OpenVMS Alpha
オペレーティング・システム

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

AA–R1E8B-TE

1999 年 4 月

本書は OpenVMS VAX 版アプリケーションの OpenVMS Alpha 版を作成する方法について説明しています。

改訂／更新情報: 本書は OpenVMS V7.1 『OpenVMS VAX から OpenVMS Alpha へのアプリケーションの移行』の改訂版です。

ソフトウェア・バージョン: OpenVMS Alpha V7.2
OpenVMS VAX V7.2

コンパックコンピュータ株式会社

1999 年 4 月

本書は CD-ROM でも提供しています。

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 Alpha システムまたは複合アーキテクチャ・クラスタに移行する開発者を支援できるように設計されています。

対象読者

本書は，高級プログラミング言語または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・コードの移行を担当される経験の豊富なソフトウェア・エンジニアの方々を対象にしています。

本書の構成

本書は次の章で構成されています。

- 第 1 章では， OpenVMS と VAX アーキテクチャおよび Alpha アーキテクチャの関係の概要を示し，アプリケーションを VAX システムから Alpha システムに移行する処理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 Alpha と OpenVMS VAX の互換性が高い分野
  - Alpha アーキテクチャと他の RISC アーキテクチャおよび VAX アーキテクチャとの比較
  - 移行処理の各段階の概要
  - おもな 2 種類の移行パス，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
  - 弊社から提供される移行サポート
- 第 2 章では， 2 種類の移行パスの相違点について考慮し，アプリケーションを移行するときにどのパスを使用するかを選択するため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識別する方法と，これらの相違点を解決するために必要な処理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の手順について説明します。
- 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処理の概要について説明します。

- 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可能性がある場合，それを取り扱う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可能性がある場合，それを取り扱う方法について説明します。
- 第 7 章では，アライメントの検討事項も含めて， Alpha 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， VAX の条件処理機能にアプリケーションが依存している可能性がある場合，それを取り扱う方法について説明します。
- 第 9 章では， Alpha 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説明します。
- 第 10 章では，トランスレートされた VAX イメージを呼び出したり，そのイメージから呼び出すことができるネイティブな Alpha イメージを作成する方法について説明します。
- 第 11 章では， Alpha システムの Ada， C， COBOL， FORTRAN， Pascal プログラミング言語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 Alpha に移行するために，アプリケーションを評価するときに使用できるチェックリストを示します。

参考文献

本書は OpenVMS VAX システムから OpenVMS Alpha システムへの移行について説明した複数のマニュアル・セットの一部です。このマニュアル・セットには，本書の他に次のマニュアルが含まれています。

- 『Porting VAX MACRO Code to OpenVMS Alpha』では， OpenVMS Alpha 用の MACRO–32 コンパイラを使用して，VAX MACROコードを Alpha システムに移植する方法について説明します。コンパイラの機能について説明し， VAX MACRO コードを移植する方法を示し，移植不可能なコーディング方式を示し，このようなコーディング方式に代わる方法を示します。このマニュアルではまた，参照情報の部分でコンパイラの修飾子，指示文，組み込み関数について詳しく説明し， Alpha システムに移植するために作成されるシステム・マクロについても説明します。
- 『Creating an OpenVMS Alpha 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 デバイス・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，マクロについても記述されています。

さらに，『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も説明します。このマニュアルはオプションのレイヤード製品である DECmigrate for OpenVMS Alpha に同梱されており，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処理をサポートします。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な実行時ライブラリとトランスレートされた実行時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参照情報も示します。

また，次の一般プログラミング・マニュアルも参照してください。

- 『VAX Architecture Reference Manual』
- 『Alpha Architecture Reference Manual』
- 『VAX/VMS Internals and Data Structures』
- 『OpenVMS AXP Internals and Data Structures』
- 『OpenVMS Programming Concepts Manual』
- 『OpenVMS Programming Interfaces: Calling a System Routine』
- 『Guide to DECthreads』

OpenVMS 製品とサービスの詳細については，Digital OpenVMS World Wide Web サイトにアクセスしてください。URL は次のとおりです。

`http://www.openvms.digital.com`

本書で使用する表記法

本書では，OpenVMS Alpha は OpenVMS Alpha オペレーティング・システムを指します。

OpenVMS AXP オペレーティング・システムはバージョン 6.2 から OpenVMS Alpha オペレーティング・システムに名称が変更されました。OpenVMS AXP あるいは AXP の表記は OpenVMS Alpha または Alpha と同じ意味です。

VMScluster システムは，現在 OpenVMS クラスタ・システムと同じ意味を示します。OpenVMS クラスタまたはクラスタは本書では，VMSclusters と同じ意味です。

本書では，DECwindows および DECwindows Motif はすべて DECwindows Motif for OpenVMS ソフトウェアを意味します。

また，本書では次の表記法も使用しています。

表記法	意味
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または GOLD x	PF1 xまたは Gold xという表記は， PF1 または GOLD 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。
	EVE コマンドでは， GOLD キーのあとにスラッシュ(/)，ダッシュ(–)，またはアンダースコア(_)を区切り文字として使用できます。
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。
. . .	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継くことを示します。
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

多くのアプリケーションにとって，OpenVMS VAX から OpenVMS Alpha への移行 (migration) 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，Alpha 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，Alpha 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，およびもっと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 Alpha システムの互換性

OpenVMS Alpha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 (compatibility) 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とって，OpenVMS Alpha は OpenVMS VAX と同じインタフェースを備えています。プログラマにとっての目標は，「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です。

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 OpenVMS Alpha 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・インターフェイスであり，OpenVMS Alpha 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 Alpha 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 Alpha システムでもまったく変更せずに動作します。しかし，ビルド・プロシージャなどのある特定のコマンド・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，特に共用可能イメージ (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 インターフェイ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，おもな例外が 1 つあります。それはデバイス構成管理機能で， OpenVMS 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 Alpha 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。[1] 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・サービスとランタイム・ライブラリ・ルーチン (メモリ管理システムと例外処理サービス) は， VAX システムと Alpha 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。
- 一部のランタイム・ライブラリ・ルーチンは VAX アーキテクチャに密接に関係しており， Alpha 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$ESTABLISH	Alpha システムでは，類似する機能をコンパイラがサポートする
LIB$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$REVERT	Alpha システムではコンパイラがサポートする
LIB$SIM_TRAP	VAX 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 に置換されている

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， VEST (VAX Environment Software Translator) を使ってトランスレートし， OpenVMS

[1] バージョン 7.0 では， OpenVMS Alpha は 64 ビット・アドレッシングをサポートするために，多くのシステム・サービスと RTL ルーチンを提供します。これは VAX システムでは提供されません。したがって VAX から Alpha への移行の問題にはならないため，本書では説明しません。

Alpha 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，第 3.2.2.1 項と 『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フォーマットは， VAX システムと Alpha システムとで同じです。しかし， ODS-1 ファイルは OpenVMS Alpha で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 Alpha Alpha アーキテクチャでもそのまま使用できます。しかし，システム全体の性能を向上するために， H 浮動小数点 (H_floating) と完全な精度の D 浮動小数点 (D_floating) のハードウェアによるサポートはなくなりました。

Alpha ハードウェアは D 浮動小数点データを処理のために G 浮動小数点に変換します。 VAX システムでは， D 浮動小数点は 56 ビット (D56) であり， 16 桁の精度です。 Alpha システムでは， D 浮動小数点は 53 ビット (D53) であり， 15 桁の精度です。

H 浮動小数点データ型と D 浮動小数点データ型は通常， G 浮動小数点または IEEE フォーマットのいずれかに変換されます。しかし， H 浮動小数点が必要な場合や， D56 (56 ビットの D 浮動小数点) の精度が必要な場合には，アプリケーションの一部をトランスレートしなければなりません。

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 Alpha 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 Alpha 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
  - FDDI
- プロトコル
  - DECnet (バージョン 7.1 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
  - TURBOchannel
  - SCSI
  - Ethernet
  - CI
  - DSSI
  - XMI
  - Futurebus/Plus
  - VME
  - PCI

1.2 VAX アーキテクチャと Alpha アーキテクチャの相違点

VAX アーキテクチャは強力で柔軟な命令を備えた，複合命令セット・コンピュータ (CISC)・アーキテクチャであり， VAX システム・ファミリ全体で使用されています。統一アーキテクチャの VAX ファミリを， OpenVMS オペレーティング・システムと組み合わせて使用すれば，アプリケーションを VAXstation で開発し，小型 VAX システムでプロトタイプを作成し，大型 VAX プロセッサのプロダクション環境で使用したり，フォールト・トレラントな VAXft プロセッサで実行することができます。VAX システムのアプローチの利点は，個別のソルーションを変更し，大規模な問題全体のソルーションとして容易に統合できるということです。 VAX プロセッサのハードウェア設計は特に，可用性の高いアプリケーションに適しています。たとえば，重要な使命を担うビジネス業務を実行するための，信頼性の高いアプリケーションや，様々な分散クライアント/サーバ環境でのサーバ・アプリケーションを実行するのに適しています。

弊社が実現した Alpha アーキテクチャは，高性能の縮小命令セット・コンピュータ (RISC)・アーキテクチャであり， 1 つのチップで 64 ビット処理を実現できます。 64 ビットの仮想アドレスと物理アドレス， 64 ビットの整数， 64 ビットの浮動小数点数値を処理します。 64 ビット処理は特に，高い性能ときわめて大きいアドレッシング空間を必要とするアプリケーションにとって役立ちます。たとえば， Alpha プロセッサは，グラフィック処理アプリケーションや，膨大な数値を処理する経済予測や気象予報などのソフトウェア・アプリケーションに適しています。これらは，イメージ処理，マルチメディア，ビジュアライゼーション，シミュレーション，モデリングなどの処理を必要とします。

Alpha アーキテクチャはスケーラブルでオープンなアーキテクチャとして設計されています。シングル・チップによる手のひらサイズのパームトップ・システムから，数千個のチップからなる超並列スーパーコンピュータまで, 様々なシステムでの実現が可能です。このアーキテクチャはまた， OpenVMS Alpha も含めて，複数のオペレーティング・システムをサポートします。

表 1–1 は， Alpha アーキテクチャと VAX アーキテクチャの主な違いを示しています。

表 1–1 Alpha アーキテクチャと VAX アーキテクチャの比較

Alpha	VAX
• 64 ビット・アドレス†	• 32 ビット・アドレス
• 64 ビット処理	• 32 ビット処理
• 命令	• 命令
– 単純	– やや複雑
– すべて同じ長さ (32 ビット)	– 可変長
• ロード/ストア・メモリ・アクセス	• 操作とメモリ・アクセスを 1 つの命令に組み合わせることが可能
• アラインされていないデータに対しては重大な性能低下	• アラインされていないデータに対して中程度の性能低下
• 多くのレジスタ	• 比較的数の少ないレジスタ
• 命令は要求の順序と無関係に終了	• 命令は要求された順に終了
• 多段のパイプラインと分岐予測	• パイプラインは限定的に使用
• 大きいページ・サイズ (ハードウェアに応じて 8KB ～ 64KB)	• 小さいページ・サイズ (512 バイト)

†64 ビット・アドレスについては，『OpenVMS Alpha 64 ビット・アドレッシングおよび VLM 機能説明書』を参照してください。

RISC の一般的な特性

Alpha アーキテクチャの特徴の一部は，新しい RISC アーキテクチャの典型的な特徴です。次の特徴は特に重要です。

- 単純な命令セット

  Alpha アーキテクチャではかなり単純な命令を使用しており，これらはすべて 32 ビットの長さです。これらの共通の命令は 1 クロック・サイクルだけを必要とします。このようにサイズが統一された単純な命令の採用により，**複数命令発行** (mullti-instruction issue)や最適化された命令スケジューリングなどの方式を採用でき，その結果，きわめて高い性能を実現できます。

- 複数命令発行

  初期の Alpha プラットフォームは，1 クロック・サイクルで 2 つの命令を出しました。現在のシステム (EV5 以上) では，1 クロック・サイクルに 4 つの命令を出します。

- ロード/ストア操作モデル

  Alpha アーキテクチャでは，32 個の 64 ビット整数レジスタと，32 個の 64 ビット浮動小数点レジスタを定義しています。大部分のデータ操作は，レジスタ間で実行されます。操作の前に，オペランドがメモリからレジスタにロードされます。操作が終了した後，結果はレジスタからメモリにストア (格納) されます。

  このように操作をレジスタ・オペランドに制限すれば，単純で統一された命令セットを使用できます。さらに，メモリ・アクセスを算術演算から分離することにより，完全なパイプライン，命令スケジューリング，および並列操作ユニットを実現できるシステムとして，大幅に性能を向上できます。

- マイクロコードの排除

  Alpha アーキテクチャではマイクロコードを使用しないため，Alpha プロセッサはマシン命令を実行するために，ランダム・アクセス・メモリ (RAM) からマイクロコード命令をフェッチするのに必要な時間を削減できます。

- 命令のランダムな終了

  Alpha アーキテクチャでは，命令が発行された順番に完了することは保証されません。その結果として，算術演算例外や浮動小数点例外は，最適化された命令列を乱さないように，少し時間をおいて報告されます。

Alpha 固有の特性

これまで説明した RISC の一般的な特性の他に，Alpha アーキテクチャは，移行した VAX アプリケーションを Alpha システムで実行するのに使用される多くの機能を備えています。Alpha アーキテクチャでは次の機能が提供されます。

- H 浮動小数点と D 浮動小数点を除き，他のすべての VAX データ型に対するハードウェア・サポート (アプリケーションで H 浮動小数点，または D 浮動小数点データを使用しているときに，どのような処理を実行しなければならないかについては，第 2.5.2 項を参照してください)。
- 4 種類のプロセッサ・モード (ユーザ，スーパーバイザ，エグゼクティブ，およびカーネル) などの特定の特権付きアーキテクチャ機能，32 段階の割り込み優先順位レベル (IPL, Interrupt Priority Level)，および非同期システム・トラップ (AST)。
- 特権付きアーキテクチャ・ライブラリ (PAL) は PALcode と呼ばれる環境の一部であり，チェンジ・モード (CHM*x*)，プローブ (PROBE*x*)，キュー命令，REI など，特定の VAX 命令に対応する不可分な実行をサポートします。

  Alpha アーキテクチャは特定のオペレーティング・システムにだけ対応するわけではありません。異なるオペレーティング・システムに対応できるように，特権アーキテクチャ・ライブラリ・コード (PALcode) を作成できます。

さらに，C や MACRO–32 コンパイラなどのように，特定の OpenVMS Alpha コンパイラは PALcode 組み込み関数を提供しており，Alpha 命令セットで提供される命令を補足します。たとえば，MACRO–32 コンパイラは，Alpha に対応する命令のない VAX 命令をエミュレートする組み込み関数を提供します。

PALcode を使用すると，内部ハードウェア・レジスタと物理メモリにアクセスします。PALcode は物理メモリと仮想メモリの直接的な対応関係を提供できます。PALcode の詳細については，『Alpha Architecture Reference Manual』を参照してください。

1.2.1 ユーザ作成デバイス・ドライバ

ユーザ作成デバイス・ドライバと Step 2 ドライバ・インタフェースと呼ぶ新しいインタフェースの正式なサポートは，OpenVMS AXP バージョン 6.1 から開始されました。Step 2 ドライバ・インタフェースは C プログラミング言語 (および MACRO と BLISS) でユーザ作成デバイス・ドライバをサポートします。これは OpenVMS Alpha バージョン 1.0 とバージョン 1.5 で提供されていた一時的な Step 1 ドライバ・インタフェースに代わるものです。既存の OpenVMS VAX デバイス・ドライバを Alpha システム上で実行したい場合で，OpenVMS Alpha バージョン 6.1 に必要な変更を行っていない場合は，『Creating an OpenVMS Alpha Device Driver from an OpenVMS VAX Device Driver』を参照してください。

C で OpenVMS VAX デバイス・ドライバを作成する機能は，正式にはサポートされません。たとえば，OpenVMS VAX では内部の OpenVMS (lib) データ構造体に対して.h ファイルを提供しません。

Step 2 ドライバ・インタフェースの導入により，OpenVMS Alpha と OpenVMS VAX デバイス・ドライバの相違点が大きくなりました。VAX MACRO または BLISS で作成されたデバイス・ドライバ・ソース・ファイルは，条件付きコンパイルとユーザ作成マクロを使用することにより，OpenVMS Alpha と OpenVMS VAX の間で共通に使用できます。

このアプローチが適切であるかどうかは，個々のドライバの性質に大きく左右されます。(OpenVMS バージョン 7.0 では，64 ビットのサポートにより，相違点がさらに大きくなります。) OpenVMS Alpha の将来のバージョンでは，入出力 (I/O) サブシステムはデバイス・ドライバに影響を与えるような方向にさらに進歩していく可能性があります。この結果，OpenVMS Alpha と OpenVMS VAX のデバイス・ドライバの相違点が大きくなり，共通のドライバ・ソースを維持するのはますます複雑になります。この理由から，長期的に見ると，完全に共通なドライバ・ソース・ファイルを使用するアプローチは適切でないと考えられます。

個々のドライバに応じて，ドライバを共通モジュールとアーキテクチャ固有のモジュールに分割する方法が適している可能性があります。たとえば，ディスク圧縮を行うデバイス・ドライバを作成する場合，圧縮アルゴリズムはアーキテクチャから独立したモジュールに簡単に分離できます。異なる種類のオペレーティング・システム

間でいくつかの共通モジュールを作成し，オペレーティング・システム固有のデータ構造体をこのような共通のモジュールから分離することができます。たとえば，OpenVMS， Windows NT， Digital UNIX の間でこのようなアプローチを採用できます。

新しい OpenVMS Alpha のデバイス・ドライバを作成する方法の詳細については，『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 Alpha システムで実行するために変換するプロセスは，次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 Alpha 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 Alpha 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。

1.4 移行の手段

Alpha 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**

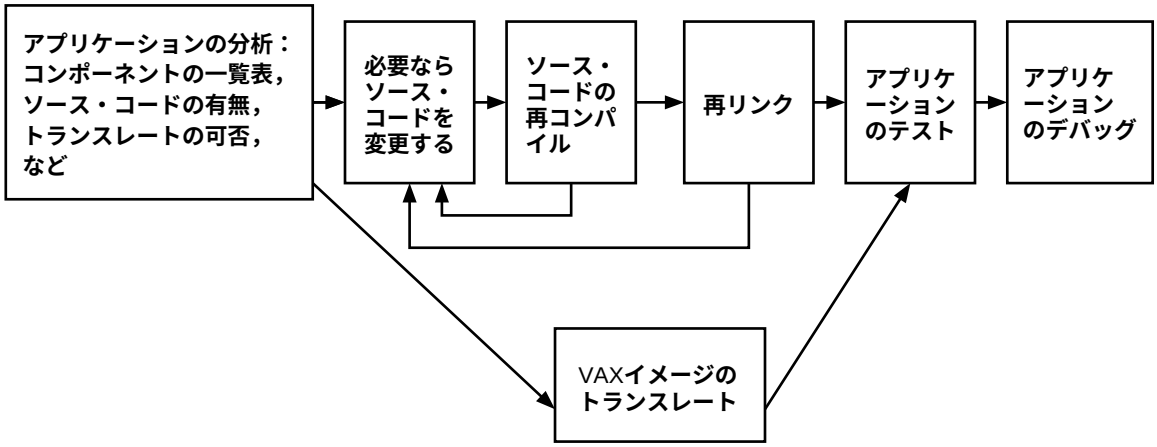
  この方法ではネイティブな Alpha イメージが作成されます。

- **トランスレーション**

  この方法でも Alpha イメージが作成されますが，一部のルーチンは TIE のもとでエミュレートされます。

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。

図 1–1 VAX アプリケーションを Alpha システムに移行する方法



JRD-4988A

再コンパイルと再リンク

プログラムを OpenVMS VAX から OpenVMS Alpha に変換するための，もっとも効果的な方法は， Alpha コンパイラ (DEC C や DEC 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 リンカで適切なスイッチとオプション・ファイルを使用して，作成されたオブジェクト・ファイルと必要な他の共有可能なイメージを再リンクする方法です。この方法では， Alpha システムのスピードを完全に活用できる，ネイティブな Alpha イメージが作成されます。

トランスレーション

VAX システムと Alpha システムにはいくつかの相違点がありますが， VAX Environment Software Translator (VEST) を使用すれば，大部分のユーザ・モードの VAX イメージを， Alpha システムでエラーなしに実行することができます。VEST は DECmigrate for OpenVMS Alpha の一部です。例外については，第 2.3 節を参照してください。

この方法では，ソースを再コンパイルする場合より高いレベルで VAX との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に，代わりとなる安全な手段として使用してください。たとえば，次の状況ではトランスレーションが適しています。

- OpenVMS Alpha 用のコンパイラ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

VEST は VAX バイナリ・イメージ・ファイルをネイティブな Alpha イメージにトランスレートします。このイメージは，Alpha システム上の Translated Image Environment (TIE) のもとで実行されます (TIE は，OpenVMS Alpha でバンドル・ソフトウェアとして提供される共有可能イメージです)。トランスレートされた VAX イメージは，エミュレータやインタプリタのもとで実行されるわけではなく (ただし，一部の例外があります)，元の VAX イメージで命令が実行する操作と同じ操作を実行する Alpha 命令が，新しい Alpha イメージに盛り込まれるのです。

トランスレートされたイメージを Alpha システムで実行する速度は，元のイメージを VAX システムで実行する速度と同じくらいになります。しかし，トランスレートされたイメージは，Alpha アーキテクチャを完全に活用するための最適化コンパイラの恩恵を受けないため，通常，ネイティブな Alpha イメージの速度と比較すると，約 25 ～ 40%の速度でしか動作しません。このように性能が低下する主な理由は，データがアラインされていないためと，複雑な VAX 命令が広範囲にわたって使用されているためです。

イメージのトランスレーションと VEST についての詳しい説明は，第 3.2.2.1 項と『DECmigrate for OpenVMS AXP Systems Translating Images』を参照してください。

ネイティブな Alpha イメージとトランスレートされたイメージの混在

1 つのアプリケーションを構成するイメージで，2 種類の移行方式を混在させて使用できます。また，1 つのアプリケーションを，移行の 1 段階として部分的にトランスレートすることもできます。このようにすれば，完全に再コンパイルする前に，アプリケーションを Alpha ハードウェアで実行し，テストできます。1 つのアプリケーション内のネイティブな Alpha イメージと，トランスレートされた VAX イメージの相互操作性についての詳しい説明は，第 2.7.2 項を参照してください。

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。

1. 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
2. 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
3. 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入手できます。

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 1 段階は，何を移行するかを正確に判断することです。この段階では，アプリケーション自体を評価するだけでなく，アプリケーションを正しく実行するために，何が必要であるかも判断しなければなりません。アプリケーションの評価を開始するには，次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
  - 共有可能イメージ
  - 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，共有可能イメージ，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

  - サード・パーティ・プロダクト

    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 は，アプリケーションが依存している実行可能イメージや共有可能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。

- 必要な操作環境
  - 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，必要なディスク空間などです。

  - ビルド・プロシージャ

    このプロシージャは，Code Management System (CMS) や Module Management System (MMS) などの Digital tool を必要とします。

  - テスト・スーツ

    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ために，テストが必要です。

これらの多くは，すでに OpenVMS Alpha に移行されています。たとえば，次のソフトウェアやライブラリはすで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)
  - 他の共有可能ライブラリ (shareable library)。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
- サード・パーティ・プロダクト

  現在多くのサード・パーティ・アプリケーションが，OpenVMS Alpha で実行可能です。各アプリケーションが移行されているかどうかかについては，各アプリケーションの提供業者にお問い合わせください。

ビルド・プロシージャとテストも含めて，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 2.4 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と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

この章では，アプリケーションを移行するにあたって，可能な限り再コンパイル，再リンクを行い，イメージのトランスレーションについては，再コンパイル，再リンクできない部分に用いるか，移行の過程での一時的な処理のみに用いると仮定しています。

この後の節では，移行方法を選択するプロセスについて，その概要を説明します。このプロセスでは，次のステップを踏んでください。

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か， VAX イメージをトランスレートすることができます。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は，第 2.3 節を参照してください。

2. 再コンパイルに影響を与える可能性のあ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， Alpha 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 2.4 節では，これらの依存性について説明し，このような固有の機能に依存する部分を識別し，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 2.6 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，どの移行方法を使用するのかを決定しなければなりません。第 2.7 節では，各方法の利点と欠点のバランスをとることにより，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。

プログラムを再コンパイルおよび再リンクできない場合や，VAX イメージが 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しなければなりません。第 2.7.1 項では，トランスレートされたイメージの互換性と性能を向上するための方法を説明しています。

アプリケーションの評価のプロセスは，図 2–1 のように表わすことができます。弊社では，この図に示されたアプリケーションの評価に役立つツールを提供しています。これらのツールは，移行プロセスの各段階の説明で，必要に応じてふれます。

2.3 どの移行方法が可能か?

