビット (整数オーバフロー， 10 進オーバフロー，浮動小数点アンダフロー)，およびトレース・イネーブル・ビットが含まれる。

プログラム・カンウタ (PC)
CPU の中で，次に実行される命令の仮想アドレスを保持している部分。現在の大部分の CPU はプログラム・カウンタをレジスタとして実現している。プログラマは命令セットを通じてこのレジスタを参照できる。

クォドワード
任意のアドレッシング可能なバイト境界から始まる，連続した 4 ワード (64 ビット)。各ビットには右から左に 0 ～ 63 の番号が付けられる。クォドワードのアドレスは最下位ビット (ビット 0) を含むワードのアドレスである。アドレスが 8 で割り切れる場合には，クォドワードは**自然にアラインされている**。

クォドワード粒度
メモリ・システムの特性であり，隣接する**クォドワード**を異なるプロセスまたはプロセッサが同時に個別に書き込むことができる特性。

読み取り/変更/書き込み操作
メイン・メモリのデータを， 1 つの割り込み不可能な操作として読み取り，変更し，書き込むハードウェア操作。

読み取り/書き込みの順序
1 つの CPU のメモリに対する読み取り，書き込みなどの操作が，実行エージェント (密接に結合されたシステム内の個々の CPU や装置) から確認できるようになる順序。

縮小命令セット・コンピュータ (RISC)
命令セットの複雑さが削減されたコンピュータ。ただし，命令の数は必ずしも削減されているとは限らない。 RISC アーキテクチャでは通常，特定の操作を実行するために，*CISC*アーキテクチャより多くの命令を必要とする。これは，各命令が CISC 命令より少ない作業しか実行しないからである。

RISC
縮小命令セット・コンピュータを参照。

同期
マルチプロセシング環境やユニプロセシング環境で共用データを使用して動作するときに，きちんと定義された予測可能な結果が得られるように，一部の共用資源に対するアクセスを制御する方法。

トランスレートされたコード
トランスレートされたイメージ内のネイティブ OpenVMS I64 オブジェクト・コード。トランスレートされたコードとしては，次のコードがある。

- 元のイメージの対応する VAX コードの動作を再現する OpenVMS I64 コード
- *Translated Image Environment (TIE)*の呼び出し

トランスレートされたイメージ
VAX イメージのオブジェクト・コードの**トランスレーション**によって作成された OpenVMS I64 の実行イメージまたは共有イメージ。トランスレートされたイメージは，トランスレーションの元になる VAX イメージと同じ機能を実行し，**トランスレートされたコード**とオリジナル・イメージの両方を含む。*VAX Environment Software Translator*を参照。

Translated Image Environment (TIE)
トランスレートされたイメージの実行をサポートするネイティブ Alpha 共有イメージ。 TIE はネイティブ Alpha システムとのすべてのやりとりを処理する。また，VAX の状態を管理し，例外処理や AST の実行要求，複雑な VAX 命令など， VAX の機能をエミュレートし，トランスレートされていない VAX 命令を解釈することにより，トランスレートされたイメージに対して OpenVMS VAX に類似した環境を提供する。

トランスレーション
VAX バイナリ・イメージを OpenVMS I64 イメージに変換する処理。変換されたイメージは Alpha システムで TIE の援助によって実行される。変換は静的な処理であり，できるだけ多くの VAX コードがネイティブ Alpha 命令に変換される。トランスレートされなかった VAX コードに対しては， TIE が実行時に解釈する。

VEST
*VAX Environment Software Translator*を参照。

VAX Environment Software Translator (VEST)

ソフトウェア移行用のツールであり，VAX の実行イメージと共有イメージを，Alpha システムで実行されるトランスレートされたイメージに**トランスレート**する。**トランスレートされたイメージ**を参照。

ワード粒度

メモリ・システムの特性であり，隣接するワードを異なるプロセスまたはプロセッサが同時に個別に書き込むことができる特性。

書き込み可能グローバル・セクション

プロセス間通信で使用するためにシステム内のすべてのプロセスが使用できるデータ構造 (たとえば FORTRAN のグローバル・コモン) や共有イメージ・セクション。

索引

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

X

ア

イ

エ

オ

カ

キ

ク

ケ

コ

サ

シ

ス

セ

ソ

タ

チ

テ

ト

ナ

ネ

ハ

ヒ

フ

へ

ホ

マ

メ

モ

ユ

ヨ

ラ

リ

レ

ロ

ワ