ほとんどのアプリケーションは，ソース・コードの再コンパイルおよび再リンクと，イメージのトランスレートのどちらの方法を用いても移行が可能です。しかし，アプリケーションの設計に応じて，次に示すように，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

  次のイメージはトランスレートしなければなりません。

  – OpenVMS Alpha 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VAX SCANや Dibol のようなソフトウェア

  – ソース・コードを入手できない実行可能イメージと共有可能イメージ

  – 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

  次のイメージのソース・コードは再コンパイルおよび再リンクしなければなりません (変更も必要となる可能性があります)。

  – 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 5.5-2 以降に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・サービスがそれ以来更新されていない場合

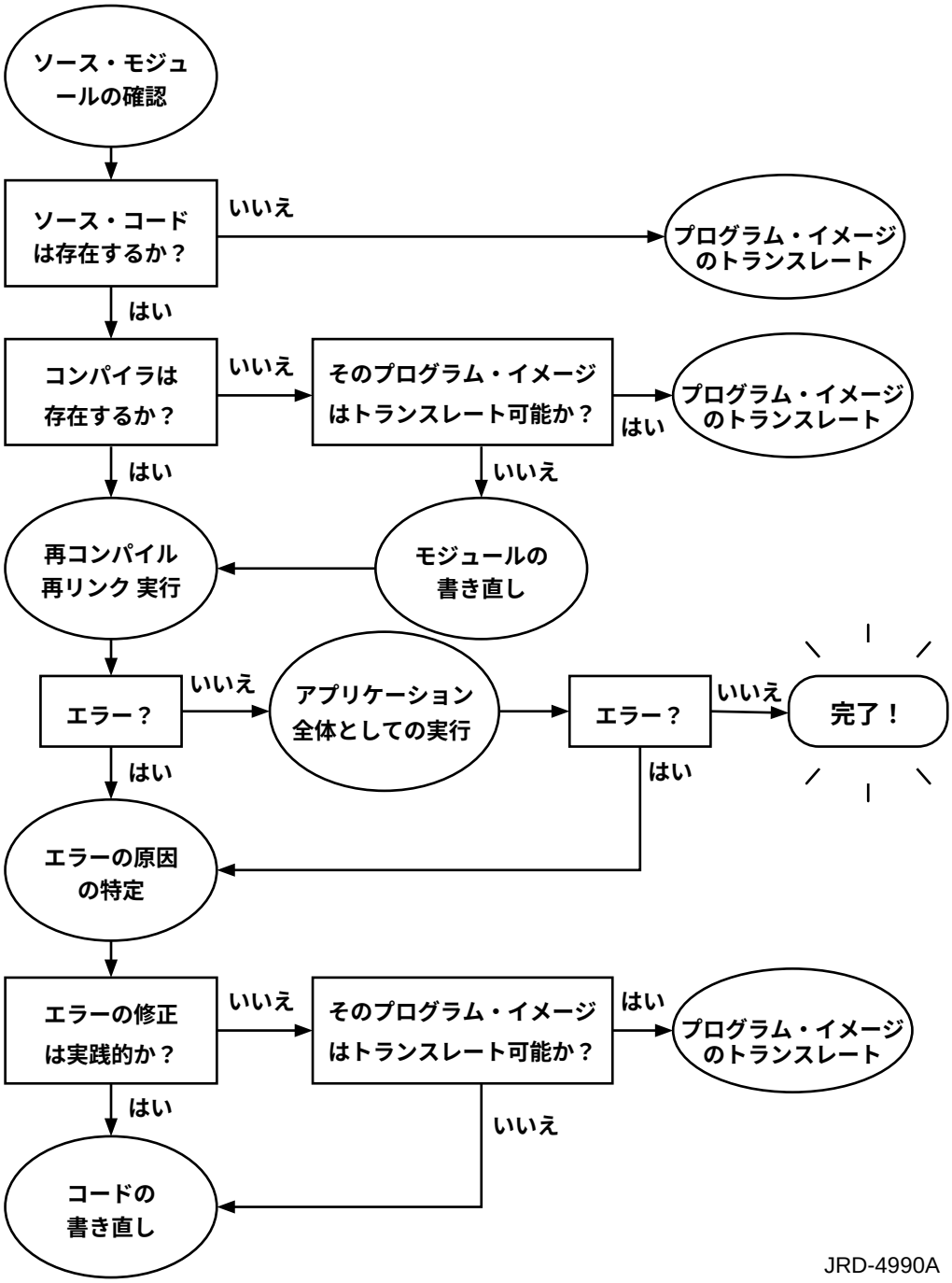
  – Ada で書かれたイメージ

  – Ada で書かれたイメージによって呼び出される/呼び出されたイメージ

  – PDP–11互換モードを使用するイメージ

  – リンカ・オプションでベースを指定したイメージ

図 2–1 プログラム・イメージの移行



JRD-4990A

– 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，次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

- 解説書に説明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
- リターン・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) をもとに，他の判断を下すコード
- 512 バイトのサイズのメモリ・ページに依存するため，正確なアクセス違反動作に依存するコード
- 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)
- 大部分の VAX P0 空間または P1 空間を使用するコード，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 は次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しい説明は，『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 (portability) 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 Alpha 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 Alpha 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
- 特権付きコード
- 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まった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。

2.4.1 VAX MACROアセンブリ言語

Alpha システムでは，VAX MACROはアセンブリ言語ではなく，コンパイラの 1 つでしかありません。しかし，高級言語のための Alpha コンパイラと異なり， VAX MACRO–32 Compiler for OpenVMS Alpha 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 Alpha 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， Alpha システムでは解消されました。

- RISC 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。 Alpha コンパイラ・セットに含まれているコンパイラなどの RISC コンパイラは，プログラマが手作業で最適化するより効率よく，もっと容易にアーキテクチャやインプリメントの機能を利用して，最適化されたコードを生成できます。
- 新しいシステム・サービスは，これまでアセンブリ言語を必要としていた一部の機能を実行できます。

MACRO コードの移行についての詳しい説明は，『Porting VAX MACRO Code to OpenVMS Alpha』を参照してください。

2.4.2 特権付きコード

内部アクセス・モード (カーネル，エグゼクティブ，またはスーパーバイザ・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 Alpha には存在しない VAX データ呼び出し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 Alpha 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可能イメージ

  詳しくは，『OpenVMS Programming Concepts Manual』と『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・ドライバとパフォーマンス・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 オプションを選択して$CRMPSC システム・サービスを使用するコード

  メモリ・マッピングの詳細については，第 5 章をご覧ください。

- 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。

2.4.3 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， Alpha 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， Alpha 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を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。

2.5 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

ネイティブな Alpha 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 Alpha オペレーティング・システムは OpenVMS VAX と高度な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 Alpha 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の場合は，問題が発生する可能性があります。以下の節では，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。

2.5.1 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に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に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をアクセスすると， VAX システムでも Alpha システムでも性能が大幅に低下します。 VAX システムでは，大部分の言語は省略時の設定により，データを次の使用可能なバイト境界にアラインするため，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ます。

しかし， Alpha システムでは，各データを自然なアラインメントにすることが省略時の設定であるため， Alpha は他の典型的な RISC アーキテクチャと同様に，アラインされていないデータを使用することによって発生する性能の低下を最低限に抑えるためのハードウェア・サポートを準備していません。この結果， Alpha 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 〜 100 倍も速くなります。

Alpha 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には，次の方法が有効です。

- 大部分の Alpha コンパイラが提供するスイッチを使用する方法。このスイッチを使用すれば，コンパイラはアラインされていないデータのコンパイル時参照を報告できます。たとえば， Digital 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。
- 実行時にアラインされていないデータを検出するために， 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，次に示す方法を用います。

- データをクォドワード境界にアラインすることにより，性能を最大限に向上する方法。これは， Alpha システムが一般にクォドワード**粒度**のみをサポートするからです (第 2.5.4 項を参照)。
- 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する方法。データが確実にアラインされるように間隔をあけると，メモリ・サイズが拡大す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最初に宣言する方法があります。

- データ構造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する方法。たとえばDEC C では，自然なアラインメントが省略時のオプションです。VAX Cの省略時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。 DEC Fortran の場合には，省略時の設定により，ローカル変数は自然なアラインメントになります。レコード構造とコモン・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#PRAGMA NO_MEMBER_ALIGNMENT のようなコンパイラ・スイッチを使用する方法。これらのスイッチを使用すると，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Alpha 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・フォーマットは，アラインされていないデータを含むレコードを指定す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を渡されることを要求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しい説明は，第 7 章と第 8.4.2 項を参照してください。

2.5.2 データ型

Alpha 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 データ型はサポートされていません (表 2–1 を参照)。アプリケーションが，下位のネイティブな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 2–1 浮動小数点データ型のサポート

データ型	VAX	Alpha
D53 浮動小数点 (G 浮動小数点) (省略時の倍精度形式)	サポートされない。	サポートされる。 D53 浮動小数点の代わりに D56 浮動小数点を使用すると，3 ビット分の精度が失われ，結果に多少の誤差が生じる。
D56 浮動小数点 (省略時の倍精度形式)	サポートされる。	サポートされない。 DECmigrate を使用してコードをトランスレートすると，このデータ型を完全にサポートできるようになる。また，アプリケーションで 3 ビット分の精度が必要ない場合は，D56 浮動小数点の代わりに D53 浮動小数点を使用できる。
F 浮動小数点	サポートされる。	サポートされる。
G 浮動小数点	サポートされる。	サポートされる。
H 浮動小数点 (128 ビットの浮動小数点)	サポートされる。	サポートされない。 DECmigrate を使用すると，H 浮動小数点を完全にサポートできるようになる。 DECmigrate を使用すると，H 浮動小数点構造体を含むコード・モジュールをトランスレートすることができ，サポートされるデータ型を使用して，アプリケーションを再コーディングすることもできる。
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 (Alpha; IEEE に適した))	サポートされない。	DEC Fortran バージョン 6.2 以降と DEC C バージョン 4.0 以降でサポートされる。 X 浮動小数点データ形式は H 浮動小数点と同じではないが，どちらも同じ範囲の値をサポートする。 Fortran アプリケーションの場合は， FOR$CONVERTnnn 論理名， OPTIONS 文，/CONVERT コンパイラ修飾子， OPEN 文の CONVERT=キーワードのいずれかを使用して，X 浮動小数点メモリ形式と H 浮動小数点ディスク・データの間で自動的な変換を行うことができる。

Alpha プロセッサ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，および完全な精度の D 浮動小数点データ型をソフトウェアによって実現します。

- 10 進数

  18 桁のパック 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

  Alpha 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

  Alpha プラットフォームでは， D 浮動小数点データは次の方法で実現されます。

  – G 浮動小数点ハードウェア (D53) を使用する方法。 Alpha ハードウェアは D 浮動小数点データ (D53) を処理のために G 浮動小数点に変換します。この結果， D 浮動小数点データを格納した既存のバイナリ・ファイルとの間で，速度およびデータ型の互換性を維持できますが，現在の VAX システムの D 浮動小数点算術演算と比較すると， 3 ビットが失われてしまいます。つまり， VAX システムの D56 では 16 桁の精度で処理されるのに対し， D 浮動小数点データは 15 桁の精度で処理されます。

  – トランスレートされたイメージのためにソフトウェア・エミュレーション (D56) を使用する方法。この方法では，正確な D56 フォーマットの結果が得られますが， D53 や G 浮動小数点より処理速度が遅くなります。

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，次の方法を用います。

- 可能な場合には， H 浮動小数点のかわりに G 浮動小数点または IEEE T 浮動小数点を使用してください。これは次の理由によります。
  – どちらのデータ型も10^{-308}～10^{308}の範囲のデータをサポートするため
  – 約 15 桁の精度であるため
- 可能な場合には，パック 10 進数データ型のかわりに整数データ型を使用してください。

Alpha のデータ型についての詳しい説明は，第 7 章を参照してください。

2.5.3 データへの共有アクセス

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確認できない
- 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 Alpha では，データをメモリから読み込む操作，メモリ内の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，複数の命令に分割され，これらの命令の間で割り込みをかけることができます。この結果，アプリケーションで共有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。

次の条件が満足される場合，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有すること

- 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**のデータを共有すること
- 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有すること

問題の検出

不可分性への依存を検出するには，共有変数 (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なければなりません。

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，次の方法を用います。

- 可能な場合には，共有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) またはライブラリ (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 を再度有効に設定します。

同期についての詳しい説明は， 第 6 章を参照してください。

2.5.4 クォドワードより小さいデータの読み込みまたは書き込み

粒度という用語は，隣接するメモリ位置に格納されているデータを妨害せずに，不可分な操作として，メモリとの間で読み込みまたは書き込みを実行できるデータ・サイズを示します。 VAX のようにバイト・レベルの粒度を提供するマシンを**バイト粒度マシン**と呼びます。 Alpha システムは**クォドワード粒度**のシステムです。[1]

OpenVMS Alpha はクォドワード粒度のシステムであるため，共有されるバイト，ワード，またはロングワードに書き込むと，共有データと同じクォドワードに格納されている他のデータを破壊する可能性があります。このような状況は次の場合に発生します。

- プログラムでバイト，ワード，またはロングワードを変更する場合
- 任意のサイズのアラインされていないフィールドが，アラインされたクォドワード境界と交差し，個別に書き込まなければならないバイト，ワード，またはロングワードを作成する場合

[1] Alpha アーキテクチャはロングワード粒度もサポートしますが，ロングワード粒度を仮定することは望ましくありません。省略時の設定で，コンパイラはソース・コードがクォドワード未満の粒度に依存しないものと仮定していますが，多くの Digital 言語は/GRANULARITY 修飾子を使用して小さな粒度を指定することができます。

注意

不可分性に関する説明 (第 2.5.3 項) で説明したデータ共有の種類はすべて，共有データを格納しているクォドワードの残りの部分で，粒度に関する問題を発生させる可能性があります。

さらに，結果をプロセス空間に戻すような，非同期的に終了するライブラリ関数，または非同期システム・サービスをプロセスが起動した場合には，戻されたデータを格納するクォドワードで，粒度に関する問題が発生する可能性があります。たとえば，次の操作では，粒度に関する問題が発生する可能性があります。

- **状態ブロックに書き込む非同期システム・サービス**
- **プロセス・バッファに書き込む入出力操作**
- **ダイレクト・メモリ・アクセス** (DMA)**・コントローラがプロセス・バッファに書き込みを実行する入出力操作**

問題の検出

バイト粒度，ワード粒度，またはロングワード粒度の使用を検出するには，次の方法を用います。

- 意識的に共有されるデータ (AST とメイン・スレッドの間またはプロセス間で) を検出します。共有データが，書き込まれる可能性のある他のデータと同じクォドワードを使用するかどうかを確認します。
- 非同期システム・サービス，または非同期的に終了するライブラリ呼び出しから戻されたデータを確認します。そのデータが，他のプロセスによって書き込まれた他のデータと同じクォドワードを使用するかどうかを確認します。
- 非同期的に戻されたデータを格納する装置からデータを受信する入出力バッファを調べます。バッファの先頭と末尾が，他のプロセスによって書き込まれたデータと同じクォドワードを使用するかどうかを確認します。

問題の対処方法

クォドワード未満の粒度の使用に対処するには，次の方法を用います。

- 共有データを固有のクォドワードに格納します。
- 入出力バッファの先頭をクォドワード境界にそろえ，バッファの後に続くデータを次のクォドワードに移動します。
- 問題の原因がシステムと共有されるデータでない場合には，高いレベルの同期メカニズムを使用して，意識的に共有されるデータとバックグラウンド・データの両方を同じクォドワードでインターロックします。

Digital コンパイラは，省略時の設定では粒度がクォドワードであるものと解釈しますが，現在のコードと互換性を維持するために，/GRANULARITY 修飾子を使用することにより，バイト，ワード，アラインされないロングワード，およびアラインされないクォドワードの粒度を指定することができます。詳細については，各コンパイラのマニュアルを参照してください。

読み込み/書き込み粒度の詳細については，第 6 章を参照してください。

2.5.5 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，混在環境の 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礎にもなります。

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。しかし，全体的なシステム性能を向上するために， Alpha システムでは，各ハードウェア・プラットフォームに応じて， 8KB ～ 64KB の大きなページ・サイズを採用しています。

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 バイトの各メモリ領域ごとに設定できます。 Alpha 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・サイズを大きく変更した結果，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，次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**
  - **メモリの使用状況を計算するため**
  - **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 バイトひとかたまりで仮想メモリを操作するコードを識別するか，または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識別します。コードから 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 Alpha 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，次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リック値に変更します (実行時に$GETSYI を呼び出すことにより割り当てられます)。
- ページ・サイズとディスク (ファイル)・ブロック・サイズが等しいと仮定しているコードを再評価します。 Alpha システムでは，この仮定は正しくありません。
- メモリ管理関連システム・サービス (たとえば，$CRMPSC，$MGBLSC) を使用して，ファイルを固定のページ・サイズ依存アドレス空間 (グローバル・セクション) にマッピングできるものと仮定しないでください。このような場合には，$EXPREG システム・サービスを使用してください。

ページ・サイズについての詳しい説明は， 第 5 章を参照してください。

2.5.6 マルチプロセッサ・システムでの読み込み/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，書き込み操作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。

全体のシステム性能を向上するために， Alpha システムをはじめ， RISC システムでは，メモリへの読み込み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結果をシステム内の他の CPU から確認すると，その順序は実際に書き込まれ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，読み込み/書き込みの順序に依存している部分を検出するには，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

読み込み/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かわりに，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。
- Alpha アーキテクチャでは，メモリ・バリア命令が準備されており，この命令を使用すると，ハードウェアはバリアの前のすべてのメモリ読み込みと書き込みを終了した後で，バリアの後の読み込みと書き込みを実行します。 OpenVMS Alpha の一部の言語では，この命令を挿入する方法が準備されていますが，この命令を使用すると性能が低下します。

同期についての詳しい説明は，第 6 章を参照してください。

2.5.7 算術演算例外の報告の即時性

Alpha (およびベクタ VAX) システムでは，スカラ VAX システムと異なる方法で例外を取り扱います。スカラ VAX システムでは，「正確な例外報告」を使用します。つまり，命令を実行した結果，例外が発生した場合には，報告される**プログラム・カウンタ** (PC)は，その例外の原因となった命令のアドレスです。後続の命令は実行されておらず，プロセスのコンテキストに影響を与えないため，条件ハンドラは例外の原因を修正し，障害が発生した命令から，またはその次の命令からプログラムの実行を再開できます。

パイプライン環境で可能な最高の性能を実現するために，ベクタ VAX システムと Alpha システムでは「あいまいな例外報告」を採用しています。つまり，例外ハンドラが報告する PC は，算術演算例外の原因となった命令の PC であるとは限りません。さらに，例外が報告される前に後続の命令が終了している可能性があります。

実際には，算術演算の例外となった特定の命令を知らなければならないプログラムはほとんどありません。通常，算術演算例外が発生した場合には，プログラムは次のいずれかの操作を実行します。

- 例外を記録し，処理を継続します。
- エラー・メッセージを印刷し，サブルーチンまたはプログラムを強制終了します。
- サブルーチン全体を再起動し，オーバーフローまたはアンダーフローを防止するために，異なるアルゴリズムを使用してデータを再計算します。

VAX プログラムが算術演算例外を検出したときに，これらのいずれかの動作を実行する場合には，そのプログラムは，あいまいな例外報告を採用している RISC システムに移行しても，まったく影響を受けません。

注意

あいまいな例外報告は算術演算例外にだけ適用されます。フォルトやトラップなどの他の例外の場合には，OpenVMS Alpha は正確に例外を報告し，例外の原因となった特定の命令を報告します。

例外処理についての詳しい説明は，第 8.4.1 項を参照してください。

2.5.8 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 Alpha 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， Alpha 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，次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位置を判断するコード

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

- スタックのリターン・アドレスを変更するコード
- コール・フレームの内容を解釈するコード

VAX コンパイラも Alpha コンパイラも，プロシージャ引数をアクセスするための方法を準備しています。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， Alpha 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。ここに示したコードを含むイメージは，トランスレートすることにより，Alpha システムで正しく実行できるようになります。

2.5.9 VAX データ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 Alpha システムとで異なります。算術演算例外が， VAX システムと Alpha 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。この節では主に，コードが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をアクセスするメカニズムについて説明します。

2.5.9.1 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できる方法が数多く準備されています。 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コール・フレームの先頭に書き込むことによって，そのプロシージャ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わけですが， Alpha プロシージャのプロシージャ・ディスクリプタには，これに対応する場所は確保されていません。

たとえば，VAX MACROプログラムは次の一連の命令を使用して，条件ハンドラのアドレスをコール・フレームに移動できます。

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

VAX MACRO–32 Compiler for OpenVMS Alpha 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 Alpha アセンブリ言語コードを作成します。詳しくは『Porting VAX MACRO Code to OpenVMS Alpha』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，** Alpha **システムには存在しません。しかし，コンパイラ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 **の組み込み機能によって)。詳しくは第** 11 **章とアプリケーションに関連するコンパイラの解説書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。

特定の Alpha コンパイラ (およびクロス・コンパイラ) は，動的な条件ハンドラを設定するためのメカニズムを準備しています。

条件ハンドラについての詳しい説明は，第 8 章を参照してください。

2.5.9.2 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 Alpha 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・アレイの内容およびフィールドの長さは， VAX システムと Alpha システムとで異なります。シグナル・アレイ，またはメカニズム・アレイ内のデータを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説明は，『OpenVMS Programming Concepts Manual』を参照してください。

注意

条件ハンドラは，ハードコードされた引数ポインタ (AP) からの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検索することができません。

2.5.10 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で作成された 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，この情報をアクセスします。 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。

Alpha 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 Alpha 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。

2.5.11 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， Alpha 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 Alpha 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・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・ステータス・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しい説明は，『Porting VAX MACRO Code to OpenVMS Alpha』を参照してください。

2.5.12 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 Alpha 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しい説明は，『Porting VAX MACRO Code to OpenVMS Alpha』を参照してください。

2.6 VAX システムと Alpha 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 Alpha 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。このときに役立つコンパイラ・スイッチについての説明は，各コンパイラの解説書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。

DEC コンパイラは，移植に関して発生する可能性のある問題を識別す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 修飾子に 10 個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，コンパイラは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を報告します。

DEC 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組み合わせて実行したときに，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，次の方法を使用して， Alpha 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 2.4 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実際に不可能な場合には，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，アーキテクチャ上の問題点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出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 Alpha コンパイラの違い

  – データ・アライメント

  特定のコンパイラは， VAX アーキテクチャと Alpha アーキテクチャのその他の違いも検出します。

- VEST の使用によるイメージの分析

  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， Alpha 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操作に関して，役立つ多くの情報が得られます。たとえば， VEST は次の問題を検出するのに役立ちます。

  – 静的にアラインされていないデータ (データ構造内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

  – 特権付きコード

  – 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 (動的に作成されたデータ構造内の変数)

  – 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 Alpha 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しい説明は，第 3.3.3 項を参照してください。

2.7 再コンパイルするか，または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， Alpha 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を予測し，イメージのトランスレートに必要な作業とネイティブな Alpha イメージに変換するのに必要な作業を判断し，性能と作業量のバランスを考えなければなりません。

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，ネイティブな Alpha イメージからトランスレートされた共有可能イメージ (translated shareable image) 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しい説明は，第 2.7.2 項を参照してください。

表 2–2 では， 2 種類の移行方法を比較しています。

表 2–2 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 Alpha の機能	通常，ネイティブ Alpha の 25 〜 40%の性能
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 以前のソースのコンパイラ	OpenVMS VAX バージョン 4.0 またはそれ以降のイメージのみがサポートされる

(次ページに続く)

表 2–2 (続き) 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
– 制約事項	特権付き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 のソース保守は，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，次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?

  Alpha システムの速度を有効に使用したいイメージは，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， Performance and Coverage Analyzer(PCA) を使用します。

  – Alpha コンパイラが作成するイメージだけが Alpha 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 Alpha 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必要な作業量はどの程度ですか?

  – アプリケーションの複雑さによりますが，通常ソフトウェア・トランスレーションのほうが，再コンパイルおよび再リンクよりも少ない作業量と作業時間で済みます。

    ネイティブ・バージョンに移行する場合， OpenVMS Alpha で実行するために，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ができます。

– VAX アーキテクチャ固有の機能や， VAX 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，または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。

アーキテクチャ間で互換性が維持されていない部分については，次の方法で対処できます。

- 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・アーキテクチャにとって適切な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・コードの変更箇所をできるだけ少なくし，正しくプログラムが動作するように，高級言語のコンパイラ・スイッチを使用します。

表 2–3 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 Alpha システムに移行するための 2 つの方法に，どのような影響を与えるかを示しています。詳しくは，次の章以降を参照してください。

表 2–3 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]	
省略時の設定では，大部分のコンパイラはデータを自然なアラインメントにする。VAX アラインメントを保存するための修飾子についての説明は，第 11 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。
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 フォーマットの 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 16 桁の精度 (D56 フォーマット) 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 進数は操作のために自動的にバイナリ形式に変換される。 詳しくは第 7 章を参照。	D 浮動小数点の 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，省略時の設定である/FLOAT=D53_FLOAT を使用した場合より低下する。

[1]アラインされていないデータは主に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 システムでは性能をある程度低下させるだけですが， Alpha 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 100 倍も時間がかかる可能性があります。

(次ページに続く)

表 2–3 (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
読み込み/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準備しているオプションに応じて異なる (詳しくは第 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。
バイト書き込み操作とワード書き込み操作の不可分性と粒度	
適切にソース・コードを変更し，コンパイラ・オプションを使用することによりサポートされる (詳しくは第 6 章を参照)。	/PRESERVE=MEMORY_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	
OpenVMS リンカは省略時の設定により，大きい Alpha スタイルのページを作成する。	512 バイトのページ・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，アクセス違反を検出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， Alpha システムで正しく実行されない。
読み込み/書き込みの順序	
適切な同期命令 (MB) 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。 (詳しくは第 6 章を参照。)	/PRESERVE=READ_WRITE_ORDERING 修飾子を使用する。
例外報告の即時性	
コンパイラ・オプションの使用によって部分的にサポートされる (詳しくは第 8 章を参照)。	/PRESERVE=FLOAT_EXCEPTIONS 修飾子または/PRESERVE=INTEGER_EXCEPTIONS 修飾子を使用する。実行速度は半分に低下する。
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 アーキテクテ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。

2.7.1 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のトランスレートもでき，移行プロセスの一段階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ができます。

再コンパイルに影響を与える多くの相違点について第 2.4 節で説明しましたが，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。次の方法を使用すれば， VAX アーキテクチャに依存するイメージの互換性を向上できます (詳しくは『DECmigrate for OpenVMS AXP Systems Translating Images』を参照してください)。

- データ・アラインメント

  VEST のトランスレート時修飾子/OPTIMIZE=NOALIGNMENT を使用すれば， VEST は特別なインラインの Alpha コードを生成し，アラインされていないデータの参照に対する例外を生成しないようにします。この修飾子を使用した場合， VEST は，アラインされたデータ参照だけを使用するコードよりも実行速度が約 10 倍も遅い Alpha コードを作成します (省略時のオプションである/OPTIMIZE=ALIGNMENT を使用した場合には，アラインされていないデータは例外を生成しますが，この結果，アラインされたデータを実行する場合より約 100 倍の時間がかかります)。

- 命令の不可分性

  トランスレータを起動するときに，トランスレート時修飾子 /PRESERVE=INSTRUCTION_ATOMICITY を指定すれば， VEST は，指定した VAX 命令セットに対して AST が不可分に実行されるような Alpha 命令シーケンスを作成します。 AST は，このような不可分な操作を実行する Alpha 命令シーケンスの途中でも受け付けることができますが， AST ルーチンが終了したときに，命令シーケンスは最初から再起動されます。このため，/PRESERVE=INSTRUCTION_ATOMICITY 修飾子を指定した場合，特定のコード・シーケンスの実行速度は半分に低下する可能性があります (トランスレータが AST の不可分なコードを生成する VAX 命令の一覧と，ソフトウェア・トランスレータについての詳しい情報については，『DECmigrate for OpenVMS AXP Systems Translating Images』を参照してください)。

- 読み込み/書き込みの粒度

  トランスレート時修飾子/PRESERVE=MEMORY_ATOMICITY を使用した場合， VEST は，バイト書き込みまたはワード書き込みの不可分性を保証します。この修飾子を指定すれば，メインライン・ルーチンと AST ルーチンはロングワードまたはクォドワードに格納されている隣接するバイトを相互に妨害せずに更新できます。/PRESERVE=MEMORY_ATOMICITY 修飾子は，自然にアラインされていないロングワードおよびクォドワード境界と交差するデータの不可分なアクセスを保証します。ただし，これらの修飾子を指定した場合，実行速度は半分に低下する可能性があります。

- ページ・サイズとゆるやかな保護

  VAX イメージを Alpha システムで実行できるようにするために， VEST とイメージ・アクティベータは， VAX イメージ・セクションを大きいページにマッピングします。 8KB ページをサポートする Alpha プロセッサでは，最大 16 ページの VAX ページを 1 ページに格納できます。しかし，この大きいページは 1 つのページ・テーブル・エントリによってのみ記述されるため，そのページには 1 つの保

護と 1 つのバッキング・ストアだけしか割り当てることができません。したがって， VEST が Alpha ページに割り当てる保護は，マッピングする Alpha イメージ・セクションに関連する保護のうち，もっとも制限のゆるやかな保護です。このため，アクセス違反を検出するために厳しい保護に依存している VAX イメージは，トランスレートされた場合， Alpha システムで正しく実行されません。

この問題に対処するための方法として，省略時のリンカ・オプション/BPAGE を使用して VAX でプログラムを再リンクし，ページを 64KB 境界にアラインする方法があります。

- 正確な算術演算例外

  VEST では，トランスレート時に/PRESERVE=FLOAT_EXCEPTIONS 修飾子または/PRESERVE=INTEGER_EXCEPTIONS 修飾子を使用することにより，特定の例外タイプに対して正確な例外報告を設定できます。これらの修飾子を指定した場合には，一部のコード・セグメントの実行速度は半分に低下します。

- VAX 命令の実行時作成

  実行時に作成される VAX 命令は，トランスレーションのもとでエミュレーションによって実行されます。しかし，エミュレートした命令はトランスレートされた命令より大幅に実行速度が遅く，アプリケーションの重要な部分の性能を向上するために実行時にコードを生成する場合，これは問題になります。

アーキテクチャ上のさまざまな相違点がイメージのトランスレートによってどのようにサポートされるかについては，表 2–3 を参照してください。

2.7.2 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， Alpha システムではネイティブな Alpha 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，ネイティブな Alpha イメージからトランスレートされた共有可能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。

ネイティブな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， VAX 呼び出し規則と Alpha 呼び出し規則の間で呼び出しを実行できなければなりません。ネイティブ・イメージとトランスレートされたイメージが次の条件を満たす場合には，特別な処理は必要ありません。

- ネイティブな Alpha イメージのルーチン・インターフェイス・セマンティックとデータ・アラインメント規則が VAX イメージのこれらのセマンティックおよび規則と等しいこと。
- すべてのエントリ・ポイントが CALL*x*であること。つまり，外部 JSB エントリ・ポイントが存在しないこと。高級言語で作成されたコードの場合には，この条件は満たされます。

- ネイティブな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。

一方の呼び出し規則を使用するプロシージャ (呼び出し元) が別の呼び出し規則を使用するプロシージャ (呼び出し先) を呼び出す場合には，**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コール・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の対応するコール・フレームと引数リストを作成し，制御を呼び出し先プロシージャに渡します。呼び出し先プロシージャが実行を終了すると，ジャケット・ルーチンを通じて，制御を呼び出し元に戻します。ジャケット・ルーチンは呼び出し先ルーチンのリターン・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，制御を呼び出し元プロシージャに戻します。

OpenVMS Alpha 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中でネイティブな Alpha の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 を使用してします。

特定の場合には，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，次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 共有可能ライブラリ
- 転送ベクタに読み込み/書き込みデータを含むライブラリ
- VAX 固有の関数を登録したライブラリ
- 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有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な Alpha 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) に登録されたことを意味します。)

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 Alphaオペレーティング・システムとともに提供されるトランスレートされた共有可能イメージ (たとえば，ネイティブな Alpha 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，これらのジャケット・ルーチンを使用すれば，トランスレートされた共有可能イメージをネイティブな Alpha イメージから呼び出すことができます。

3
アプリケーションの移行

アプリケーションを実際に Alpha 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. 移行評価の基礎を設定するために VAX システムでアプリケーションをテストする
3. Alpha 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

3.1 移行環境の設定

ネイティブな Alpha 環境は， VAX システムと同様に完全な開発環境です

現在のところ，移行したアプリケーションのデバッグおよびテストは， Alpha システム上で行わなければなりません。

Alpha 移行環境の重要な要素は DEC からサポートされ， DEC 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

Alpha 移行環境にとって必要な資源を見積もるには，次の問題を考慮しなければなりません。

- Alpha システムでは，イメージ・サイズが従来より大きくなること

  VAX システムと Alpha 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。

- Alpha 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こと
- 必要な CPU

VEST を使用すると，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。これは，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大きく依存するからです)。また， VEST 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，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 命令と新しい Alpha 命令の両方が含まれます。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 のメモリを装備した 6 VUP 以上のマルチプロセッサ・システム
- 1 GB のシステム・ディスク
- 各アプリケーション当り 1 GB のディスク

マルチプロセッサ・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。

- コンパイラまたは 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，次の要素を確認しなければなりません。

- 移行ツール

  次のツールも含めて，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
    - VEST と VEST/DEPENDENCY
    - TIE
  - 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

- 論理名

  VAX バージョンと Alpha バージョンのツールとファイルをそれぞれ適切に指すように，論理名は矛盾のないように定義しなければなりません。詳しくは第 3.4 節を参照してください。

- コンパイルとリンク・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。

- ソースの管理とイメージのビルドのためのツール
  - CMS
  - MMS

ネイティブな Alpha 開発
VAX で使用できる標準的な開発ツールはすべて， Alpha システムでもネイティブ・ツールとして提供されています。

トランスレーション
VEST ソフトウェア・トランスレータは VAX システムと Alpha システムの両方で実行されます。 Translated Image Environment (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な環境であり， OpenVMS Alpha 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 Alpha システム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，移行計画を適切に作成している場合には，この最終段階はかなり簡単に終了できます。多くのプログラムはまった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 Alpha システムで実行できるようになります。

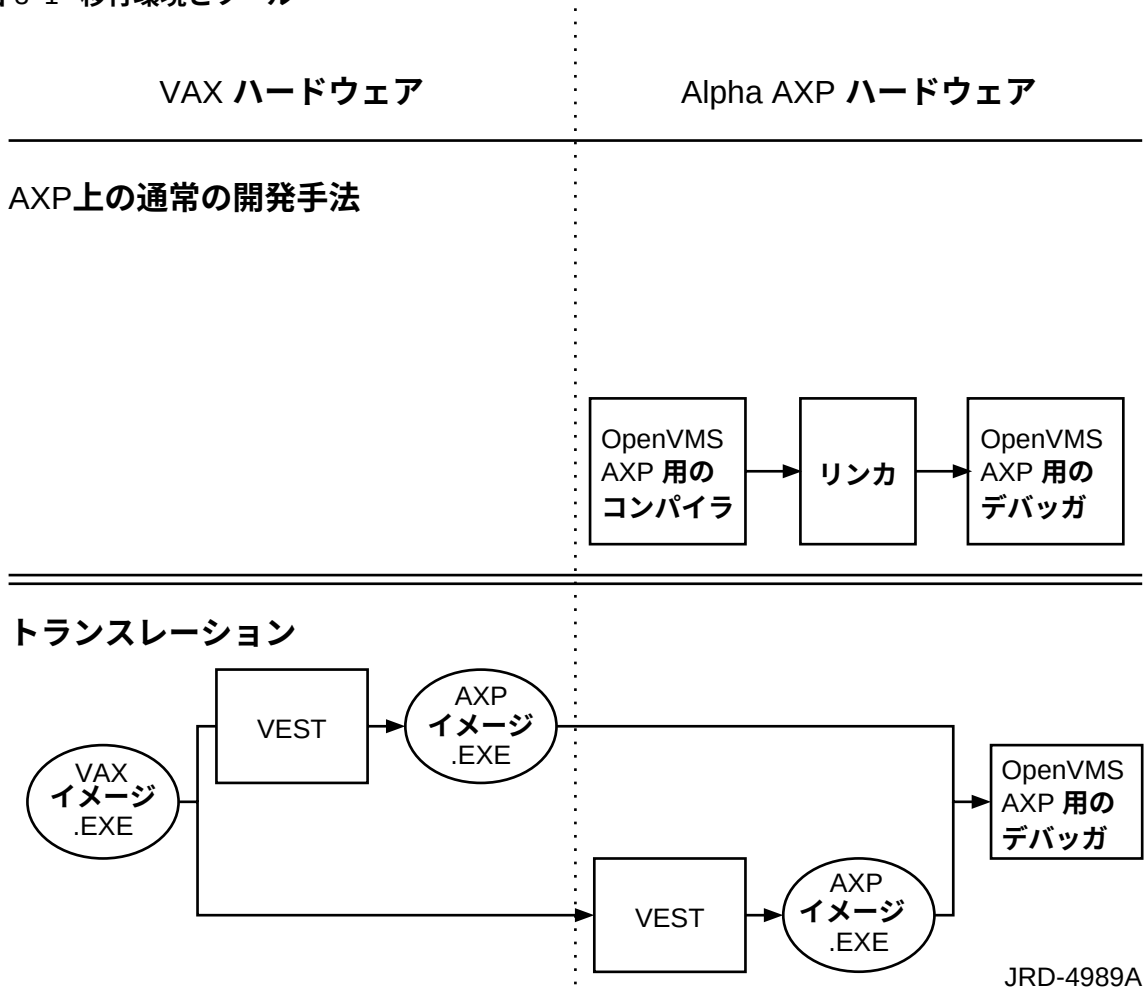
コードの実際の変換についての詳しい説明は， OpenVMS Alpha の移行に関する次の解説書を参照してください。

- 『DECmigrate for OpenVMS AXP Systems Translating Images』
- 『Porting VAX MACRO Code to OpenVMS Alpha』

これらの各解説書についての説明は，本書のまえがきを参照してください。

2 つの移行環境と，各環境で使用される基本的なツールは，図 3–1 に示すとおりです。

図 3–1 移行環境とツール



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。

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，複数の Alpha プラットフォーム間で移植可能にするために，リンカは Alpha システムの省略時のページ・サイズを 64KB として設定します。この結果，各プロセッサのシステム・ページ・サイズとは無関係に， OpenVMS Alpha イメージはどの Alpha プロセッサでも実行可能になります。また， Alpha の共有可能イメージは，普遍的なエントリ・ポイントとシンボルを宣言するために， VAX 転送ベクタ・メカニズムではなく，リンカ・オプション・ファイル内のシンボル・ベクタ宣言を使用します。

Alpha プラットフォー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ます。

3.2.1.1 ネイティブな Alpha 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，ネイティブな Alpha イメージが作成され，このイメージは RISC アーキテクチャの性能上の利点をすべて利用して， Alpha 環境で実行されます。 Alpha コードでは，一連の高度に最適化されたコンパイラを使用します。これらのコンパイラは共通の最適化コード・ジェネレータを備えています。しかし，各言語に対して異なるフロント・エンドを使用し，これらの各フロント・エンドは現在の VAX コンパイラと互換性があります。

OpenVMS Alpha オペレーティング・システムバージョン 7.1 では，次の言語に対してネイティブな Alpha コンパイラが提供されています。

- Ada
- BASIC
- C
- C++
- COBOL
- FORTRAN
- Pascal
- PL/I
- MACRO-32 (クロス・コンパイラ)

OpenVMS Alphaの将来のリリースでは， LISP も含めた他の言語のためのネイティブ・コンパイラも提供されます。

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

一般に， Alpha コンパイラには，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おり， VAX アーキテクチャに依存するコードをほとんど変更せずに， Alpha システムでも実行できるようにしています。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 2–3 を参照してください。

OpenVMS Alpha システムの一部のコンパイラは， OpenVMS VAX システムの対応するコンパイラでサポートされない新しい機能をサポートします。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ます。たとえば， OpenVMS Alpha システム用の DEC C コンパイラでは，VAX C互換性モードをサポートし，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。

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をインプリメントしています。VAXコンピュータのハードウェア・アーキテクチャはAlphaコンピュータの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 Alphaシステム用のDEC 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可能性のあるアーキテクチャに依存する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。

DEC Cコンパイラにはヘッダ・ファイルがあり，各データ型に対してマクロを定義しています。これらのマクロは，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マクロを使用します。

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。

Alphaコンパイラを使用して，OpenVMS VAXプログラムをOpenS Alphaシステムに移行する処理の詳細については，第11章を参照してください。

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

既存のVAX MACROコードをOpenVMS Alpha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 Alphaを使用します。OpenVMS Alpha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)のルーチン・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 Alpha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。しかし，CALL_ENTRYルーチンの内部からでなければ，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。

コンパイラはOpenVMS Alphaシステム用に最適化されたコードを生成しますが，高レベルの制御機能をプログラマに提供するVAX MACRO言語の多くの機能は，OpenVMS Alphaシステム用の最適なコードを生成することを困難にします。OpenVMS Alpha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コンパイラの詳細については，『Porting VAX MACRO Code to OpenVMS Alpha』を参照してください。

3.2.1.3 その他の開発ツール

ネイティブな Alpha イメージを作成するために，コンパイラの他にもいくつかのツールが提供されます。

- OpenVMS リンカ

  OpenVMS リンカは， VAX オブジェクト・ファイルまたは Alpha オブジェクト・ファイルを受け付け， VAX イメージまたは Alpha イメージを作成できるようになりました。また， VAX ハードウェアで Alpha イメージを作成できるため，クロス・リンカとして機能します。

- OpenVMS デバッガ

  OpenVMS Alpha で実行される OpenVMS デバッガは，現在の OpenVMS VAX デバッガと同じコマンド・インターフェイスを使用します。 OpenVMS Alpha システムでも， VAX システム上のグラフィカル・インターフェイスが提供されています。

- 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， VAX ライブラリまたは Alpha 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。

- MACRO–64 Assembler for OpenVMS Alpha

  OpenVMS Alpha システム用の MACRO–64 アセンブラは，すべての Alpha コンピュータを対象にしたネイティブ・アセンブラです。VAX MACROアセンブラは OpenVMS VAX オペレーティング・システムに含まれていますが， MACRO–64 アセンブラは OpenVMS Alpha オペレーティング・システムに含まれていません。このアセンブラは個別に購入できます。一般に，中級言語および高級言語コンパイラは， MACRO–64 アセンブラより性能の高いコードを OpenVMS Alpha システム用に生成します。 OpenVMS Alpha システム用に新たなプログラムを開発する場合には，中級コンパイラまたは高級コンパイラのご使用をお勧めします。MACRO–64アセンブラの詳細については，『MACRO–64 Assembler for OpenVMS AXP Systems Reference Manual』を参照してください。

- ANALYZE/IMAGE

  ANALYZE/IMAGE ユーティリティは VAX イメージまたは Alpha イメージを分析できます。

- ANALYZE/OBJECT

  ANALYZE/OBJECT ユーティリティは VAX オブジェクトまたは Alpha オブジェクトを分析できます。

- DECset

  DECset は包括的な CASE ツール群であり， Language Sensitive Editor (LSE), Source Code Analyzer (SCA), Code Management System (CMS), Module Management System (MMS) をはじめ，多くの要素で構成されます。

3.2.2 トランスレーション

Alpha 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』を参照してください。一般に，この処理は簡単ですが，エラーなしにトランスレートするには，コードを少し変更しなければならないことがあります。

3.2.2.1 VAX Environment Software Translator (VEST) と Translated Image Environment (TIE)

ユーザ・モードの VAX イメージをOpenVMS Alphaに移行するための主なツールは，静的トランスレータと実行時サポート環境です。

- VAX Environment Software Translator (VEST) は， VAX イメージを分析し，イメージをトランスレートし，同じ機能を実行するイメージを作成するユーティリティです。 VEST を使用すれば，次の処理が可能です。
  - VAX イメージをトランスレートできるかどうかを判断する
  - VAX イメージを Alpha イメージにトランスレートする
  - イメージ内で OpenVMS Alpha と互換性のない部分を識別し，これらの互換性のない部分をソース・ファイルで修正する方法を入手する
  - トランスレートされたイメージの実行時の性能を改善する方法を判断する
- Translated Image Environment (TIE) は，トランスレートされたイメージを実行時にサポートする Alpha の共有可能イメージです。 TIE は，トランスレートされたイメージを実行するために OpenVMS VAX に類似した環境を準備し，ネイティブな Alpha システムとのすべてのやりとりを処理します。 TIE には次の要素が含まれています。
  - VAX 命令インタプリタ
    次の機能をサポートします。
    - VAX システムでの実行に類似した VAX 命令の実行 (命令の不可分性も含む)
    - サブルーチンとしての複雑な VAX 命令
  - VAX と互換性のある例外ハンドラ
  - ネイティブ・コードとトランスレートされたコードの間のやりとりを可能にするジャケット・ルーチン
  - エミュレートされた VAX スタック

  トランスレートされたイメージを実行する場合には， TIE が自動的に起動されます。 TIE を呼び出す必要はありません。

VEST は，できるだけ多くの VAX コードを Alpha コードにトランスレートします。TIE は， Alpha 命令に変換できなかった VAX コードを解釈します。たとえば，次のコードは TIE によって解釈されます。

- VEST が静的に識別できなかった命令
- H 浮動小数点および D56 (56 ビットの D 浮動小数点) 浮動小数点演算

命令の解釈は速度の遅い処理であり，おそらく平均的な 1 つの VAX 命令に対して 100 前後の Alpha 命令が必要となるため， VEST は実行時にインタプリタを通じた解釈の必要性をできるだけ少なくするために，できるだけ多くの VAX コードをトランスレートしようとします。トランスレートされたイメージは，ネイティブな Alpha イメージと比較すると，約 3 分の 1 の速度で実行されます。ただし，この速度は TIE がどれだけの VAX コードを解釈しなければならないかに応じて異なります。トランスレートされた VAX イメージは，少なくとも VAX ハードウェアと同じ速度で実行されます (同じレベルのプロセス技術を使用した CPU の場合)。

Alpha システムで VAX イメージの動的解釈を指定することはできません。イメージをOpenVMS Alphaで実行するには，その前に VEST を使用してイメージをトランスレートしなければなりません。

VAX イメージをトランスレートすると， Alpha ハードウェアで実行されるイメージが作成されます。このようにして作成される Alpha イメージは，単に VAX イメージを解釈したバージョンやエミュレートしたバージョンではなく，元の VAX イメージ内の命令が実行する操作と同じ操作を実行する Alpha 命令を含むイメージです。Alpha の.EXE ファイルには，元の VAX イメージが完全に登録されているため， TIE は VEST がトランスレートできなかったコードを解釈できます。

VEST の分析機能は，トランスレートする場合だけでなく，再コンパイルする予定のプログラムを評価する場合も役立ちます。

VEST と 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。このマニュアルでは，フローグラフなども含めて， VEST が作成するすべての出力とその解釈方法が詳しく説明されています。また， VEST が作成するイメージ情報ファイル (IIF) から提供される情報を利用して，トランスレートされたイメージの実行時の性能を向上する方法も説明されています。

3.3 移行したアプリケーションのデバッグとテスト

アプリケーションを OpenVMS Alpha に移行した後，デバッグしなければなりません。

また，正しい操作が実行されるかどうかを調べるためにアプリケーションをテストすることも必要です。

3.3.1 デバッグ

OpenVMS オペレーティング・システムでは，次のデバッガが準備されています。

- OpenVMS デバッガは VAX プログラムとネイティブな Alpha プログラムの両方のデバッグをサポートします。ただし，トランスレートされたイメージのデバッグはサポートしません。

  OpenVMS デバッガはシンボリック・デバッガです。つまり，このデバッガを使用する場合には，プログラムで使用しているシンボル (変数名やルーチン名，ラベルなど) によってプログラムの位置を参照できます。プログラム位置を参照するために，メモリ・アドレスやマシン・レジスタを指定する必要はありません。ただし，メモリ・アドレスやマシン・レジスタを指定したい場合は，必要に応じて指定できます。

  OpenVMS デバッガは通常，トランスレートされたイメージに対して動作しません。しかし，トランスレートされたイメージは内部で VAX レジスタをエミュレートしているので， SHOW CALL や SHOW STATE コマンドでデバッグに有用な VAX コンテキストを得ることができます。

- Delta デバッガは， OpenVMS VAX プログラムと OpenVMS Alpha プログラムのデバッグをサポートします。このデバッガはまた，トランスレートされたイメージのデバッグもサポートします。

  Delta デバッガはアドレス・ロケーション・デバッガです。つまり，このデバッガを使用する場合には，アドレス・ロケーションによってプログラム位置を参照しなければなりません。このデバッガは主に，特権付きプロセッサ・モードまたは高い割り込みレベルで実行されるプログラムをデバッグするために使用します。

- システムコード・デバッガはシンボリック・デバッガであり，任意の IPL で動作する非ページング・コードとデバイス・ドライバをデバッグできます。

- ヒープ・アナライザは，リアルタイムでメモリの使用状況を示すグラフィックを表示します。この機能を使用すると，非常に頻繁に行われている割り当てや，非常に大きいメモリ・ブロック，フラグメンテーション，メモリ・リークなど，効率の悪いメモリの使用状況を直ちに識別できます。

デバッグは Alpha ハードウェアで実行しなければなりません。

3.3.1.1 OpenVMS デバッガによるデバッグ

OpenVMS Alpha システムでは，次の DEC 言語で作成したプログラムに対してデバッガを使用できます。

- DEC Ada
- DEC BASIC
- DEC C
- DEC C++
- DEC COBOL

- DEC Fortran (VAX システム)
- Digital Fortran (Alpha システム)
- MACRO–32 (MACRO-32 コンパイラでのコンパイル)
- MACRO–64
- DEC Pascal
- DEC PL/I

OpenVMS デバッガには，OpenVMS Alpha コードのアーキテクチャ上の違いに対処する機能がいくつか含まれています。これらの機能を使用すると，OpenVMS Alpha システムに移植するコードを簡単にデバッグできるようになります。たとえば，SET コマンドの/UNALIGNED_DATA 修飾子を使用すると，アラインされていないデータにアクセスする命令 (たとえば，ワード境界にないデータにアクセスする load word 命令など) のすぐ後で，デバッガはブレークします。

どのルーチンでも，SET コマンドに/RETURN 修飾子を指定できます。OpenVMS VAX システムの場合のように，CALLS 命令や CALLG 命令を使用して呼び出したルーチンに制限されません。OpenVMS Alpha システム固有の機能の詳細については，『OpenVMS デバッガ説明書』を参照してください。

OpenVMS デバッガを使用して，移行したアプリケーションを Alpha システムでデバッグする場合には，次のことを考慮しなければなりません。

- Alpha コンパイラが使用できる言語で作成されたプログラムならばデバッガを使用できます。
- デバッガは，インストールされている常駐イメージのデバッグをサポートしません。インストールされている常駐イメージについての詳しい説明は，『OpenVMS システム管理者マニュアル (下巻)』を参照してください。
- このデバッガはインライン・ルーチンのデバッグをサポートしません。インライン・ルーチンをデバッグしようとすると，デバッガはルーチンをアクセスできないことを示すメッセージを出力します。

```
DBG> %DEBUG-E-ACCESSR, no read access to address 00000000
```

OpenVMS デバッガによるデバッグについての詳しい説明は，『OpenVMS デバッガ説明書』を参照してください。

3.3.1.2 Delta デバッガよるデバッグ

Delta/XDelta デバッガ (DELTA/XDELTA) は OpenVMS Alpha システムで動作し，Alpha アーキテクチャで必要となる新しいコマンドを提供し，また，既存のコマンドの一部も拡張しています。たとえば，ベース・レジスタの表示は 16 進数ではなく，10 進数で表示し，別のプロセスの内部プロセス識別 (PID) 番号を確認する機能も追加されています。Delta/XDelta デバッガが OpenVMS Alpha システムでどのように動作するかについての詳しい説明は，『OpenVMS Delta/XDelta Debugger Manual』を参照してください。

Delta デバッガを使用すれば，部分的または完全にトランスレートされたアプリケーションをデバッグできます。

トランスレートされたアプリケーション
トランスレートされたイメージをデバッグする場合には，次のことを確認しなければなりません。

- トランスレートを行うプログラムが OpenVMS VAX バージョン 7.1 のもとで正しく動作するかどうかを確認しなければなりません。
- ランタイム・ライブラリの IIF ファイルと VEST が使用中の OpenVMS Alpha のバージョンと同じリリースであるかどうかを確認しなければなりません。
- Alpha 命令および VAX 命令を得るために， VEST の/DEBUG，/LIST および /SHOW=MACHINE_CODE 修飾子を使用します (リストのアスタリスクは， VAX 命令を示しています)。比較を行うために， VAX イメージのマップ・ファイルとリスティング・ファイルを用意してください。

アプリケーションの混在
ネイティブな Alpha コードとトランスレートされたコードが混在するアプリケーションをデバッグするには，/TIE 修飾子を使用してアプリケーションのネイティブな部分がコンパイルされているかどうかを確認しなければなりません。さらに，/NONATIVE_ONLYリンカ・オプションを使用してアプリケーションをリンクしなければなりません。

Delta デバッガによるデバッグについての詳しい説明は，『OpenVMS Delta/XDelta Debugger Manual』を参照してください。

3.3.1.3 OpenVMS Alpha システムコード・デバッガによるデバッグ

OpenVMS Alpha のシステムコード・デバッガは，任意の IPL で動作する非ページング・システム・コードとデバイス・ドライバをデバッグするために使用します。OpenVMS Alpha のシステムコード・デバッガはシンボリック・デバッガです。したがって，ソース・コードに指定するときと同様に，変数名，ルーチン名などを指定できます。また，ソフトウェアが実行しているソース・コードを表示し，ソース行を 1 ステップずつ実行することもできます。

システムコード・デバッガを使用するには， 2 台の Alpha システムが必要です。

このデバッガを使用すると，次の言語で作成されたコードをデバックできます。

- C
- BLISS
- VAX MACRO

注意

BLISS コンパイラは，OpenVMS VAX バージョン 7.1 と OpenVMS Alpha バージョン 7.1 に同梱されている OpenVMS フリーウェア CD に登録されています。

OpenVMS Alpha のシステムコード・デバッガは，各言語の構文，データ型，演算子，式，有効範囲に関する規則，その他の構造を認識します。プログラムが複数の言語で作成されている場合には，デバッグ・セッションでデバッグ・コンテキストを 1 つの言語から別の言語に変更できます。

Step 2 ドライバと OpenVMS Alpha システムコード・デバッガの詳細については，『Writing OpenVMS Alpha Device Drivers in C』を参照してください。

3.3.2 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。それはシステム・ダンプ・アナライザ (SDA) とクラッシュ・ダンプ・ユーティリティ・エクストラクタ (CLUE) です。

3.3.2.1 システム・ダンプ・アナライザ

OpenVMS Alpha システムのシステム・ダンプ・アナライザ (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，クラッシュ・ダンプ・ユーティリティ・エクストラクタ (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・レジスタやデータ構造体など OpenVMS Alpha 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 Alpha 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 種類のコンピュータで少し違います。 Alpha の実行パスは， VAX の実行パスの場合より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理解しておく必要があります。同様に， Alpha システムで SDA を使用するには，まず， Alpha 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，スタックで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 は次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・**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，性能を大幅に向上できます。

- 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・オフセットに変換できます。
- 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。

3.3.2.2 クラッシュ・ログ・ユーティリティ・エクストラクタ (CLUE)

クラッシュ・ログ・ユーティリティ・エクストラクタ (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 Alpha) は，システム・クラッシュを永久的に記録します。

OpenVMS VAX と OpenVMS Alpha でのインプリメント上の相違点は，表 3–1 に示すとおりです。

表 3–1 OpenVMS VAX と OpenVMS Alpha の CLUE の相違点

属性	OpenVMS VAX	OpenVMS Alpha
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・ダンプのデバッグの他の使い方	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。
ドキュメンテーション	『OpenVMS システム管理者マニュアル』と『OpenVMS システム管理ユーティリティ・リファレンス・マニュアル』の Bookreader バージョン。	『OpenVMS システム管理者マニュアル』と『OpenVMS Alpha System Dump Analyzer Utility Manual』の Bookreader バージョン。

3.3.3 テスト

移行したバージョンの性能と機能を元の 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。

テストではまず，VAX アプリケーションに対して一連のテストを実行することにより，アプリケーションに対して基準となる結果を設定します。

アプリケーションを Alpha システムで実行した後，次の 2 種類のテストを実行できます。

- 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

3.3.3.1 VAX テスト

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 Alpha にアプリケーションを移行した後，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，次の操作が必要です。

1. 移行する前に，アプリケーションにとって必要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，一連のテストも移行する (テストが Alpha でまだ準備されていない場合)。
3. Alpha システムでテストを検証する
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，リグレッション・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。

3.3.3.2 Alpha 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かなり長くかかりますが，移行固有の問題を調べるためのテストをいくつか追加しなければなりません。特に次の点に注意してください。

- コンパイラの相違点

  最適化およびデータ・アラインメントの変更

- アーキテクチャの相違点

  たとえば命令の不可分性，メモリの不可分性，読み込み/書き込み順序などの変更

- 統合

  異なる言語で作成されたモジュールや，トランスレートしなければならなかったモジュールの統合

3.3.4 潜在的なバグの発見

作業方法に何も問題がなく，移行に関するすべての指示に従っているにもかかわらず， OpenVMS VAX システムでは問題が発生したことのないプログラムでバグを検出することがあります。たとえば， VAX コンピュータでは，プログラムで一部の変数を初期化しないエラーが発生しても問題になりませんが， Alpha コンピュータでは演算例外が発生します。 2 つのアーキテクチャでは使用できる命令が違い，コンパイラが命令を最適化する方法も変更されているため，移行処理で同じような問題が発生する可能性があります。これまで隠れていたバグをすべて解決できるような新しい方法はありませんが，プログラムを移植した後，他のユーザが実際に使用を開始する前に，プログラムを徹底的にテストする必要があります。

3.4 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，移行によって他のソフトウェアとのやりとりに問題が発生していないかどうかを確認しなければなりません。

VAX と Alpha システム間の相互操作性に関する問題の原因として，次のことが考えられます。

- VMS クラスタ環境の Alpha 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，確認する必要があります。
- イメージ名

  混在する環境では，アプリケーションが正しいバージョンを参照するかどうかに関して，次の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な Alpha バージョンが同じ名前を持つこと

  – イメージをトランスレートしたバージョンで名前の最後に "_TV"という文字列が追加されていること
- 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と考えられますが，トランスレートされたイメージでは，元の VAX イメージと同様にデータはアラインされていない可能性があります。

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 Alpha システムに移行する (migrate) 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 Alpha 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マシン・アーキテクチャから分離します。さらに， Alpha 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。したがって， Alpha 版の各言語のコンパイラと OpenVMS リンカ・ユーティリティを使用すれば，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でき，ネイティブな Alpha 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 Alpha 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。

内側の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 ネイティブな Alpha 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， Alpha 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。 Alpha システムで共通に使用できるプログラミング言語のコンパイラについての詳しい説明は，第 11 章を参照してください。

Alpha システムで使用出来るコンパイラは，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，また， 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省略時のファイル・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。

しかし， VAX システムのコンパイラがサポートする一部の機能は， Alpha 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， Alpha 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ない新しい機能をサポートします。また互換性を維持するために，一部の Alpha コンパイラは互換モードをサポートします。たとえば， DEC C for OpenVMS Alpha システムのコンパイラはVAX C互換モードをサポートし，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。

4.2 Alpha 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な Alpha イメージを作成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，出力ファイルを作成します。たとえば，省略時の設定では，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。

Alpha 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構築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。次のリストは，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，これらのリンカの変更点を説明しています。詳しくは『OpenVMS Linker Utility Manual』を参照してください。

- **共有可能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可能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，共有可能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。 Alpha システムでは，転送ベクタ・ファイルを作成するかわりに， SYMBOL_VECTOR オプション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。 Alpha 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。

- **イメージの性能の最適化**— Alpha システムでは，リンカは作成したイメージの性能を向上させるために最適化を行います。さらにリンカは，常駐イメージとしてインストール可能な共有可能イメージを作成できます。この結果，さらに性能を向上できます。
- **共有可能イメージの暗黙の処理**—VAX システムでは，リンク操作で共有可能イメージを指定した場合，リンカは共有可能イメージがリンクされる対象となるすべての共有可能イメージも処理します。 Alpha システムでは，ビルド・プロシージャがこれらの共有可能イメージを含む場合，これらのイメージを明示的に指定しなければなりません。

リンカは表 4–1 にある Alpha システム固有の修飾子とオプションをサポートします。表 4–2 は， VAX システムではサポートされ， Alpha システムのリンカではサポートされないリンカ修飾子を示しています。

表 4–1 OpenVMS Alpha 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	リンカがデマンド・ゼロ・イメージ・セクションを作成する方法を制御する。
/DSF	OpenVMS Alpha システムコード・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。
/GST	共有可能イメージに対してグローバル・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 (省略時の設定)。ランタイム・キット用に共有可能イメージを作成する場合には，/NOGST を指定する方が一般的である。
/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 (省略時の設定)。/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 (省略時の設定)。 リンク中に/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。ネイティブな Alpha イメージが，トランスレートされた VAX イメージと連携動作するには，ジャケット・ルーチンが必要である。
/REPLACE	コンパイラによって要求された場合，作成中のイメージの性能を向上するための最適化を行うことをリンカに要求する (省略時の設定)。

(**次ページに続く**)

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

修飾子	説明
/SECTION_BINDING	常駐イメージとしてインストール可能な共有可能イメージを作成することをリンカに要求する。
/SYSEXE	リンク操作で解釈されなかったシンボルを解釈するために 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=オプション	共有可能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。省略時の設定では，リンカはユニバーサル・シンボルだけを登録する。
SYMBOL_VECTOR=オプション	Alpha 共有可能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 OpenVMS VAX システム固有のリンカ・オプション

オプション	説明
BASE=オプション	リンカがイメージを割り当てるベース・アドレス (先頭アドレス) を指定する。
DZRO_MIN=オプション	リンカがイメージ・セクションからページを取り出し，それを新たに作成したデマンド・ゼロ・イメージ・セクションに配置する前に，リンカがイメージ・セクションで検出しなければならない連続した初期化されていないページの最小数を指定する。デマンド・ゼロ・イメージ・セクション (初期化されたデータを含まないイメージ・セクション) を作成すると，リンカはイメージのサイズを小さくできる。
ISD_MAX=オプション	イメージ内で認められるイメージ・セクションの最大数を指定する。
UNIVERSAL=オプション	共用可能イメージ内のシンボルをユニバーサルとして宣言し，リンカがそのシンボルを共用可能イメージのグローバル・シンボル・テーブルに登録するようにする。

4.3 VAX システムと Alpha システムの算術演算ライブラリ間の互換性

OpenVMS Mathematics (MTH$) ランタイム・ライブラリに対して標準的な VMS 呼び出しインターフェイスを使用する算術演算アプリケーションを， Alpha システムに移行するときには， MTH$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$ルーチンを Digital Portable Mathematics Library (DPML) for Alpha システムの対応する math$に変換するためのジャケット・ルーチンが準備されているからです。しかし， 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ーフェイスを使用するのではなく，選択した高級言語 (たとえば DEC C や DEC FORTRAN など) を通じて提供される DPML ルーチンを使用することが適切です。 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しい説明は，『Digital Portable Mathematics Library』を参照してください。

4.4 ホスト・アーキテクチャの判断

アプリケーションが VAX システムで実行されているのか， Alpha 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 アイテム・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，$GETSYI システム・サービスは 1 という値を戻します。アプリケーションが Alpha システムで実行されている場合には，$GETSYI システム・サービスは 2 という値を戻します。

例 4–1 は， F$GETSYI DCL コマンドを呼び出し， ARCH_TYPE アイテム・コードを指定することにより， DCL コマンド・プロシージャでホスト・アーキテクチャを判断する方法を示しています (アプリケーションで$GETSYI システム・サービスを呼び出す例については，第 5.4 節を参照してください。その例では， Alpha システムのページ・サイズを入手するためにシステム・サービスが使用されています)。

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
$ ON_VAX:
$ !
$ ! Do VAX-specific processing
$ !
$ exit
$ ON_ALPHA:
$ !
$ ! Do Alpha-specific processing
$ !
$ exit
```

しかし， ARCH_TYPE アイテム・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 システムだけ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 4–3 に示した$GETSYI システム・サービスの他のアイテム・コードを使用しなければなりません。

表 4–3 ホスト・アーキテクチャを指定する$GETSYI アイテム・コード

キーワード	使用方法
ARCH_TYPE	VAX システムでは 1 を戻す。 Alpha システムでは 2 を戻す。 Alpha システムと， OpenVMS バージョン 5.5 またはそれ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 マシンでは“VAX”というテキスト文字列を戻し， Alpha マシンでは“Alpha”というテキスト文字列を戻す。 Alpha システムと，OpenVMS 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戻す。 1024 以上のすべての値は Alpha システムを示す。
CPU	特定の CPU を識別する整数を戻す。 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 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・セットとしてロックする操作
- 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値を指定する場合は， 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。 Alpha アーキテクチャでは， 8K バイト， 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。したがって，ルーチンへの引数として指定する値を調べ，それらの値がアプリケーションの必要条件を満足するかどうかを確認しなければなりません。この後の節では，これらの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，上位レベル・ルーチンを使用するメモリ割り当てには影響を与えません。たとえば， C の `malloc`や `free`ルーチンなど，言語固有のメモリ割り当てルーチンや，仮想メモリ領域を操作するランタイム・ライブラリ・ルーチンなどは，ページ・サイズの違いの影響を受けません。

5.1.1 互換性のある機能

システム・サービスやランタイム・ライブラリ・ルーチンは，Alpha システムにおいてもできる限り VAX システムと同じインターフェイスや戻り値を維持しています。たとえば Alpha 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・カウント値を戻すルーチンは，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

Alpha システムでは，$CRMPSC **システム・サービスを使用して (さらに** SEC$M_PFNMAP **フラグ・ビットをセットして) ページ・フレーム・セクションを作成する場合には，ページ・カウント引数** (pagcnt) **に指定された値は** CPU **固有のページ・サイズとして解釈され，ページレットの値としては解釈されません。**

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，一部のルーチンの Alpha システムでの動作は 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 Alpha 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。Alpha 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，これらの引数がルーチンの OpenVMS Alpha バージョン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しい説明は，『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

引数		Alpha システムの動作
Adjust Working Set Limit ($ADJWSL) ルーチン		
pagcnt	現在のワーキング・セット・リミットに加算される (またはワーキング・セット・リミットから減算される) ページ数を指定する。	値をページレットとして解釈し，CPU 固有のサイズのページを表現するために切り上げるか，または切り捨てる。
wsetlm	現在のワーキング・セット・リミットの値を指定する。	値をページレットとして解釈し，CPU 固有のサイズのページを表現するために切り上げるか，または切り捨てる。
Create Process ($CREPRC) ルーチン		
quota	省略時のワーキング・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。
Create Virtual Address ($CRETVA) ルーチン		
inadr	割り当てられるメモリの先頭のアドレスと末尾のアドレスを指定する。末尾アドレスが先頭アドレスと同じである場合には，1 ページが割り当てられる。	CPU 固有のページ境界にアラインされるように，アドレスは切り上げられるか，または切り捨てられる。
retadr	呼び出しの影響を受けるメモリの実際の先頭アドレスと末尾アドレスを指定する。	変更されない。
Create and Map Section ($CRMPSC) ルーチン		
inadr	再びマッピング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$M_EXPREG フラグが設定されている場合には，割り当てが P0 空間であるのか，P1 空間であるのかを判断し，その結果をもとに先頭アドレスが解釈される。	アドレスは CPU 固有のページにアラインしなければならない (SEC$M_EXPREG フラグが設定されていない場合)。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しい説明は第 5.3 節を参照)。
retadr	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指定する。	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戻す。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	作成またはマッピングされるセクションのタイプと属性を指定する。	フラグ・ビット SEC$M_NO_OVERMAP は，既存のアドレス空間をマッピングしてはならないことを示す。フラグ・ビット SEC$M_PFNMAP がセットされている場合には，pagcnt 引数は CPU 固有のページとして解釈され，ページレットとしては解釈されない。

(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

引数		Alpha システムの動作
Create and Map Section ($CRMPSC) ルーチン		
relpag	セクション・ファイルのマッピングを開始するページ・オフセットを指定する。	セクション・ファイルへのインデックスとして解釈され，単位はページレットであると解釈される。
pagcnt	マッピングされるファイル内のページ数 (ブロック数) を指定する。	ページレットとして解釈される。切り上げや切り捨ては実行されない。フラグ・ビット SEC$M_PFNMAP がセットされている場合には，pagcnt 引数は CPU 固有のページとして解釈され，ページレットとしては解釈されない。
pfc	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，Alpha 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。システムが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。
Delete Virtual Address ($DELTVA) ルーチン		
inadr	割り当てが解除されるメモリの先頭アドレスと末尾アドレスを指定する。	CPU 固有のページ境界にアラインされるように，アドレスは切り上げられるか，または切り捨てられる。
retadr	削除されたメモリの実際の先頭アドレスと末尾アドレスを指定する。	変更されない。
Expand Program/Control Region ($EXPREG) ルーチン		
pagcnt	512 バイト単位で割り当てるメモリ・サイズを指定する。	ページレットとして解釈される。
retadr	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指定する。	変更されない。
Get Job/Process Information ($GETJPI) ルーチン		
itmlst	プロセスに関して戻される情報を指定する。	JPI$_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。詳しくは『OpenVMS System Services Reference Manual』を参照。
Get Queue Information ($GETQUI) ルーチン		
itmlst	func 引数によって指定された関数を実行するときに使用さ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。詳しくは『OpenVMS System Services Reference Manual』を参照。

(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

引数		Alpha システムの動作
Get Systemwide Information ($GETSYI) ルーチン		
itmlst	1 つ以上のノードに関して戻される情報を指定する。	一部の項目はページレット単位の値として解釈される。SYI$_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。詳しくは『OpenVMS System Services Reference Manual』を参照。
Get User Authorization Information ($GETUAI) ルーチン		
itmlst	ユーザのユーザ登録ファイルからどの情報が戻されるかを指定する。	一部の項目はページレット単位の値を戻す。詳しくは『OpenVMS System Services Reference Manual』を参照。
Lock Page($LCKPAG) ルーチン		
inadr	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。
retadr	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指定する。	変更されない。
Lock Working Set ($LKWSET) ルーチン		
inadr	ロックされるメモリの先頭アドレスと末尾アドレスを指定する。	CPU 固有のページ境界にアラインされるように，アドレスは切り上げられるか，または切り捨てられる。
retadr	実際にロックされたメモリの先頭アドレスと末尾アドレスを指定する。	変更されない。
Map Global Section ($MGBLSC) ルーチン		
inadr	再びマッピング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$M_EXPREG フラグがセットされている場合には，割り当てが P0 空間で実行されるのか，P1 空間で実行されるのかを判断し，その結果に従って先頭アドレスが解釈される。	アドレスは CPU 固有のページにアラインしなければならない (SEC$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しい説明は第 5.3 節を参照)。
retadr	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指定する。	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戻す。
relpag	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。
Purge Working Set ($PURGWS) ルーチン		
inadr	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。

(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

引数		Alpha システムの動作
Set Protection ($SETPRT) ルーチン		
inadr	保護されるメモリの先頭アドレスと末尾アドレスを指定する。	CPU 固有のページ境界にアラインされるように，アドレスは切り上げられるか，または切り捨てられる。
retadr	保護されたメモリの実際の先頭アドレスと末尾アドレスを指定する。	変更されない。
Set User Authorization File ($SETUAI) ルーチン		
itmlst	ユーザのユーザ登録ファイルから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。詳しくは『OpenVMS System Services Reference Manual』を参照。
Send to Job Controller ($SNDJBC) ルーチン		
itmlst	func引数によって指定された関数を実行するときに使用される情報を指定する。	いくつかの項目はページレット単位の値として解釈される。詳しくは『OpenVMS System Services Reference Manual』を参照。
Unlock Page ($ULKPAG) ルーチン		
inadr	アン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。
retadr	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指定する。	変更されない。
Unlock Working Set ($ULWSET) ルーチン		
inadr	アンロックされるメモリの先頭アドレスと末尾アドレスを指定する。	CPU 固有のページ境界にアラインされるように，アドレスは切り上げられるか，または切り捨てられる。
retadr	アンロックされたメモリの実際の先頭アドレスと末尾アドレスを指定する。	変更されない。
Update Section ($UPDSEC) ルーチン		
inadr	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。ディスク上の記憶空間によって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	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。

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	引き数	Alpha システムでの解釈
LIB$GET_VM_PAGE	number-of-pages 割り当てる連続ページのページ数を指定する。	ページレット単位の値として解釈され，CPU 固有のページに切り上げられるか，または切り捨てられる。
LIB$FREE_VM_PAGE	number-of-pages 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 固有のページになるよう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) を使用すれば，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 または P1 領域のサイズを拡張する方法
- 指定した位置からはじまり，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

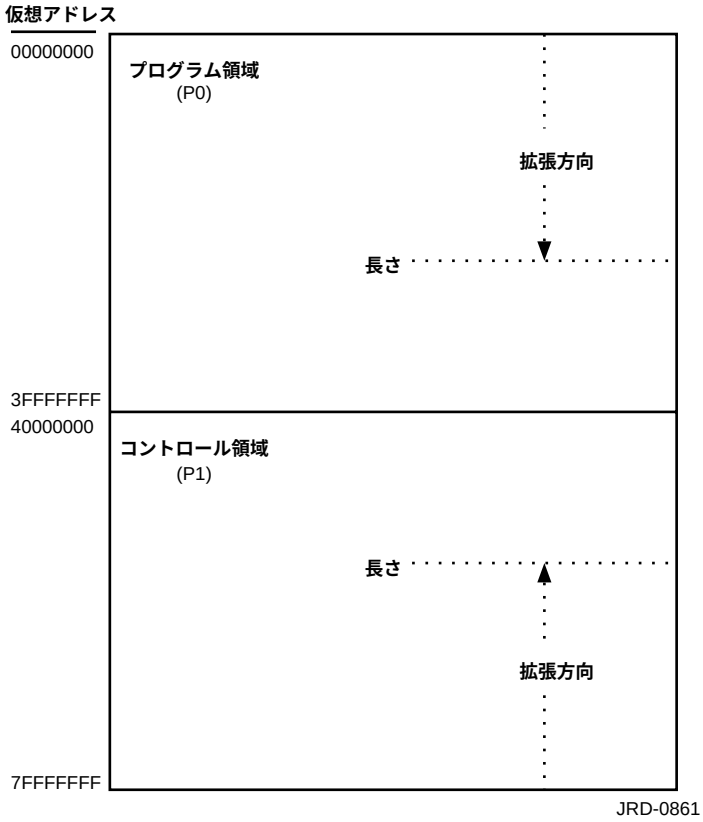
Alpha アーキテクチャでは，VAX アーキテクチャと同じ仮想アドレス空間レイアウトを定義しており，VAX システムの場合と同じ方向に P0 領域と P1 領域を拡大できます。図 5–1 はこのレイアウトを示しています。

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 Alpha 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。

- Alpha 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) は 512 バイト単位で解釈されます。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。

図 5–1 仮想アドレスのレイアウト



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 Alpha 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の違いは，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，$EXPREG システム・サービスが戻すメモリ・サイズは， Alpha アーキテクチャを実現した各システムで異なる可能性があるため，システムが割り当てた正確なメモリ境界を確認しておくことが適切でしょう。正確なメモリ境界を確認するには，$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 Alpha 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，Alpha アーキテクチャの特定の動作のページ・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];

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

}
```

この後の各項目は，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する必要があるかもしれません。

- VAX ページ境界にアラインするために，inadr 引数に指定したアドレスを 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。 Alpha システムでは，$CRETVA システム・サービスが先頭アドレスを CPU 固有のページ境界で切り捨てますが，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として指定する再割り当てのサイズは， Alpha システムの方が 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・サイズに切り上げられるからです。この結果，隣接データが破壊される可能性があり，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，次の操作を実行してください。

- 可能性のあるすべてのページ・サイズに対して，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境界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]);
```

(次ページに続く)

例 5–2 (続き) 既存のアドレス空間でのメモリの割り当て

```
    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$DELTVA システム・サービスを呼び出す場合，$DELTVA システム・サービスに対して inadr 引数として，retadr 引数に戻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戻された値) をアプリケーションで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設定する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) を使用すれば，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 (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) 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
- 仮想アドレス空間の中で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，おもに$CRMPSC システム・サービスと$MGBLSC システム・サービスに対する次の引数によって決定されます。

- 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。$CRMPSC システム・サービスはこの引数を次の方法で解釈します。
  - inadr 引数に指定した 2 つのアドレスがどちらも同じであり， SEC$M_EXPREG ビットが flags 引数でセットされている場合には，システム・サービス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を割り当てますが，指定された位置は使用しません。
  - 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$M_EXPREG フラグがセットされていない場合には，指定した位置を先頭アドレスとして 1 ページがマッピングされます ($CRMPSC システム・サービスのこの操作モードは Alpha 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，システム・サービスは指定された境界を使用して，セクションをメモリにマッピングします。
- 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。ページの残された空間をアプリケーションで使用すべきではありません。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からです。

5.3.1 拡張した仮想アドレス空間へのマッピング

アプリケーションで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 Alpha 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からです。このように， 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， Alpha 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できますが，retadr 引数をアプリケーションで指定していない場合には，この引数を指定することにより，呼び出しによってマッピングされたメモリの正確な境界を判断するようにしてください。

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しい説明は，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**

```
#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;
```

(**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 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;

/**********  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する場合には，Alpha システムの$CRMPSC および$MGBLSC システム・サービスで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。

Alpha 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 Alpha アーキテクチャの各実装ごとに異なるため，アプリケーションでセクション・ファイルをマッピングするために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) を戻します。

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，同時に，flags 引数の SEC$M_EXPREG ビットがセットされていない場合には，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 ビットをセットし，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しい説明は，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しい説明は，第 5.3.3 項を参照してください。

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する場合には，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，Alpha システムでは$CRMPSC および$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からです。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なければなりません。 VAX システムでは，$CRMPSC および$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。 Alpha 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， Alpha 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) を戻します。
- retadr 引数に戻されるアドレスは，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異なります。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すべきではありません。retadr 引数に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。この場合，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。 Alpha 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。詳しくは，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がマッピングされるように，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。 Alpha 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，隣接するデータが重ね書きされて破壊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。 Alpha でこの方針に従うためには，アプリケーションでバッファのサイズを 64K バイト単位で定義してください。

  セクションがマップされるときに，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，リンカにバッファを独立したイメージ・セクションとして指定することです (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，隣接するデータはつぎのページ境界から配置します。このためバッファを独自のイメージ・セクションに独立されることによって，隣接データがマッピング操作によって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・ファイルを独自のイメージ・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 SOLITARY プログラム・セクション属性を設定する必要があります (詳しくは『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を，コンパイラが定義したバッファを別のプログラム・セクションに置くことを確認するために，使用する必要があるかもしれません。詳しくは，コンパイラの解説書を参照してください。

- $CRMPSCK と $MGBLSC システム・サービスの値として指定する先頭または末尾アドレスが， CPU 固有ページの先頭または末尾アドレスととも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，システム・サービスはアドレス

がページ境界にそろうように調整します。 Alpha システムでは，システム・サービスは，ユーザが指定したアドレスをページ境界にそろうように調整しません。

セクションを独自のイメージ・セクションに分離する場合には， SOLITARY プログラム・セクション属性を使用して，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，リンカが省略時の設定によりイメージ・セクションをページ境界にアラインするからです。

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たことを確認するためには，アプリケーションを実行してい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。また，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，inadr引数内でシステム・サービスに渡すこともできます。

システムがマップした使用可能なメモリ量を判断するためには，retadr 引数を指定してください。たとえアプリケーションがページの一部分しか使用しないとしても，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 引数で指定された末尾アドレスは使用できるメモリの上限を示します (Alpha システムでアプリケーションが$CRMPSC システム・サービスに relpag 引数を指定する場合，必ず retadr 引数を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，これは 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];
```

(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

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

(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

```
    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 Alpha 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 Alpha のページ境界にアラインされることと，指定する末尾アドレスが Alpha ページの末尾にアラインされることを確認しなければなりません。
- Alpha システムの大きいページをマッピングするときに，隣接データの上に重ね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・セクションを独自のイメージ・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は 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言語の種類により異なります。セクションが独自のプログラム・セクションにあることをコンパイラの解説書で確認してください)。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 固有のページ境界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・サイズを入手し，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・サイズを判断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 Alpha 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 (pagcnt) 引数の値とセクション・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定されます。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 引数に戻された値を使用します。

例 5–5 は， Alpha 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 Alpha 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;
```

(次ページに続く)

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

```
/********  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;

/**********  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] );
```

(次ページに続く)

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

```
    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 );
    }
}
```

次のリストの各項目は，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 アイテム・コードの定義が登録されています。
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 Alpha システムで実行するまで判断できないため， DEC C for OpenVMS Alpha コンパイラは，_align(page) が指定されているときに，データを Alpha の最大ページ・サイズ (64KB) にアラインします。
3. この構造は，実行時にページ・サイズを入手するために使用される項目リストを定義します。
4. この変数には，戻されたページ・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は，実行時にページ・サイズが入手されます。
6. バッファの末尾アドレスは，戻されたページ・サイズ値から 1 を減算することにより指定されます。

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。

$CRMPSC システム・サービスは互換性を維持するために， VAX システムと Alpha システムの両方のシステムにおいて，relpag 引数の値を 512 バイト単位で解釈します。しかし， Alpha システムの CPU 固有のページ・サイズは 512 バイトより大きいため，relpag 引数にオフセットとして指定する値はおそらく CPU 固有のページ境界にアラインされません。$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， Alpha 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・アドレスを含む CPU 固有のページの先頭から開始され，オフセットによって指定されるアドレスから正確に開始されるわけではありません。

注意

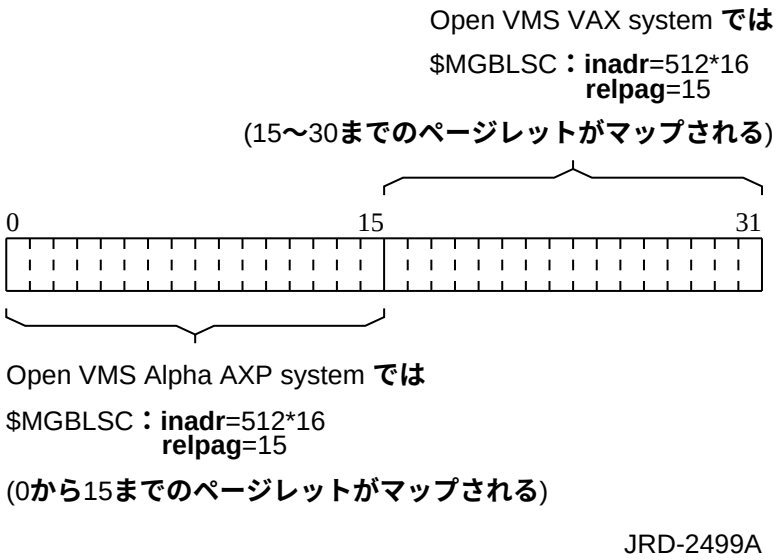
ルーチンは，オフセットによって指定されるアドレスを含む CPU 固有のページの先頭からマッピングを開始しますが，retadr 引数に戻される先頭アドレスはオフセットによって指定されたアドレスであり，実際にマッピングが開始されたアドレスではありません。

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， Alpha 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 引数に指定されるアドレス範囲のサイズを拡大する必要があります。指定されるアドレス範囲が小さすぎる場合には，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・ファイルの先頭アドレスから開始されるからです。

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 を指定できます。これと同じマッピングを Alpha 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 Alpha 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。 Alpha システムでは，$CRMPSC システム・サービスはセクション・ファイルのマッピングを CPU 固有のページ境界から開始するため， 16 番目から 30 番目までのブロックを正しくマッピングできません。マッピングを正しく実行するには， Alpha システムで$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。このように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 はこの状況を示しています。

図 5–2 オフセットによるマッピングに対してアドレス範囲が与える影響



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，次の公式を使用すると便利です。この公式は，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。

$$\frac{(number\_of\_pagelets\_to\_map + (2 * pagelets\_per\_page) - 2)}{pagelets\_per\_page}$$

たとえば，この公式を使用すれば，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，pagelets_per_page 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，この公式は inadr 引数に指定するアドレス範囲が， CPU 固有のページの 2 ページに対応し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

Alpha 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・サイズを確認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;

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・セットとしてロックする操作

$LKWSET システム・サービスは，VAX システムでも Alpha システムでも同様に，アドレス範囲として inadr 引数に指定したページ範囲をワーキング・セットとしてロックします。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。

しかし，Alpha 命令は完全な仮想アドレスを指定できないため，Alpha イメージはプロシージャ記述子に対するポインタを通じて，間接的にプロシージャとデータを参照しなければなりません。プロシージャ記述子には，実際のコード・アドレスも含めて，プロシージャに関する情報が格納されます。プロシージャ記述子とデータに対するこれらのポインタは，**リンケージ・セクション**と呼ぶ新しいプログラム・セクションに収集されます。

対処方法

Alpha システムでは，単にコード・セクションをメモリにロックするだけでは性能を向上するのに不十分です。関連するリンケージ・セクションもワーキング・セットとしてロックしなければなりません。

リンケージ・セクションをメモリ内でロックするには，リンケージ・セクションの先頭アドレスと末尾アドレスを判断し，$LKWSET システム・サービスを呼び出すときに，inadr 引数の値としてこれらのアドレスを指定しなければなりません。

6

共有データの整合性の維持

この章では，共有データの整合性を維持するの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， Alpha システムで共有データの整合性を保護するために，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を妨害する可能性があり，その結果，データは予測できない状態になる可能性があります。

VAX システムでは，必要とされる同期のレベルは実行スレッドの関係に応じて異なります。次に示すいくつかの場合は，同期について考慮しなければなりません。

- 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可能性があります。

  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・システムでは AST を使用して，指定したユーザ・バッファへのバッファを介した読み込み操作を終了します。

- 1 つのプロセッサ上で複数のプロセスが実行されているとき，ある共通のグローバル・セクションにアクセスする，これらのプロセスによるスレッド
- 複数のプロセッサ上で複数のプロセスが**並行**に実行されているとき，ある共通のグローバル・セクションにアクセスする，これらのプロセスによるスレッド

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常に，ロックやセマフォ，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，共有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，明示的に共有データを保護し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)。

たとえば，複数のスレッドが重要なコード領域をアクセスするときに，アクセスの同期をとるためにセマフォ変数を使用するアプリケーションは，不可分な操作によってインクリメントされるセマフォを必要とします。 VAX システムでは，このような不可分な操作は VAX アーキテクチャによって保証されています。

Alpha アーキテクチャでは， VAX アーキテクチャと同じように同期がとられるという保証はありません。 Alpha システムでは，このセマフォへのアクセスや，複数の実行スレッドがアクセスできるデータへのアクセスは，明示的に同期をとらなければなりません。 VAX システムの場合と同じ保護を実現するために使用できる Alpha アーキテクチャの機能については，第 6.1.2 項を参照してください。

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を保証していません)。

- **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) としてリード・モディファイ・ライト (読み込み/変更/書き込み) 操作を実行できます。しかし，Alpha アーキテクチャでは，このような命令はサポートされません。 VAX システムで不可分な操作として実行できる操作は， Alpha システムでは一連の命令として実行しなければならず，その途中で割り込みが発生する可能性があり，その結果，データは予測できない状態になる可能性があります。

  たとえば， VAX Increment Long (INCL) 命令は指定されたロングワードの内容をフェッチし，その値をインクリメントし，値を元のロングワードに格納します。これらの操作は割り込み不可能な方法で実行されます。しかし， Alpha システムでは，各ステップを別々の命令で実行しなければなりません。

  VAX システムとの互換性を維持するために， Alpha アーキテクチャでは，リード/ライト (読み込み/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，1 組の命令を定義しています。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使用した場合に， Alpha システムのコンパイラがどのような操作を実行するかについては，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，インクリメント操作 (X = X + 1) のような不可分な命令が記述されている場合でも，最適化のためにこのような命令を実現しない可能性があります。

- **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ます (VAX アーキテクチャは他のサイズのデータも処理できるような命令をサポートします)。 Alpha アーキテクチャでは，ロングワード・サイズとクォドワード・サイズのデータを処理する命令のみをサポートします。 Alpha システムでバイト・サイズおよびワード・サイズのデータを処理するには，複数の命令が必要です。つまり，バイトまたはワードを格納したロングワードまたはクォドワードをフェッチし，不要なバイトをマスクしなければならず，処理の対象となるバイトまたはワードを操作した後，ロングワードまたはクォドワード全体を格納しなければなりません。このシーケンスは割り込み可能であるため，バイト・データとワード・データに対する操作は， VAX システムでは不可分な操作ですが，Alpha システムでは不可分な操作ではありません。

  このようにメモリ・アクセスの粒度が変更された結果，どのタイプのデータを共有するかについても考慮しなければなりません。 VAX システムでは，共有されるバイト・サイズまたはワード・サイズのデータは個別に操作できます。 Alpha システムでは，バイト・サイズまたはワード・サイズの項目を含むロングワードまたはクォドワード全体を操作しなければなりません。したがって，明示的に共有されるデータに隣接しているという理由だけで，隣接データも暗黙のうちに共有されることになります。

  コンパイラは第 6.1.2 項で説明する Alpha 命令を使用して，バイト・サイズおよびワード・サイズのデータの整合性を保証します。

- **読み込み/書き込みの順序**—VAX ユニプロセッサおよびマルチプロセッサ・システムでは，一連の書き込み操作と読み込み操作は，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。 Alpha ユニプロセッサ・システムでも，読み込み操作と書き込み操作の順序は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，同期がとられているように見えます。しかし， Alpha 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書き込み操作を確認するには，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， Alpha 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，読み込み/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 Alpha アーキテクチャが提供する機能についての説明と，高級言語プログラムでこの機能を利用する際に， Alpha システムのコンパイラがどのような操作を実行するかについての説明は，第 6.3 節を参照してください。

6.1.2 Alpha の互換性機能

VAX アーキテクチャの不可分な機能との互換性を維持するために，Alpha アーキテクチャでは 2 つのメカニズムを定義しています。

- Load-locked/Store-conditional **命令**—Alpha 命令セットには，Load-locked(LDxL) と Store-conditional (STxC) という名前の 1 組の命令があり，ロック・ビットをセットおよびテストすることにより，不可分なロード/ストア操作を可能にします。これらの命令についての詳しい説明は，『Alpha Architecture Reference Manual』を参照してください。

  Load-locked/Store-conditional 命令を使用することにより，Alpha システムのコンパイラはバイト・サイズおよびワード・サイズのデータに対して不可分なアクセスを実現できます。さらに，Alpha システムのコンパイラでは，volatile 属性によって宣言されたバイト・サイズおよびワード・サイズのデータをアクセスするときに，Load-locked/Store-conditional 命令を生成できます (Alpha アーキテクチャでは，ロングワード・サイズとクォドワード・サイズのデータの不可分なロード/ストア操作は準備されています)。

- **メモリ・バリア**—Alpha 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込み/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・バリアと呼ばれ，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。

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有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。共有データを検出する場合には，意図的に共有されるデータだけでなく，暗黙のうちに共有されるデータも検出しなければなりません。暗黙のうちに共有されるデータとは，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有されるデータです。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有されるデータです。

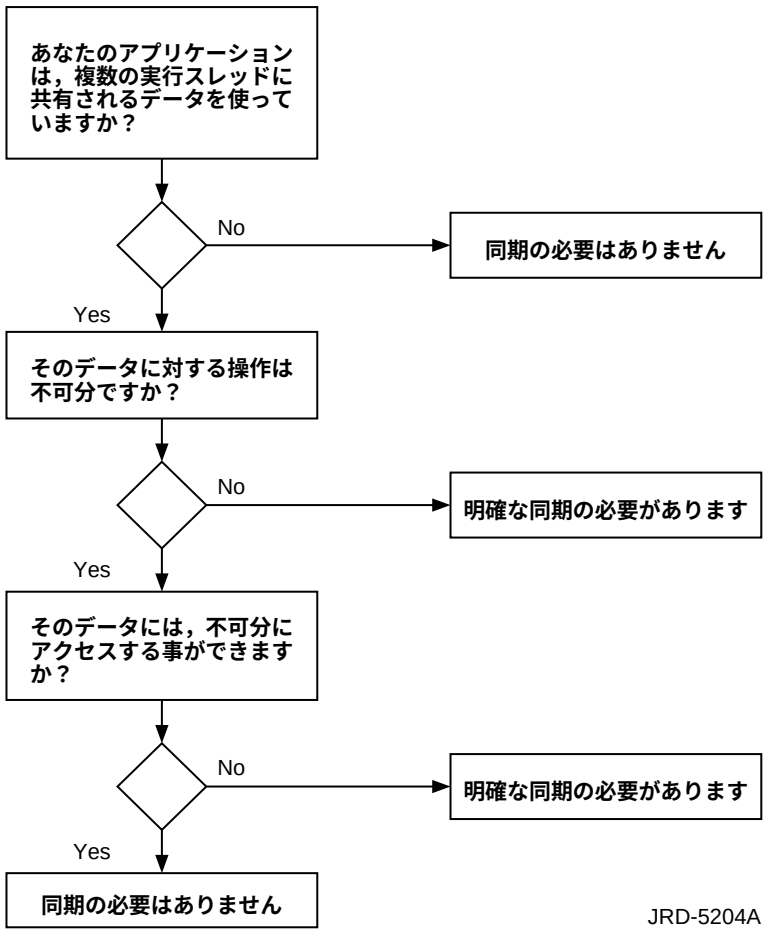
Alpha システムのコンパイラはある状況では，省略時の設定でクォドワード命令を使用するため，共有データが格納されているクォドワードと同じクォドワード内のすべてのデータは暗黙のうちに共有される可能性があります。たとえば，コンパイラは自然な境界にアラインされていないデータをアクセスするためにクォドワード命令を使用します (アドレスがデータ・サイズで割り切れる場合には，データは自然にアラインされています。詳しくは第 7 章を参照してください。コンパイラは省略時の設定により，宣言されたデータを自然な境界にアラインします)。

データ・アクセスを調べる場合には，別のスレッドが処理中の状態のデータを確認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，共有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。

- 共有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 はこの判断を下す処理を示しています。

図 6–1 同期に関する判断



6.2.1 明示的に共有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 flag を最大値に設定する AST 割り込みをキューに登録し，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

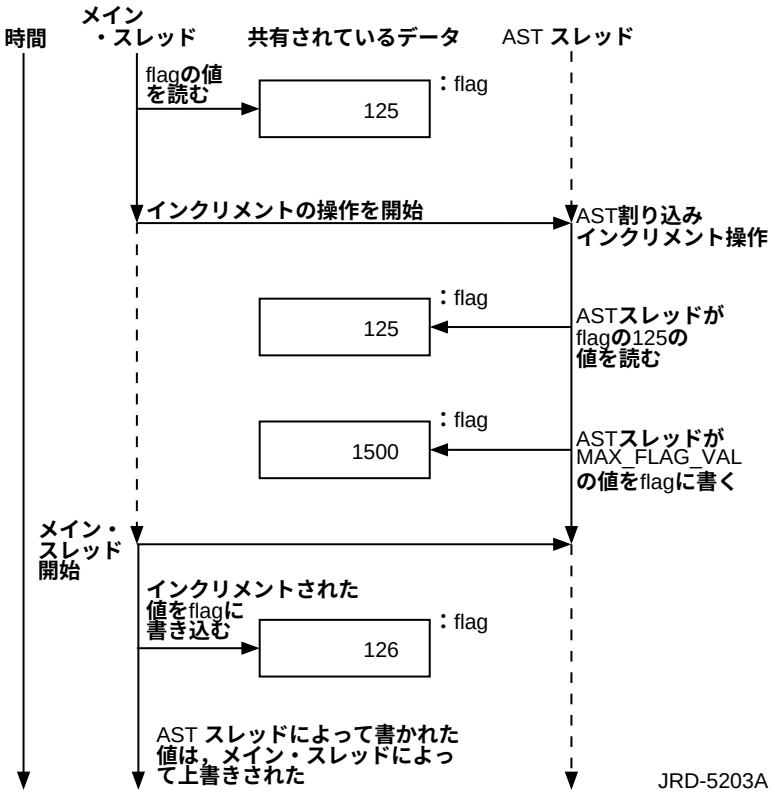
```
    flag = 0;   /* loop until flag = MAX_FLAG_VAL */
    while( flag < MAX_FLAG_VAL )
    {
          printf("main thread processing (flag = %d)\n",flag);
          flag++;
    }
    printf("Done\n");
}

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有されます。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。

Alpha システムでは，このプログラムは常に VAX システムと同じよう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，新しい値をメモリに格納する前に，メイン実行スレッドがインクリメント操作の途中で 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 スレッドによって割り込みが発生する可能性がもっと高くなります)。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，変数の値を最大値に設定します。しかし，制御がメイン・スレッドに戻された後，インクリメント操作は終了し， 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。

- データがアクセスされている間，$SETAST システム・サービスを使用して AST の実行要求を禁止し，アクセスが終了した後で実行要求を可能にします。
- コンパイラ・メカニズムを使用して，データを明示的に保護してください。たとえば， DEC C for OpenVMS Alpha システムは不可分性に関する組み込み機能をサポートします。さらに，このデータへのアクセスの同期をとるために他のメカニズムを使用できます。たとえば，$ENQ システム・サービスを使用したり (マルチプロセッサ・システムで実行され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)，また， 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かわりに， DEC C for OpenVMS Alpha システムがサポートする不可分性に関する組み込み機能 (__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。

共有変数を不可分性に関する組み込み機能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。Alpha 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。しかし，バイト粒度と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可能性があります。

例 6–2 は，例 6–1 に示したプログラムで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;
}
```

次のリストの各項目は例 6–2 に示した番号に対応しています。

1 DEC C for OpenVMS Alpha システムの不可分性に関する組み込み機能を使用するには，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，変数 flag はロングワードとして宣言されているため，不可分なアクセスが可能です (不可分性に関する組み込み機能を使用するには，変数をこのように宣言しなければなりません)。

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

6.2.2 無意識に共有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。しかし，Alpha システムでは，暗黙のうちに共有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す。この例では，2 つの変数はロングワードまたはクォドワードの境界内で物理的に隣接しています。VAX システムでは，各変数は個別に処理できます。Alpha 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，不可分な読み込み操作と書き込み操作がサポートされるため，処理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・アクセス粒度の変更についての詳しい説明は，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てみましょう。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データ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，次の処理を実行してください。

- 共有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，Alpha 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する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をアクセスするためにクォドワード命令を使用します。

  データ構造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 Alpha コンパイラは省略時の設定により，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妨害を受けずに，データ構造の各フラグ変数を確実に変更できるようにするには， 64 ビットの値となるように変数の宣言を変更します。 DEC C を使えば，double データ型を使用できます。次の例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機能や volatile 属性などのような，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，またはコンパイラのインターロック・キュー操作を使用することにより同期をとることができます。

6.3 読み込み/書き込み操作の同期

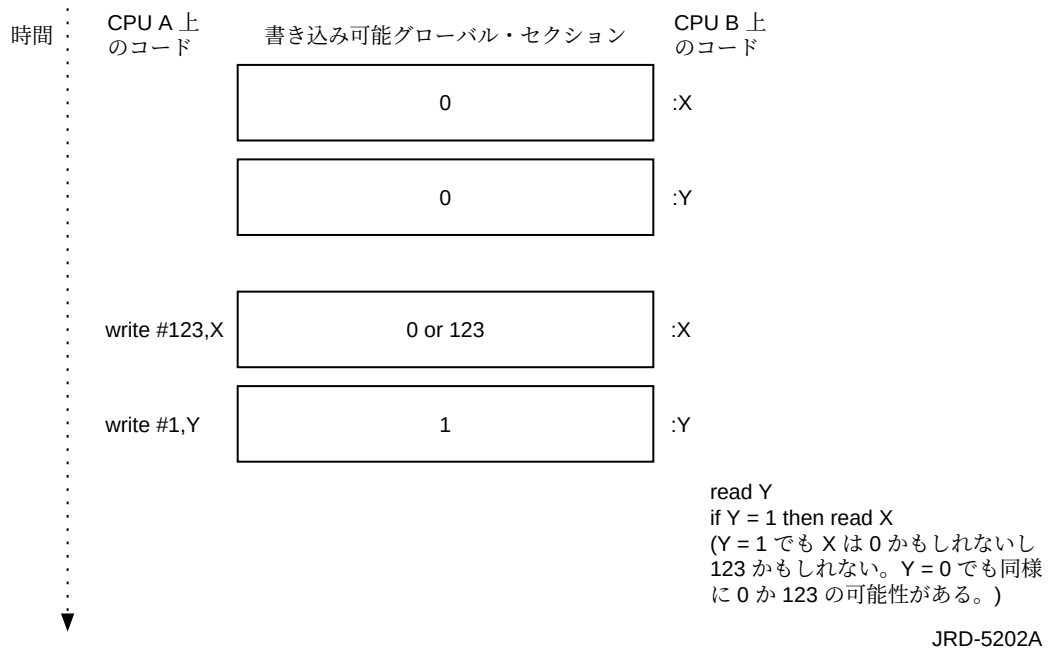
VAX マルチプロセシング・システムは従来，マルチプロセシング・システム内の 1 つのプロセッサが複数のデータを書き込むときに，これらのデータが書き込まれた順序と同じ順序で，他のすべてのプロセッサから確認できるように設計されていました。たとえば， CPU A がデータ・バッファを書き込み (図 6–3 で X によって表現されるもの)，その後でフラグを書き込んだ場合 (図 6–3 で Y によって表現されるもの)，CPU B はフラグの値を確認することにより，データ・バッファが変更されたことを判断できます。

Alpha システムでは，メモリ・サブシステム全体の性能を向上するために，メモリとの間の読み込み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込み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を，前もって判断できません。つまり， CPU A によって実行される書き込み操作は，実際に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 からの読み込み操作を要求し， Y の新しい値を確認し， X の読み込み操作を開始します。 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 は前の値を読み込みます。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確認する可能性があり，その結果，誤ったメモリの内容を使用してしまいます。

図 6–3 Alpha システムでの読み込み/書き込み操作の順序



対処方法

並列に実行され，読み込み/書き込みの順序に依存するプログラムは，Alpha 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，次の方法を使用してください。

- 終了の順序が重要な，すべての読み込み命令と書き込み命令の前後では，Alpha メモリ・バリア命令 (MB) を使用してください。たとえば， DEC C for

OpenVMS Alpha システムコンパイラは組み込み機能として，メモリ・バリア命令をサポートします。

- LIB$ランタイム・ライブラリで提供される VAX インターロック命令のメモリ・インターロックを使用するように，アプリケーションの設計を変更してください。
- ロックによってデータを保護するために，$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， Alpha システムでトランスレートされた VAX イメージを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可能性があります(/PRESERVE 修飾子の指定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)。

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妨害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。

読み込み/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，アプリケーションが使用する VAX アーキテクチャに依存したデータを確認する方法について説明します。この章ではまた，データ型の選択が Alpha 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。各高級言語は，ターゲット・プラットホームでサポートされるデータ型に対し，各言語の持つデータ型をマッピングします。この理由から， VAX システムのアプリケーションは，データ宣言をまったく変更せずに Alpha 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。 Alpha アーキテクチャは大部分の VAX データ型をサポートします。しかし，サポートされないデータ型もあります。 Alpha システムで，このようなサポートされないデータ型のサイズやビット・フォーマット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 Alpha システムで異なる可能性があります。たとえば VAX システムでは，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することができました。 Alpha システムでは，逆に必要なメモリが拡大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， Alpha 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なデータ型を示しています (デ

ータ型のフォーマットについての詳しい説明は，『Alpha Architecture Reference Manual』を参照してください)。

表 7–1　VAX と Alpha のネイティブなデータ型の比較

VAX データ型	Alpha 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
D 浮動小数点 (56 ビットの精度)	D 浮動小数点 (53 ビットの精度)
G 浮動小数点	G 浮動小数点
H 浮動小数点	–
–	S 浮動小数点 (IEEE)
–	T 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	–

対処方法

アプリケーションが VAX データ型のフォーマットやサイズに依存していない限り，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， Alpha システムのコンパイラは，そのデータ型を VAX システムと同じやり方で，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 Alpha 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対応する Alpha データ型にマッピングします (Alpha 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な Alpha データ型に対しどのような方法でマッピングするかについての詳しい説明は，第 11 章とコンパイラの解説書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。

- D **浮動小数点データ**— 大部分の Alpha システムのコンパイラは省略時の設定で，倍精度の浮動小数点データ型を VAX のネイティブな G 浮動小数点データ型にマッピンクします。これは， Alpha アーキテクチャで VAX D 浮動小数点データ型をサポートしないからです。 OpenVMS VAX コンパイラは倍精度の浮動小数点データ型を D 浮動小数点データ型にマッピングします。たとえば，VAX Cでは double

データ型を D 浮動小数点データ型にマッピングし， DEC C for OpenVMS Alpha システムでは double データ型を G 浮動小数点データ型にマッピングします。

ほとんどのアプリケーションにとって，この変更はまったく影響ありません。しかし， G 浮動小数点データ型から戻される値 (小数点以下 15 桁の有効桁数) は D 浮動小数点データ型から戻される値 (小数点以下 16 桁の有効桁数) より，少し精度が低くなります。

OpenVMS ランタイム・ライブラリは変換ルーチン (CVT$CONVERT_FLOAT) をサポートし，これらのルーチンを使用すれば，浮動小数点データを 1 つのフォーマットから別のフォーマットに変換できます。たとえば，このルーチンを使用すれば， D 浮動小数点形式のデータを IEEE 形式に変換したり，その逆に変換することができます。また， Alpha アーキテクチャは IEEE 倍精度浮動小数点形式 (T 浮動小数点) もサポートします。

DEC C for OpenVMS Alpha システムは， long float データ型を使用する宣言を見つけると警告メッセージを出します。 VAX システムでは， long float データ型は double と同意語です。 Alpha システムでは， DEC C コンパイラがVAX Cモードで使用されていても， long float データ型はサポートされません。

- **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を確認してください。 Alpha システムでは，アドレスは 64 ビットです。

  たとえば，VAX Cでは，一部のプログラムでこのような仮定をしています。
  例 7–1 を参照してください。

例 7–1 VAX Cコードでのデータ型に関する仮定

```
typedef struct {
       char     small;
       short   medium;
       long     large;
       } MYSTRUCT ;
main()
{
    int        a1;
    long       b1;
    MYSTRUCT   c1;
1   a1 = &c1;
2   b1 = &c1;
3   a1->small = 1;
    b1->small = 2;
}
```

次のリストの各項目は例 7–1 に示した番号に対応しています。

1 この例では，変数a1に構造体のアドレスを割り当てます。この変数は int データ型として宣言されます。

2 この例では，変数b1に対して構造体のアドレスを割り当てます。この変数はlong データ型として宣言されます。

3 この例では， int データ型と long データ型に割り当てた変数を使用することにより，構造体内の最初のフィールドをアクセスします。

この例を Alpha システムに移行するには，次に示すように，a1とb1の宣言を，データ構造体 (MYSTRUCT) に対するポインタに変更しなければなりません。

```
MYSTRUCT *a1,*b2;
```

7.3 データ型の選択に関する仮定の確認

アプリケーションが Alpha システムで再コンパイルし，正しく実行できる場合でも，データ型の選択のために OpenVMS Alpha アーキテクチャの特徴を完全に利用できていない可能性があります。特に，データ型の選択は Alpha 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が使用されます。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 データ型の変数を宣言します。 VAX システムでは，このようにすれば必要な記憶空間を節約でき，また， VAX アーキテクチャが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するので，効率を損いません。

Alpha システムでは，バイト・サイズとワード・サイズのデータを使用すると，ロングワード・サイズやクォドワード・サイズのデータを使用したときより多くのオーバーヘッドが発生します。これは， Alpha アーキテクチャでこれらの小さいサイズのデータ型を操作できる命令がサポートされないからです。バイトやワードを参照する操作は， VAX システムでは 1 つの命令として実現されますが， Alpha システムでは，対象となるバイトまたはワードを格納したロングワードのフェッチ，対象となるバイト，ワードの操作，ロングワード全体の格納といった一連の命令として実現されます。頻繁に参照されるデータの場合には， Alpha システムでこれらの追加された命令によって，アプリケーションのサイズが大幅に大きくなる可能性があります。

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ます)。

VAX システムでも Alpha システムでも，アラインされていないデータをアクセスすると，アラインされているデータをアクセスする場合より多くのオーバー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を使用したときの性能に対する影響を最低限に抑えるためにマイクロコードを使用しています。Alpha 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，システムの PALcode によって処理しなければならなくなります。フォルトを処理している間，命令パイプラインは停止しなければなりません。したがって，アラインされていないデータを参照したときの性能の低下は， Alpha システムではきわめて大きくなります。

Alpha 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・サイズと性能に与える影響を考慮した後，バイトとワードのアクセスのために必要な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ことを考慮しなければなりません。しかし，データ宣言の変更を考慮する前に，次の要素を考慮してください。

- **アクセスの頻度と繰り返し回数**— 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，参照時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，大量のバイト・サイズまたはワード・サイズのデータが存在する場合 (たとえば，データ構造体に同じ属性のデータが何度も繰り返される場合)，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，これは，各参照で必要となる追加命令の問題より大きな問題になる可能性があります。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，そのデータを数千回繰り返すと必要な仮想メモリは大幅に拡大します。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イメージまたはネイティブな VAX 構成イメージと共有する場合には，他の構成要素がデータのバイナリ・レイアウトに依存するため，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) は生成する

コード内に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，次のガイドラインに従ってください。

- 頻繁に参照されるが，何度も繰り返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，何度も繰り返されるデータの場合には，変更は望ましくありません。
- 頻繁に参照され，何度も繰り返されるデータの場合には，コード・サイズと性能を注意深く調べた後，判断を下さなければなりません。
- 静的データの場合には，常にバイトのかわりにロングワードを使用してください。この場合， 3 バイトのメモリが必要になりますが，ロングワードに変更しなければ， 1 回の参照で 3 つの命令 (各命令はロングワードで表現されます) が余分に必要となります。
- Alpha 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。多くの Alpha システムのコンパイラ (Digital Fortran など) は/WARNING=ALIGNMENT 修飾子をサポートします。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を確認します。
- 実行時解析ツールである 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。詳しくは，『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， Alpha システムのコンパイラが準備している自然なアラインメントを利用してください。 Alpha システムでは，コンパイラは省略時設定で可能な限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。

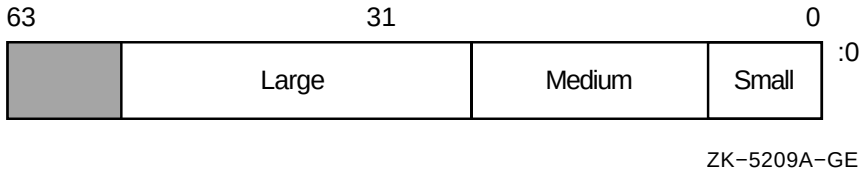
  Alpha システムのコンパイラは， VAX システムと同様のバイト・アラインメントの使用を要求することを許可する修飾子や言語プログラムをサポートしています。たとえば， DEC C for OpenVMS Alpha システムのコンパイラは /NOMEMBER_ALIGNMENT 修飾子をサポートし，また，それに対応した，データ・アラインメントの制御を許可するプラグマもサポートします。詳しくは，DEC C のコンパイラ解説書を参照してください。

例 7–1 で定義したデータ構造体は，これらのデータ型の選択に関する問題を示しています。 mystruct という構造体定義は，次に示すようにバイト・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 ;
```

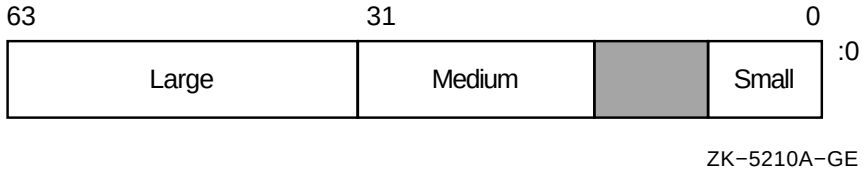
VAX Cを使用してコンパイルした場合には，この構造体は図 7–1 に示すようにメモリにレイアウトされます。

図 7–1 VAX Cの使用による mystruct のアラインメント



DEC C for OpenVMS Alpha システムの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ッドが挿入されます。最初のフィールド (small) の後に 1 バイトのパッド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 DEC C for OpenVMS Alpha システムの使用による mystruct のアラインメント



データ構造体の中でバイト・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繰り返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データ構造体を再定義することを考慮してください。しかし，フィールドが頻繁に参照されない場合や，データ構造体が何度も繰り返される場合には，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 Alpha 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 Alpha システムでも正しく機能します。特に， FORTRAN の END や ERR， IOSTAT 指定子など，アプリケーション開発において，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， Alpha 条件処理機能とそれに対応する VAX 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・アレイの形式を変更する場合
- システムから戻された条件コードを変更する場合
- アプリケーション内の条件処理に関連する他の作業を実現する方法を変更する場合。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場合など

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 Alpha の実行時ライブラリ (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 Alpha 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 Alpha 言語では， LIB$ESTABLISH の呼び出しが特別な方法で取り扱われ，実際に RTL ルーチンを呼び出さずに適切なコードを生成します。次の言語は，対応する VAX 言語と互換性のある方法で LIB$ESTABLISH をサポートします。

- DEC C と DEC C++

  OpenVMS Alpha システム用の DEC C と DEC C++ は，LIB$ESTABLISH を組み込み関数として取り扱いますが，OpenVMS VAX または OpenVMS Alpha システムで LIB$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 Alpha システム用の DEC C と DEC C++ で提供される組み込み関数です)。

- Digital Fortran

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

- DEC Pascal

  DEC 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。

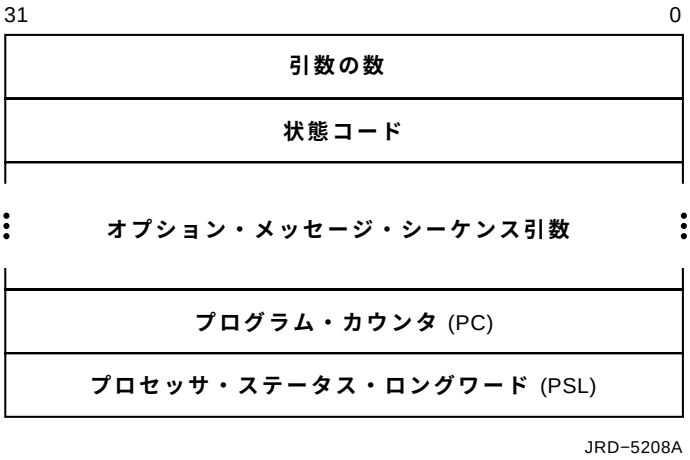
  MACRO–32 プログラムが 0(FP) のルーチン・アドレスを格納することにより，動的ハンドラを設定する場合には，これらのプログラムは，OpenVMS Alpha システムでコンパイルした場合も正しく動作します。しかし，CALL_ENTRY ルーチンの内部からでなければ，JSB (Jump to Subroutine) ルーチンの内部から条件ハンドラ・アドレスを設定することはできません。

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，Alpha 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戻すデータを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達し，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』に説明されています。Alpha システムでは，シグナル・アレイに戻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 Alpha システム上の 32 ビット・シグナル・アレイ

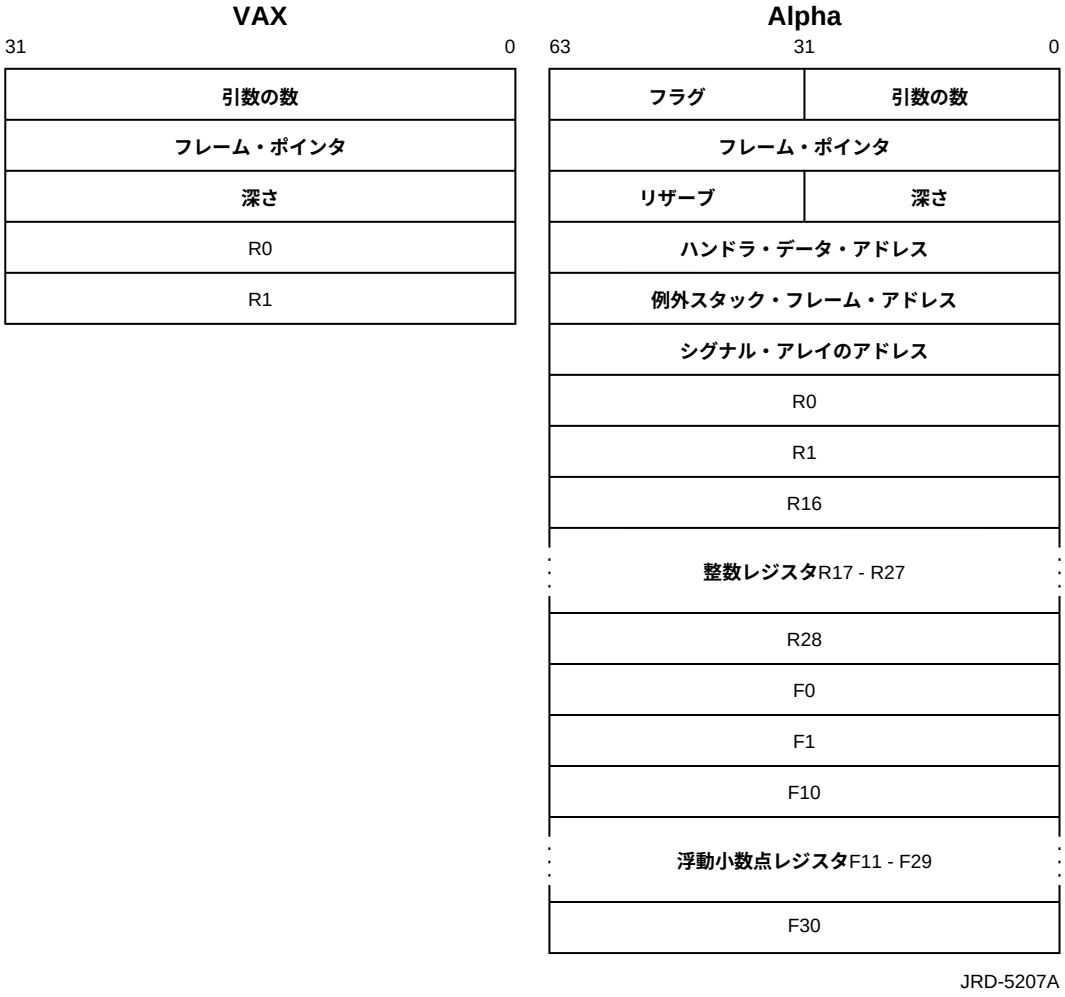


次の表はシグナル・アレイ内の各引数を示しています。

引数	説明
引数の数	Alpha 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	Alpha 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例外条件を一意に識別する。条件コードの形式は Alpha 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，Alpha システムは VAX システムで戻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 システムでは戻されない条件コードを定義している。 Alpha システムで戻すことができない 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	これらの引数は戻される例外に関する追加情報を提供し，これは各例外に応じて異なる。 VAX 例外に対す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。 Alpha 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 (Alpha システムでは， PC は 64 ビットの長さである)。
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記述する。 Alpha システムでは，この引数には Alpha の 64 ビットのプロセッサ・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。

Alpha 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戻されます。しかし，その形式は異なります。 Alpha システムで戻されるメカニズム・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， Alpha のレジスタは 64 ビットの長さであるため，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，クォドワード (64 ビット) で構成されます。図 8–2 は VAX システムと Alpha システムでのメカニズム・アレイの形式を比較しています。

図 8–2 VAX システムと Alpha システムでのメカニズム・アレイ



次の表はメカニズム・アレイ内の各引数を示しています。

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。 Alpha 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，引数カウント・クォドワードの数 (Alpha システムでは常に 43) を示すわけではない。
フラグ	Alpha 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。たとえば，ビット 0 がセットされている場合には，プロセスがすでに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)。
フレーム・ポインタ (FP)	VAX システムでも Alpha システムでも，この引数には条件ハンドラを設定したスタックの呼び出しフレームのアドレスが格納される。

引数	説明
深さ	VAX システムでも Alpha 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現する整数が格納される。
リザーブ	予約されている。
ハンドラ・データ・アドレス	Alpha 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	Alpha 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	Alpha システムでは，この引数にはシグナル・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)。
レジスタ	VAX システムでも Alpha 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。 Alpha 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，対応する浮動小数点レジスタも格納される。

対処方法

32 ビット・シグナル・アレイは Alpha 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ないでしょう。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かもしれません。特に，次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・フレームを巻き戻すために 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， Alpha システムで戻される depth 引数は VAX システムで戻される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$UNWIND システム・サービスに対して depth 引数のアドレスを指定することにより，例外処理ハンドラを設定したフレームまで巻き戻すアプリケーションや， SYS$UNWIND システム・サービスの省略時の 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巻き戻すアプリケーションは，今後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 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;
     }
}
```

次のリストの各項目は例 8–1 に示した番号に対応しています。

1 このルーチンは，システムがシグナル・アレイとメカニズム・アレイに戻すデータを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 つのデータ構造を使用して，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・ファイルに定義されています。

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，戻された条件コードと，整数オーバーフローを識別する条件コード (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 に定義されています)。

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戻します。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・アレイに戻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。次のプログラムの一文は例 8–1 から抜粋したものであり，条件処理ルーチンがランタイム・ライブラリ・ルーチン LIB$MATCH_COND を使用することにより，この作業をどのような方法で実現できるかを示しています。

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

このメカニズムは Alpha システムでも変更されていません。 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は Alpha システムでは意味がないでしょう。アーキテクチャが異なるため， VAX システムで戻されていた一部の例外条件は， Alpha 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， Alpha 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 Alpha システムでもサポートされます。さらに， Alpha アーキテクチャでは， VAX アーキテクチャでサポートされないいくつかの追加された例外を定義しています。

表 8–1 は， Alpha 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 Alpha 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する場合には，対応する Alpha 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 (Alpha システムでの算術演算例外のテストについての詳しい説明は，第 8.4.1 項を参照してください)。

注意

Alpha 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，これらの VAX 例外を戻すことができます。

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
Alpha システム固有の例外	
SS$_HPARITH – 高性能算術演算例外	VAX 算術演算例外はこの例外に変更された (第 8.4.1 項を参照)。
SS$_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$_ARTRES – 予備の算術演算トラップ	Alpha システムには対応する例外はない
SS$_COMPAT – 互換性フォルト	Alpha システムには対応する例外はない
SS$_DECOVF –10 進オーバーフロー[1]	SS$_HPARITH に変更された (第 8.4.1 項を参照)
SS$_FLTDIV –0 による浮動小数点除算 (トラップ)[1]	SS$_HPARITH に変更された (第 8.4.1 項を参照)
SS$_FLTDIV_F –0 による浮動小数点除算 (フォルト)	SS$_HPARITH に変更された (第 8.4.1 項を参照)
SS$_FLTOVF – 浮動小数点オーバーフロー (トラップ)[1]	SS$_HPARITH に変更された (第 8.4.1 項を参照)
SS$_FLTOVF_F – 浮動小数点オーバーフロー (フォルト)	SS$_HPARITH に変更された (第 8.4.1 項を参照)
SS$_FLTUND – 浮動小数点アンダーフロー (トラップ)[1]	SS$_HPARITH に変更された (第 8.4.1 項を参照)
SS$_FLTUND_F – 浮動小数点アンダーフロー (フォルト)	SS$_HPARITH に変更された (第 8.4.1 項を参照)
SS$_INTDIV –0 による整数除算[1]	SS$_HPARITH に変更された (第 8.4.1 項を参照)
SS$_INTOVF – 整数オーバーフロー[1]	SS$_HPARITH に変更された (第 8.4.1 項を参照)
SS$_TBIT – トレース・ペンディング	Alpha システムには対応する例外はない
SS$_OPCCUS – ユーザ用に確保されているオペコード	Alpha システムには対応する例外はない
SS$_RADMOD – 予備のアドレッシング・モード	Alpha システムには対応する例外はない
SS$_SUBRNG –INDEX 添字範囲チェック	Alpha システムには対応する例外はない

[1]Alpha 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 Alpha システムでの算術演算例外のテスト

VAX システムでは，アーキテクチャは算術演算例外が同期的に報告されるようにします。つまり，例外 (オーバーフローなど) の原因となった VAX 算術演算命令は，ただちに例外処理ハンドラを開始し，後続の命令は実行されません。例外ハンドラに報告されるプログラム・カウンタ (PC) は，例外の原因となった算術演算命令の PC です。このため，アプリケーション・プログラムは，たとえば，メイン・シーケンスを再開

し，例外の原因となった操作を同等の操作または別の操作によってエミュレートするか，置換することができます。

Alpha システムでは，算術演算例外は非同期的に報告されます。つまり，アーキテクチャの実現方法により，例外の原因となった命令より後の多くの命令 (分岐やジャンプも含む) を実行できます。これらの命令は，例外の原因となった命令が使用していたオペランドの上に重ね書きする可能性があるため，例外を解釈したり，修正するのに必要な情報が失われてしまいます。例外ハンドラに報告される PC は，例外の原因となった命令の PC ではなく，その後に実行された命令の PC です。例外がアプリケーションの例外ハンドラに報告される時点では，ハンドラは入力データを修正しており，命令を再起動することができない可能性があります。

このように，算術演算例外の報告方法が基本的に異なるため， Alpha システムでは，SS$_HPARITH という 1 つの条件コードを定義し，これによってすべての算術演算例外を示します。たとえば，整数オーバーフロー例外が発生したときに処理を実行する条件処理ルーチンがアプリケーションに含まれている場合， VAX システムでは，SS$_INTOVR 条件コードが例外処理ルーチンに渡されます。 Alpha システムでは，この例外は SS$_HPARITH という条件コードによって示されます。このため，アプリケーションの条件処理ルーチンは， Alpha 算術演算例外を対応する VAX 例外と誤って解釈することがありません。処理を行うアプリケーションが，アーキテクチャ固有である可能性があるため，このことは重要です。

図 8–3 は SS$_HPARITH 例外シグナル・アレイの形式を示しています。

図 8–3 SS$_HPARITH 例外シグナル・アレイ

このシグナル・アレイには， SS$_HPARITH 例外の固有の 3 つの引数が格納されます。それは**整数レジスタ・ライト・マスク**，**浮動小数点レジスタ・ライト・マスク**，および**例外サマリ**です。整数および浮動小数点レジスタ・ライト・マスクは，例外サマリのビットをセットした命令のターゲットであったレジスタを示します。マスク内

の各ビットはレジスタを表現します。例外サマリは最初の 7 ビットにフラグをセットすることにより，通知される例外のタイプ (1 つ以上) を示します。表 8–2 はこれらの各ビットがセットされているときの意味を示しています。

表 8–2 例外サマリ引数のフィールド

ビット	意味
0	ソフトウェアは正常終了した。
1	浮動小数点演算，変換，または比較操作に誤りがある。
2	浮動小数点除算で 0 による除算を実行しようとした。 0 による整数除算は報告されないので注意しなければならない。
3	浮動小数点演算または変換操作で宛先の指数部がオーバーフローした。
4	浮動小数点演算または変換操作で宛先の指数部がアンダーフローした。
5	浮動小数点演算または変換操作で正確な算術演算結果と異なる結果が報告された。
6	浮動小数点数値から整数への変換操作または整数算術演算で宛先の精度がオーバーフローした。

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 Alpha 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， Alpha システムで例外が非同期的に報告されることは問題になりません。これらの条件処理ルーチンは， SS$_HPARITH 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操作を再起動しようとする場合には，コードを変更するか，またはコンパイラ修飾子を使用することにより，算術演算例外が正確に報告されるようにしなければなりません。大部分の Alpha コンパイラは，互換性を保つために，例外の正確な報告を行うかどうかをプログラマがコンパイル時に指定するためのコンパイラ・オプションを持っています (DEC C: /IEEE_MODE， DEC Fortran: /IEEE_MODE または/SYNCHRONOUS_EXCEPTIONS)。

  ただし，これらの命令を指定すると，性能が低下する可能性があります。例外の正確な報告がアプリケーションの一部の操作でのみ必要な場合は，そのような操作を含んでいる部分のコンパイルだけにこのオプションを使用するようにしてください。詳細については，使用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イメージで算術演算例外が正確に報告されることを保証するには，イメージをトランスレートするときに VEST コマンド・ラインに/PRESERVE=FLOAT_EXCEPTIONS 修飾子を指定します。この修飾子を使用した場合には， VEST ユーティリティは，浮動小数点フォルトの原因となる各命令を実行した後，例外を報告できるようなコードを生成します。しかし，この修飾子を使用すると，トランスレートされたイメージの性能が低下する可能性があ

ります。VEST コマンド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。

注意

Alpha 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 例外条件を戻します。

8.4.2 データ・アラインメント・トラップのテスト

Alpha システムでは，アラインされていないアドレスをオペランドとして受け付ける Alpha 命令 (LDQ_U) を使わずに，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しい説明は，第 7 章を参照してください)。

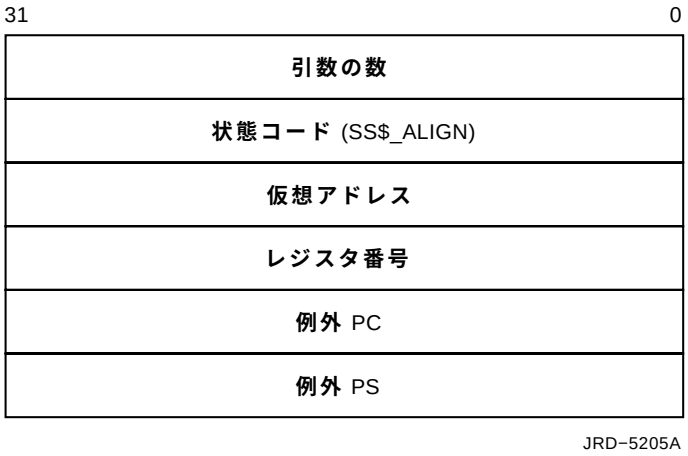
Alpha システムのコンパイラは通常，次の操作を実行することにより，アラインメント・フォルトの発生を防止します。

- 省略時の設定により，静的データを自然な境界にアラインします (この省略時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 例外によって戻されるシグナル・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，この例外条件を用います。このようにすれば，この段階で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他の操作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操作を実行するためのルーチンが準備されています。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，例外が通知されるときに実行される条件処理ルーチンを識別 (または設定) できます。

VAX アーキテクチャと Alpha 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，これらの多くの操作の実現方法は同じではありません。表 8–3 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，Alpha 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 8–3 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	Alpha 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$DEC_OVER –10 進オーバーフローの通知を許可または禁止する	サポートされない
LIB$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$FLT_UNDER – 浮動小数点アンダーフローの通知を許可または禁止する	サポートされない
LIB$INT_OVER – 整数オーバー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 Alpha 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 Alpha 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
  - 浮動小数点オーバーフロー

  省略時の設定により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$ESTABLISH を呼び出すことにより，条件処理ルーチンを指定する場合には，ソース・コードを変更する必要はありません。 Alpha システムの大部分のコンパイラは互換性を維持するために， LIB$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$ESTABLISH はこの変数を設定します。 LIB$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込み，どのルーチンを呼び出すかを判断します。

たとえば，例 8–2 に示したプログラムは FORTRAN で作成されており，条件コード SS$_INTOVF を指定することにより，整数オーバーフローをテストする条件処理ルーチンを指定するために， RTL ルーチン LIB$ESTABLISH を使用しています。 VAX システムでは，整数オーバー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ときに/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 Alpha システムで実行するには，条件コードを SS$_INTOVF から SS$_HPARITH に変更しなければなりません (オーバーフローのタイプはシグナル・アレイ内の例外サマリ引数を調べることにより判断できます。詳しくはコンパイラに関する解説書を参照してください)。 VAX システムの場合と同様に，オーバーフロー検出を可能にするためにはコンパイル・コマンド・ラインに/CHECK=OVERFLOW 修飾子を指定なければなりません。 DEC 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削除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) 2
        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。

2 Alpha システムでは，条件コードを SS$_INTOVF から SS$_HPARITH に変更しなければなりません。条件処理ルーチンは LIB$STOP ルーチンを呼び出すことにより，プログラムの実行を終了します。

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

```
$ FORTRAN/EXTEND_SOURCE/CHECK=OVERFLOW  HANDLER_EX.FOR
$ LINK  HANDLER_EX
$ RUN   HANDLER_EX
 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 Name   Module Name     Routine Name    Line Number  rel PC      abs PC
 INT_OVR_HAND INT_OVR_HANDLER HANDLER               1637 00000238    00020238
 DEC$FORRTL                                           0 000651E4    001991E4
----- above condition handler called with exception 00000504:
%SYSTEM-F-HPARITH, high performance arithmetic trap, Imask=00000001, Fmask=00000
000, summary=40, PC=000200E0, PS=0000001B
-SYSTEM-F-INTOVF, arithmetic trap, integer overflow at PC=000200E0, PS=0000001B
----- end of exception message
                                                     0 84FE9FFC    84FE9FFC
 INT_OVR_HAND INT_OVR_HANDLER INT_OVR_HANDLER       15 000000E0    000200E0
                                                     0 84EFD918    84EFD918
                                                     0 7FF23EE0    7FF23EE0
```

9
アプリケーションのトランスレート

この章では，Alpha システムで実行するために，VAX アプリケーションをトランスレートするときに使用するリソースについて説明します。

9.1 DECmigrate for OpenVMS Alpha

DECmigrate for OpenVMS Alpha は，ソース・コードを入手できないイメージをトランスレートするときに使用します。DECmigrate の VAX Environment Software Translator (VEST) コンポーネントは，VAX のバイナリ・イメージ・ファイルをネイティブな Alpha イメージにトランスレートします。トランスレートされたイメージは，Alpha コンピュータの Translated Image Environment (TIE) のもとで動作します (TIE は OpenVMS Alpha オペレーティング・システムで提供される共用可能イメージです)。トランスレーションでは，エミュレーションやインタプリテーションのもとで OpenVMS VAX イメージを実行することは行われません (ただし，特定の例外があります)。その代わり，新しい OpenVMS Alpha イメージには，元の OpenVMS VAX イメージの命令が実行していた操作と同じ操作を実行する Alpha 命令が含まれます。

トランスレートされたイメージは一般に，VAX コンピュータでオリジナル・イメージを実行するときと同じ速度で，Alpha コンピュータでも動作します。しかし，トランスレートされたイメージは，Alpha アーキテクチャのすべての利点を活用する最適化コンパイラを利用できません。したがって，トランスレートされたイメージは一般に，ネイティブな OpenVMS Alpha イメージの約 25 ～ 40 パーセントの速度で動作します。このように性能が低下する主な理由は，アラインされないデータと複雑な VAX 命令を広範囲にわたって使用しているためです。

DECmigrate によるトランスレーション・サポートは，OpenVMS VAX バージョン 5.5-2 に存在する言語機能，システム・サービス，実行時ライブラリ・エントリ・ポイントに制限されています。

DECmigrate のもう 1 つの機能は，イメージを分析して，Alpha コンピュータで互換性が維持されない部分を識別することです。このように互換性が維持されない部分は，その種類に応じて，問題を修正するためにコンパイラ修飾子を指定したり，コードを変更することができます。

イメージ・トランスレーションと VEST の詳細については，『DECmigrate for OpenVMS AXP Systems Translating Images』を参照してください。

9.2 DECmigrate: トランスレートされたイメージのサポート

DECmigrate バージョン 1.1A は，OpenVMS バージョン 6.1 以降を実行する Alpha システムで動作します。トランスレートされたイメージを実行するには，このバージョンまたはそれ以降のバージョンが必要です。トランスレートされたイメージは一般に，それ以降のバージョンと互換性を維持しますが，以前のバージョンとの互換性は維持しません。つまり，DECmigrate バージョン 1.1A を使用してトランスレートされたイメージは，OpenVMS バージョン 6.1 以降を稼動している Alpha システムのみで動作できますが，DECmigrate バージョン 1.0 を使用してトランスレートされたイメージは，OpenVMS Alpha バージョン 1.0 以降で動作できます。表 9–1 は，OpenVMS Alpha システムの各バージョンと，それをサポートする DECmigrate のバージョンを示しています。

表 9–1 OpenVMS Alpha の各バージョンでのトランスレートされたイメージのサポート

イメージをトランスレートするために使用した DECmigrate のバージョン	トランスレートされたイメージに対する OpenVMS Alpha のサポート		
	バージョン 1.0	バージョン 1.5	バージョン 6.1 以降
バージョン 1.0	サポートされる	サポートされる	サポートされる
バージョン 1.1	サポートされない	サポートされる	サポートされる
バージョン 1.1A	サポートされない	サポートされない	サポートされる

9.3 Translated Image Environment (TIE)

イメージのトランスレーションは，VAX アプリケーションの一部または全部を OpenVMS Alpha に移行するための 1 つの手段です。DECmigrate for OpenVMS AXP の VAX Environment Software Translator ユーティリティ (VEST) は，VAX 実行可能イメージまたは共用可能イメージを，同等の機能の Alpha イメージに変換することにより，トランスレートされたイメージを作成します。VEST は，オプションとして提供されるレイヤード製品 DECmigrate for OpenVMS AXP の構成要素です。

トランスレートされたイメージが OpenVMS Alpha で実行される場合には，Translated Image Environment (TIE) がイメージを正しく実行するのに必要な VAX 環境を提供します。TIE は TIE$SHARE と TIE$EMULAT_TV という共用可能イメージで構成されます。これらのイメージは VAX の複合命令を実行します。移行におけるイメージ・トランスレーションの役割については，『DECmigrate for OpenVMS AXP Systems Translating Images』を参照してください。

この後の節では，次のことについて説明します。

- ネイティブなイメージとトランスレートされたイメージの間の相互操作性
- トランスレートされたイメージの実行

- TIE 統計情報とフィードバック

ネイティブなイメージとトランスレートされたイメージの間の相互操作性

TIE は OpenVMS Alpha の他の構成要素と連携動作して，ネイティブなイメージとトランスレートされたイメージが相互に他のイメージを呼び出すことができるようにしています。相互操作性に依存したアプリケーションやランタイム・ライブラリを開発する場合には，コンパイル，リンク，またはトランスレートを行うときに，特定の手順に従わなければなりません。第 9.3.1.4 項に説明している最初の制限事項を参照してください。

トランスレートされたイメージの実行

トランスレートされたイメージを実行するには， DCL の RUN コマンドを使用します。次の例を参照してください。

```
$ RUN FOO_TV.EXE
```

OpenVMS Alpha に，適切なトランスレートされた共用可能イメージと実行時ライブラリが含まれていない限り，トランスレートされたイメージは正しく動作しません。イメージをトランスレートするときに， VEST は，入力イメージが参照しているイメージとライブラリに対応するイメージ情報ファイル (IIF— ファイル・タイプは.IIF) を必要とします。これらの.IIF ファイルを使用すると， VEST は共用可能イメージとライブラリのトランスレートされたバージョンを正しく参照するトランスレートされたイメージを作成できます。イメージのトランスレーションで使用されるイメージ情報ファイルは， OpenVMS Alpha で使用できるトランスレートされた共用可能イメージまたは実行時ライブラリのバージョンに正確に対応しなければなりません。

OpenVMS Alpha には，トランスレートされた実行時ライブラリと対応するイメージ情報ファイルがあります。これらは第 9.4 節に示すとおりです。トランスレートして実行するイメージで参照されているライブラリまたは共用可能イメージが含まれているかどうかを判断するには，このリストを確認してください。 OpenVMS Alpha に，必要な共用可能イメージまたはライブラリがない場合には，『DECmigrate for OpenVMS AXP Systems Translating Images』を参照してください。このマニュアルでは，イメージ情報ファイルの作成方法と使用方法を説明しています。

トランスレートされたライブラリがそのライブラリのネイティブ・バージョンに置き換えられた場合には，そのライブラリを指す論理名を適切に定義しなければなりません。つまり，image_TVをimageに再定義しなければなりません。

TIE 統計情報とフィードバック

TIE の実行時サポート機能の他に， TIE の統計情報とフィードバックを使用すると，トランスレートされたイメージの性能を向上するのに役立ちます。

- TIE は，トランスレートされたイメージの実行に関する統計情報を表示できます。これらの統計情報は，イメージが TIE リソースをどのように使用しているかとイメージ間のやり取りを記述します。

- TIE は VAX コードのインタプリテーションで検出された VAX エントリ・ポイントに関する情報を記録できます。イメージを再度トランスレートする場合には，VEST はこの情報を使用して，より多くの VAX コードを検出し，トランスレートします。

『DECmigrate for OpenVMS AXP Systems Translating Images』では，これらの機能について詳しく説明しており，また，これらの機能の使用を有効または無効に設定するための論理名の定義方法についても説明しています。

9.3.1 問題点と制限事項

この節では， TIE の問題点と制限事項について説明します。

9.3.1.1 条件ハンドラに関する制限事項

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両方に対して設定できる条件ハンドラのタイプには，永久的な制限事項があります。ネイティブ・ルーチンは，トランスレートされた条件ハンドラを設定することができず，トランスレートされたルーチンはネイティブな条件ハンドラを設定できません。ネイティブ・イメージまたはトランスレートされたイメージがこの制限に違反すると，実行時の結果は予測できなくなります。

9.3.1.2 例外ハンドラに関する制限事項

次の例外ハンドラの制限事項は永久的なものです。

- 正しいプログラム・ステータス・ロングワード (PSL) を受け取ることに依存している例外ハンドラを含むトランスレートされたイメージは，正しく機能しないことがあります。例外が報告されるときに， VAX の PSL がないため， Alpha のプログラム・ステータス (PS) がシグナル・アレイに報告されます。
- トランスレートされたイメージで，シグナル・アレイ内の PSL の変更に依存している例外ハンドラを使用している場合には，そのイメージは正しく機能しません。変更された PSL は障害のあるコードに伝達されません。

9.3.1.3 浮動小数点に関する制限事項

次の浮動小数点の制限事項は永久的なものです。

- 場合によっては，同じデータを操作する浮動小数点命令が， Alpha システムではトラップを生成するのに， VAX システムでは生成しないという状況があります。とくに， OpenVMS Alpha で VAX 浮動小数点命令を実行すると， VAX ハードウェアが正しく処理できる“ダーティ・ゼロ”に対してトラップが生成されます。“ダーティ・ゼロ”とは，ゼロの代わりに使用される浮動小数点値です。ダーティ・ゼロを使用する操作を実行するトランスレートされたイメージとの互換性を維持するために， TIE にはダーティ・ゼロを修正し，浮動小数点演算を再実行する条件ハンドラが含まれています。しかし，このハンドラが正常に機能するのは，イメージをトランスレートするときに，/PRESERVE=FLOAT_EXCEPTIONS 修飾子を使用した場合だけです。

トランスレートするときに/PRESERVE=FLOAT_EXCEPTIONS 修飾子を使用せず，ダーティ・ゼロに対して演算を実行するイメージは， HPARITH 例外を発生させ，要約状態のビット 1 がセットされます。トランスレートされたアプリケーションでこのような例外が発生した場合には，/PRESERVE=FLOAT_EXCEPTIONS を指定して再トランスレートしてください。 VAX のダーティ・ゼロが発生するのは一般に，浮動小数点データを 0 に初期化しなかったためです。この場合には，ダーティ・ゼロを使用するアプリケーションを OpenVMS Alpha に移植するために，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。

- Alpha の D53 浮動小数点 (小数部が 56 ビットではなく， 53 ビットの D 浮動小数点データ型) は， G 浮動小数点表現に変換された VAX の D 浮動小数点です。この変換の結果，次のような問題が発生します。次の VAX 命令について考えてみましょう。

```
MOVD    (SP),R2
MOVD    R2,-(SP)
```

VEST は，これらの VAX 命令を次のような Alpha コードにトランスレートします。

```
LDD     F2,0(R14)       ! Pick up D float
CVTDG   F2,F2           ! Convert to Canonical G Form with rounding
CVTGD   F2,F17          ! Convert back to D Form for storing
STD     F17,-8(R14)     ! Store the result
```

実行時に， VEST で生成されたコードは， D56 浮動小数点を G 正規形式に変換するときに，もっとも正確な G 浮動小数点の値を求めるために丸めを行います。場合によっては， G 正規形式に変換すると， D 浮動小数点の値が切り上げられ， D 浮動小数点では表現できない指数部を作成することがあります。この場合には，CVTGD 操作は HPARITH トラップを発生させ，その理由として浮動小数点オーバフローを示します。

トランスレートされたイメージで実行時にこの問題が発生した場合には， VEST の/FLOAT=D56_FLOAT 修飾子を使用して再度トランスレートを行い，プログラムが正しく実行されるようにしなければなりません。

9.3.1.4 相互操作性に関する制限事項

相互操作性に関して，次の制限事項に注意してください。

- トランスレートされたイメージを呼び出すネイティブ・ルーチンや，トランスレートされたイメージから呼び出されるネイティブ・ルーチンは，/TIE 修飾子を使用してコンパイルし，/NONATIVE_ONLY 修飾子を使用してリンクしなければなりません。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間の相互操作性の確認は実行時に行われます。ネイティブ・ルーチンのコンパイルとリンクで/TIE 修飾子と/NONATIVE_ONLY 修飾子を使用しなかった場合には，ネイティブ・ルーチンとトランスレートされたイメージが相互操作しようとするときに，実行時にエラーが発生します。このようなエラーが発生した場合には，ネイティブ・ルーチンの再コンパイルと再リンクを行ってください。

- /TIE 修飾子を指定せずにコンパイルしたネイティブ・ルーチンで，トランスレートされたルーチンに対して間接呼び出しを実行すると，実行時にアクセス違反が発生することがあります。間接呼び出しは，トランスレートされたルーチンのアドレスを格納した変数を通じて行われます。この場合，ネイティブ・ルーチンからトランスレートされたルーチンへの呼び出しを援助するための自動ジャケット・コードはありません。ネイティブ・コードはネイティブ・プロシージャ記述子としてルーチン・アドレスを使用しようとします。ネイティブ・プロシージャのコード・アドレスは，プロシージャ記述子のベースからオフセット PDSC$L_ENTRY の位置にあります。オフセットの値は 8 です。トランスレートされたルーチンのアドレスはプロシージャ記述子として取り扱われるため，そのアドレスからオフセット 8 の値が呼び出すコードとして使用されます。この結果，通常はアクセス違反が発生します。

  この問題が発生した場合には，デバッガを使用して次のことを確認してください。

  - R27 が，トランスレートされたイメージを指しているかどうかを確認します。
  - 8(R27) のビット<31:2>がアクセス違反アドレスのビット<31:2>に等しいことを確認してください (Alpha 命令はロングワード・アラインされるため，すべてのビットが使用されるわけではありません)。
  - R26 がネイティブ・イメージを指していることを確認してください。
  - -4(R26) が JSR R26, (26) 命令であることを確認してください。

  これらのすべての確認で問題が検出されなかった場合には，/TIE 修飾子を使用してネイティブ・ルーチンを再コンパイルし，実行時に自動ジャケットが有効になるようにします。

9.3.1.5 VAX C: **トランスレートされたプログラムの制限事項**

次のトランスレートされたVAX Cプログラムの制限事項は永久的なものです。

- プログラムでVAX C RTL ルーチンの brk()を使用して動的メモリを解放する場合には (つまり，現在のブレーク・アドレスより小さいブレーク・アドレスを要求した場合)， TIE が次に複合命令ルーチンを使用しようとしたときに，致命的なメモリ・アクセス違反が発生することがあります。このような違反が発生するのは，複合命令ルーチンが別のイメージ TIE$EMULAT_TV.EXE にあるからであり，これはルーチンの 1 つを最初に使用するときに， LIB$FIND_IMAGE_SYMBOL によって動的に起動されます。このことが発生する時期と，メモリを解放する brk()呼び出しに渡したアドレスに応じて，TIE$EMULAT_TV.EXEがロードされるメモリも解放される可能性があります。

  この問題を回避するには，メモリを解放するために brk()を絶対に使用しないか，またはメモリを解放するために後で使用するブレーク・アドレスを入手する前に，複合 VAX 命令を確実に実行します。メモリを割り当てるために brk()を使用しても問題はありません。

- vfork()とエグゼクティブ関数を使用するトランスレートされたVAX Cプログラムは，実行時にハングする可能性があります。VAX Cプログラムの子プロセスが強制終了され，エラーが発生した場合には，メールボックス入出力が完了するのを待ってハングしている可能性があります。この問題を回避するには，子プロセスが強制終了されないようにします。

9.4 トランスレートされたイメージのサポート

トランスレーション・サポートは，ユーザが Alpha に移行する際に発生する障害を削減するために提供されています。それは次の理由によるものです。

- 最初は言語サポートが完全に行われなかったため
- 再コンパイルのためにソース・コードを入手できないため
- VAX アーキテクチャの特定の機能に大きく依存しているコードを再コンパイルするのが困難なため

VAX バージョンが現在も活発に開発されている言語に対して，ネイティブな Alpha バージョンも提供されています。 TIE は， OpenVMS VAX バージョン 5.5-2 のリリースの時点で提供されていた言語機能をサポートできるように維持管理されています。

同様に，システム・サービスや実行時ライブラリのエントリ・ポイントの使用が，OpenVMS VAX バージョン 5.5-2 で提供されていたものに制限されているイメージに対しては，トランスレーションがサポートされます。

もっと新しい VAX レイヤード製品がインストールされている場合には，アプリケーションでトランスレーションに適したイメージを再構築することが必要なときに，いくつかの追加手順が必要になることがあります。たとえば， OpenVMS VAX 用の DECwindows Motif バージョン 1.2 またはバージョン 1.2–3 では，トランスレーションにとって適切であるように， OSF Motif バージョン 1.2.2 やバージョン 1.2.3 のライブラリではなく， OSF Motif バージョン 1.1.3 ライブラリまたは DECwindows XUI ライブラリを使用してイメージを作成しなければなりません。

同様に， DEC Fortran for VAX の最近のバージョンを使用するイメージの場合，トランスレーションに適した Fortran プログラムをコンパイルするために追加の修飾子が必要です。

詳細については，それぞれの VAX 製品のリリース・ノートを参照してください。

OpenVMS VAX イメージを将来，再構築したり，再トランスレーションしなければならない状況では，安全措置として，関連する OpenVMS VAX バージョン 5.5-2 共用可能イメージのコピーを別の VAX ディレクトリに保存し，これらのイメージに対して VAX アプリケーションの新しいバージョンをリンクするようにしてください。この手法を使用する場合，作成されるイメージは新しい OpenVMS VAX 共用可能イ

メージと互換性があり (OpenVMS の既存の機能を拡張した機能を使用できます)，また， OpenVMS Alpha にトランスレーションするためにイメージを正しく構築できます (共用可能イメージの新しいバージョンは必要ありません)。

この後の節では，トランスレートされたイメージ，イメージ情報ファイル，および OpenVMS Alpha で提供されるその他の関連ファイルについて説明します。

OpenVMS Alpha には，トランスレートされたメッセージ・イメージはありません。メッセージ・イメージはすべて，ネイティブ・イメージとして作成されます。

SYS$LIBRARY: **内のトランスレートされたイメージ**

BASRTL2_D53_TV.EXE
BASRTL2_D56_TV.EXE
BASRTL_D56_TV.EXE
BASRTL_TV_SUPPORT.EXE
BLAS1RTL_D53_TV.EXE
BLAS1RTL_D56_TV.EXE
COBRTL_D56_TV.EXE
DBLRTL_D56_TV.EXE
EDTSHR_TV.EXE
FORRTL2_TV.EXE
FORRTL_D56_TV.EXE
LIBRTL2_D56_TV.EXE
LIBRTL_D56_TV.EXE
MTHRTL_D53_TV.EXE
MTHRTL_D56_TV.EXE
PASRTL_D56_TV.EXE
PLIRTL_D56_TV.EXE
RPGRTL_TV.EXE
SCNRTL_TV.EXE
TECOSHR_TV.EXE
TIE$EMULAT_TV.EXE
UVMTHRTL_D53_TV.EXE
UVMTHRTL_D56_TV.EXE
VAXCRTLG_D56_TV.EXE
VAXCRTL_D56_TV.EXE
VMSRTL_TV.EXE

SYS$SYSTEM: **内のトランスレートされたイメージ**

DBLMSGMGR_TV.EXE
EDF_TV.EXE
EDT_TV.EXE
MONITOR_TV.EXE
TECO32_TV.EXE

IMAGELIB: **内のトランスレートされた** RTL **イメージ**

BASRTL2_D53_TV.EXE
BASRTL_D56_TV.EXE
BLAS1RTL_D53_TV.EXE
COBRTL_D56_TV.EXE
DBLRTL_D56_TV.EXE
FORRTL2_TV.EXE
FORRTL_D56_TV.EXE
LIBRTL_D56_TV.EXE
PLIRTL_D56_TV.EXE
RPGRTL_TV.EXE
SCNRTL_TV.EXE

トランスレートされた RTL の大部分は， D53 形式ではなく， D56 形式で提供されます。一部は両方の形式で提供されます。両方の形式が提供される場合には，省略時の形式は D53 です。トランスレートされた実行時ライブラリの詳細については，第 9.5 節を参照してください。

SYS$LIBRARY: **内のイメージ情報ファイル**

ACLEDTSHR.IIF
BASRTL2.IIF
BASRTL.IIF
BLAS1RTL.IIF
COBRTL.IIF
CONVSHR.IIF
CRFSHR.IIF
DBLRTL.IIF
DCXSHR.IIF
DISMNTSHR.IIF
DTKSHR.IIF
EDTSHR.IIF
ENCRYPSHR.IIF
EPC$SHR.IIF
FDLSHR.IIF
FORRTL.IIF
FORRTL2.IIF
INIT$SHR.IIF
LBRSHR.IIF
LIBRTL.IIF
LIBRTL2.IIF
MAILSHR.IIF
MOUNTSHR.IIF
MTHRTL.IIF
NCSSHR.IIF

P1_SPACE.IIF
PASRTL.IIF
PLIRTL.IIF
PPLRTL.IIF
PTD$SERVICES_SHR.IIF
RPGRTL.IIF
S0_SPACE.IIF
SCNRTL.IIF
SCRSHR.IIF
SECURESHR.IIF
SMBSRVSHR.IIF
SMGSHR.IIF
SORTSHR.IIF
SPISHR.IIF
TECOSHR.IIF
TPUSHR.IIF
UVMTHRTL.IIF
VAXCRTL.IIF
VAXCRTLG.IIF
VMSRTL.IIF

システム論理名の定義

次のシステム論理名は，トランスレートされた環境を使いやすくするために定義されています。

ACLEDTSHR_TV = ACLEDTSHR
CDDSHR_TV = CDDSHR
CONVSHR_TV = CONVSHR
CRFSHR_TV = CRFSHR
DCXSHR_TV = DCXSHR
DISMNTSHR_TV = DISMNTSHR
DTKSHR_TV = DTKSHR
ENCRYPSHR_TV = ENCRYPSHR
EPC$SHR_TV = EPC_SHR
FDLSHR_TV = FDLSHR
INIT$SHR_TV = INIT$SHR
LBRSHR_TV = LBRSHR
MAILSHR_TV = MAILSHR
MOUNTSHR_TV = MOUNTSHR
NCSSHR_TV = NCSSHR
PPLRTL_TV = PPLRTL
PTD$SERVICES_SHR_TV = PTD$SERVICES_SHR
SCRSHR_TV = SCRSHR
SECURESHR_TV = SECURESHR_JACKET
SMBSRVSHR_TV = SMBSRVSHR

```
SMGSHR_TV = SMGSHR
SORTSHR_TV = SORTSHR
SPISHR_TV = SPISHR
TPUSHR_TV = TPUSHR

BASRTL_TV = BASRTL_D56_TV
BASRTL2_TV = BASRTL2_D53_TV
BLAS1RTL_TV = BLAS1RTL_D53_TV
COBRTL_TV = COBRTL_D56_TV
DBLRTL_TV = DBLRTL_D56_TV
FORRTL_TV = FORRTL_D56_TV
LIBRTL_TV = LIBRTL_D56_TV
LIBRTL2_TV = LIBRTL2_D56_TV
MTHRTL_TV = MTHRTL_D53_TV
PASRTL_TV = PASRTL_D56_TV
PLIRTL_TV = PLIRTL_D56_TV
VAXCRTL_TV = VAXCRTL_D56_TV
VAXCRTLG_TV = VAXCRTLG_D56_TV

DBLMSGMGR = DBLMSGMGR_TV
EDTSHR_TV = EDTSHR
TECO32 = TECO32_TV
TECOSHR = TECOSHR_TV
VMSRTL = VMSRTL_TV

DBLRTLMSG = DBL$MSG
PASMSG = PAS$MSG
PLIMSG = PLI$MSG
RPGMSG = RPG$MSG
SCNMSG = SCN$MSG
VAXCMSG = DECC$MSG
```

9.5 トランスレートされた実行時ライブラリ

OpenVMS Alpha キットの一部として，DEC はトランスレートされた実行時ライブラリを提供しています。

VAX 実行時ライブラリのルーチンの一部は，倍精度算術演算に対して VAX D 浮動小数点データ型を使用します。

これらのライブラリのトランスレートされたバージョンでは，省略時の設定では，Alpha の D56 D 浮動小数点データ型が使用されます (VAX 実行時ライブラリでは D 浮動小数点を使用していました)。この結果，VAX D 浮動小数点の 56 ビットの仮数

部の完全な精度が提供されるため，実行時の性能は低下しますが，一貫性のある結果が求められます。

性能が重要な算術演算関連のライブラリの場合には，倍精度演算に対して Alpha の D53 D 浮動小数点データ型を使用する，トランスレートされた実行時ライブラリのバージョンも提供しています。これらのライブラリの場合，省略時の設定は D53 形式です。 D53 形式は，仮数部の下位 3 ビットの精度を犠牲にして性能を向上します。

次のトランスレートされたライブラリは， D56 形式でのみ提供されます。

- BASRTL
- COBRTL
- DBLRTL
- FORRTL
- LIBRTL
- LIBRTL2
- PASRTL
- PLIRTL
- VAXCRTL
- VAXCRTLG

次のトランスレートされたライブラリは， D56 形式と D53 形式 (省略時の設定) の両方で提供されます。

- BASRTL2
- BLAS1RTL
- MTHRTL
- UVMTHRTL

実行時ライブラリの D56 形式へのアクセス

実行時ライブラリを使用する場合には，省略時の設定により，次の処理が行われます。

- BASRTL2 の場合，倍精度データに対して MAT 関数を使用するトランスレートされた BASIC イメージは， D53 データ型を使用する BASIC 実行時ライブラリ・ルーチンを起動します。
- BLAS1RTL の場合，倍精度浮動小数点引数を使用する BLAS$関数を起動するトランスレートされたイメージは， D53 データ型を使用するルーチンを呼び出します。
- MTHRTL の場合， MTH$倍精度浮動小数点関数を起動するトランスレートされたイメージは， D53 データ型を使用するルーチンを呼び出します。

- その他の場合は，省略時の設定により， Alpha の D56 浮動小数点データ型が使用されます。

一部のユーザは D56 浮動小数点の完全な精度を必要とします。しかし， D56 ルーチンを使用すると，性能が大幅に低下します。 D56 ルーチンにアクセスするには，表 9–2 に示すように， D56 形式を指す実行時ライブラリの論理名を再定義しなければなりません。サイトの必要に応じて，論理名はプロセス単位で定義するか，またはシステム単位で定義できます。

表 9–2 実行時ライブラリの論理名

ライブラリ	論理名	D56 ルーチンの名前
BASRTL2	BASRTL2_TV	BASRTL2_D56_TV
BLAS1RTL	BLAS1RTL_TV	BLAS1RTL_D56_TV
MTHRTL	MTHRTL_TV	MTHRTL_D56_TV

9.5.1 CRF$FREE_VM と CRF$GET_VM: トランスレートされた呼び出しルーチン

トランスレートされたルーチンから CRF$FREE_VM と CRF$GET_VM を呼び出す場合，トランスレートされた呼び出し側は正しく機能しません。トランスレートされた呼び出し側は VAX JSB のセマンティックを期待していますが，ネイティブ・コードには (当然のことながら) Alpha JSB セマンティックが存在します。

この問題を回避するには，トランスレートされた呼び出し側は JSB ではなく， CALL を使用しなければなりません。

9.6 トランスレートされたVAX C実行時ライブラリ

この後の節では，トランスレートされたVAX C実行時ライブラリに関連するリリース・ノートを示します。

9.6.1 問題点と制限事項

この節では， OpenVMS Alpha のトランスレートされたVAX C実行時ライブラリ (VAX C RTL) で報告されている問題点と制限事項について説明します。

9.6.1.1 機能上の制限事項

トランスレートされたVAX C RTL は， OpenVMS VAX バージョン 5.4 のVAX C RTL のトランスレートされたバージョンです。VAX C RTL のそのリリースに存在する問題と制限事項はすべて，トランスレートされたVAX C RTL にもそのまま存在します。トランスレートされたVAX C RTL の機能上の制限事項は次のとおりです。

- fmod( )関数は D_FLOAT に対して正しい結果を生成しません。

- SIGFPE シグナルを使用する D_FLOAT プログラムは，すべての浮動小数点例外を検出できない可能性があります。/FLOAT=D56_FLOAT を使用するプログラムをトランスレートすると， SIGFPE に関するほとんどの問題を修正できます。
- sbrk( )関数は， SYS$EXPREG から戻される値と一致しないアドレスを戻します。
- HUGE_VAL 定数を使用するか，または算術演算関数 (HUGE_VAL を戻す可能性のある関数) を呼び出す D_FLOAT プログラムは，/FLOAT=D56_FLOAT を使用してトランスレートした場合を除き，異常終了する可能性があります。
- 特定の状況では，一部の算術演算関数 (D_FLOAT または G_FLOAT) は，エラー番号を ERANGE または EDOM に設定するのではなく，高い性能の算術演算トラップ例外を生成する可能性があります。

9.6.1.2 相互操作性に関する制限事項

トランスレートされたVAX C RTL がネイティブな DEC C RTL と相互操作する場合には，次の制限事項が適用されます。

- 次のように制御を移すために， longjmp 関数を使用することはできません。
  - ネイティブ・ルーチンからトランスレートされたルーチンへ
  - トランスレートされたルーチンからネイティブ・ルーチンへ
- malloc や calloc などによって割り当てられたメモリは，同じコンテキストで解放しなければなりません。つまり，トランスレートされたルーチンがメモリを割り当てる場合には，メモリの割り当てを解釈するための free 呼び出しもトランスレートされたルーチンで行わなければなりません。トランスレートされたルーチンでメモリを割り当て，そのメモリの割り当てをネイティブ・ルーチンで解除すると，ヒープが破壊されます。同様に，ネイティブ・ルーチンでメモリを割り当て，トランスレートされたルーチンでメモリの割り当てを解除すると，やはりヒープが破壊されます。
- トランスレートされたルーチンでシグナル (および関連) 関数によって設定されたシグナル・ハンドラは，シグナルが発生するときに起動されません。ネイティブ・シグナル・ハンドラだけが UNIX スタイルのシグナルを検出できます。
- SIGEMT， SIGTRAP， SIGIOT， SIGFPE の各シグナルがトランスレートされたイメージによって発生する場合，これらのシグナルを検出することはできません。
- exec 関数は類似したイメージを起動するためにだけ使用できます。つまり，ネイティブ・イメージで起動された exec 関数は，トランスレートされたイメージを実行できません。同様に，トランスレートされたイメージで起動された exec 関数は，ネイティブ・イメージを実行できません。
- 後でシステム呼び出しを実行するために，ネイティブ・イメージで vfork を実行してコンテキストを設定し，その後，トランスレートされたイメージでシステム呼び出しを起動した場合，アクセス違反が発生します。

- ファイル・ポインタとファイル記述子をネイティブ・イメージとトランスレートされたイメージの間で共用することはできません。ファイルをトランスレートされたイメージで開いた後，ネイティブ・イメージでそのファイル・ポインタを使用して読み込みまたは書き込みを実行しようとした場合，アクセス違反が発生するか，またはファイルが壊れる可能性があります。ネイティブ・イメージでファイルを開いた後，そのファイル・ポインタを使用してトランスレートされたイメージが読み込みまたは書き込みを実行しようとした場合にも，同じ結果が発生します。

これらの制限された処理を実行するプログラムでは，アクセス違反が発生するか，またはその他の例外が発生します。このような制限された操作を検出し，実行されないようにするために，徹底的なテストを行ってください。

9.7 トランスレートされたVAX COBOL プログラム

OpenVMS Alpha オペレーティング・システムでは，VAX COBOL バージョン 5.0 コンパイラ (またはそれ以前のコンパイラ) を使用してコンパイルされ，トランスレートされたVAX COBOL プログラムの実行をサポートします。

9.7.1 問題点と制限事項

VAX COBOL バージョン 5.1 コンパイラでコンパイルされたプログラムは，OpenVMS Alpha オペレーティング・システムでサポートされません。

10

ネイティブなイメージとトランスレートされたイメージの間の相互操作性の確認

この章では，トランスレートされた VAX イメージとの間で呼び出しを実行できるネイティブな Alpha イメージの作成方法について説明します。

10.1 概要

『DECmigrate for OpenVMS AXP Systems Translating Images』では， VAX Environment Software Translator (VEST) を使用して， VAX の実行可能イメージまたは共有可能イメージを，それと同じ機能を実行する Alpha イメージに変換する方法を説明しています (DECmigrate for OpenVMS Alpha はオプションとして提供されるレイヤード・プロダクトであり， VAX アプリケーションを Alpha システムに移行するのに必要なサポートを提供します。 VEST は DECmigrate ユーティリティの構成要素です)。

VEST を使用すれば，アプリケーションのすべての構成要素をトランスレートできます。たとえば，メインの実行可能イメージや，それを呼び出すすべての共有可能イメージをトランスレートできます。さらに，トランスレートされた部分とネイティブな部分が混在するアプリケーションも作成できます。たとえば，ネイティブ・イメージの高い性能を利用できるように，共有可能イメージのネイティブ・バージョンを作成し，それをトランスレートされたアプリケーションから呼び出すことができます。また，ネイティブな部分とトランスレートされた部分が混在するアプリケーションを作成しておき，その後で，段階的にアプリケーションのネイティブ・バージョンを作成することもできます。

トランスレートされた VAX イメージはネイティブな Alpha イメージと同様に使用できます。しかし，トランスレートされたイメージと相互操作可能なネイティブ・イメージを作成するには，この後の節に説明するように，特別な考慮が必要です。

10.1.1 トランスレートされたイメージと相互操作可能なネイティブ・イメージのコンパイル

トランスレートされたイメージとの間で呼び出しが可能なネイティブ・イメージを作成するには，ネイティブな Alpha イメージのソース・ファイルをコンパイルするときに/TIE 修飾子を指定しなければなりません。外部の呼び出しモジュールから使用できるプロシージャを含むソース・モジュールは，/TIE 修飾子を使用してコンパイルしなければなりません。/TIE 修飾子を指定する場合には，トランスレートされたイメージ

とネイティブ・イメージの間で正しく呼び出しができるように，コンパイラは実行時に Translated Image Environment (TIE) が必要とするプロシージャ・シグナチャ・ブロック (PSB) を作成します。 TIE はオペレーティング・システムの一部です。

トランスレートされたイメージ内に存在する可能性のあるコールバック (つまり，指定されたプロシージャの呼び出し) を実行するプロシージャを含むソース・モジュールをコンパイルする場合にも，/TIE 修飾子を指定しなければなりません。この場合には，/TIE 修飾子を指定すると，コンパイラは特殊なランタイム・ライブラリ・ルーチン OTS$CALL_PROC に対する呼び出しを生成し，トランスレートされたプロシージャへの外部呼び出しが正しく処理されるようにします。

/TIE 修飾子の他にも，トランスレートされたイメージとネイティブな共有可能イメージの間で正しく相互操作できるように，他のコンパイラ修飾子も指定しなければならないことがあります。たとえば，トランスレートされたイメージからネイティブな共有可能イメージを呼び出すときに，呼び出し側が倍精度浮動小数点演算のために VAX の D 浮動小数点形式を使用する場合 (VAX 用のコンパイラの省略時の設定)，Alpha システムでは，倍精度データの省略時の形式は VAX の D 浮動小数点ではないため，/FLOAT=D_FLOAT 修飾子を指定しなければなりません。 VAX の D 浮動小数点形式を指定するための正確な修飾子の構文については，各コンパイラの解説書を参照してください。 VAX の D 浮動小数点データ型は Alpha アーキテクチャではサポートされないため，このデータ型を使用すると，性能が低下する可能性があります。

さらにアプリケーション固有のセマンティックに応じて，バイト粒度やデータ・アラインメント， AST の不可分性などを強制的に設定するために，他のコンパイラ修飾子を指定しなければならない場合もあります。

10.1.2 トランスレートされたイメージと相互操作可能なネイティブ・イメージのリンク

トランスレートされた VAX イメージを呼び出すことができるネイティブな Alpha イメージを作成するには，/NONATIVE_ONLY 修飾子を指定してネイティブなオブジェクト・モジュールをリンクしなければなりません (/NONATIVE_ONLY はこの修飾子に対してリンカが使用する省略時の設定です)。この修飾子を指定すると，リンカは，コンパイラが作成した PSB 情報をイメージに挿入します。

/NATIVE_ONLY 修飾子はネイティブ・イメージからトランスレートされたイメージへの呼び出しにのみ影響を与えるため，トランスレートされた VAX イメージから呼び出されるネイティブな Alpha イメージを作成する場合には，この修飾子を指定する必要はありません。リンカの省略時の動作 (/NATIVE_ONLY 修飾子) は，ネイティブ・イメージからトランスレートされたイメージを呼び出すことを禁止することになりますが，トランスレートされたイメージからネイティブ・イメージが呼び出されるのを防げるものではありません。

段階的なシステム移行に際して，共有可能イメージのネイティブ・バージョンのシンボル・ベクタのレイアウトは，トランスレートされた共有可能イメージのシンボル・ベクタのレイアウトと一致しなければなりません。トランスレートされた共有可能イメージをネイティブな共有可能イメージに置き換えることについての詳しい説明は，第 10.3 節を参照してください。

10.2 トランスレートされたイメージの呼び出しが可能なネイティブ・イメージの作成

トランスレートされた VAX 共有可能イメージを呼び出すことができるネイティブな Alpha イメージを作成するには，次の操作を実行します。

1. **VAX 共有可能イメージをトランスレートする**

   VEST を使用して VAX イメージをトランスレート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。

2. **メイン・プログラムのネイティブな Alpha バージョンを作成する**

   ネイテイブな Alpha コードを作成するコンパイラを使用し，コマンド・ラインに/TIE 修飾子を指定して，ソース・モジュールをコンパイルします。

3. **ネイティブなオブジェクト・モジュールをトランスレートされた VAX 共有可能イメージとリンクする**

   他の共有可能イメージの場合と同様に，リンカ・オプション・ファイルにトランスレートされたイメージを指定します。

相互操作性について説明するために，例 10–1 と例 10–2 に示したプログラムについて考えてみましょう。例 10–1 は，例 10–2 に定義されている mysub というルーチンを呼び出します。

例 10–1 メイン・プログラム (MYMAIN.C) のソース・コード

```
#include <stdio.h>

int mysub();

main()
{
   int num1, num2, result;

   num1 = 5;
   num2 = 6;
```

```
   result = mysub( num1, num2 );
   printf("Result is: %d\n", result);
}
```

例 10–2 共有可能イメージ (MYMATH.C) のソース・コード

```
int myadd(value_1, value_2)
 int value_1;
 int value_2;
{
  int result;

  result = value_1 + value_2;
  return( result );
}

int mysub(value_1,value_2)
 int value_1;
 int value_2;
{
 int result;

 result = value_1 - value_2;
 return( result );
}

int mydiv( value_1, value_2 )
  int value_1;
  int value_2;
{
  int result;

  result = value_1 / value_2;
  return( result );
}

int mymul( value_1, value_2 )
  int value_1;
  int value_2;
{
  int result;

  result = value_1 * value_2;
  return( result );
}
```

これらの例から VAX イメージを作成するには， VAX システムの C コンパイラを使用してソース・モジュールをコンパイルします。共有可能イメージとして例 10–2 を実現するには，モジュールをリンクし，そのとき， LINK コマンド・ラインに /SHAREABLE 修飾子を指定し， UNIVERSAL オプションを使用するか，または転送ベクタ・ファイルを作成することにより，共有可能イメージでユニバーサル・シンボルを宣言します (共有可能イメージの作成方法についての説明は『OpenVMS Linker Utility Manual』を参照してください)。次の例は，共有可能イメージ MYMATH.EXE を作成する LINK コマンドを示しています。

```
$ LINK/SHAREABLE  MYMATH.OBJ,SYS$INPUT/OPT
GSMATCH=LEQUAL,1,1000
UNIVERSAL=myadd
UNIVERSAL=mysub
UNIVERSAL=mydiv
UNIVERSAL=mymul
Ctrl/Z
```

メイン・プログラムと共有可能イメージをリンクすることにより，実行可能イメージ MYMAIN.EXE を作成できます。次の例を参照してください。

```
$ LINK  MYMAIN.OBJ,SYS$INPUT/OPT
MYMATH.EXE/SHAREABLE
Ctrl/Z
```

リンカが VAX システムの省略時のページ・サイズ (512 バイト) より大きいページ・サイズを使用してイメージ・セクションを作成するようにするには， LINK コマンド・ラインに/BPAGE 修飾子を指定しなければなりません。この修飾子を指定しなかった場合には， VAX イメージをトランスレートするときに， VEST はこれらの 512 バイトの多くのイメージ・セクションを 1 ページの Alpha イメージに集めなければなりません。保護属性が矛盾する隣接イメージ・セクションを VEST が同じ Alpha ページに集める場合，すべてのイメージ・セクションに対して，もっともゆるやかな保護を割り当て，警告を出します (/BPAGE 修飾しの使い方についての詳しい説明は，『OpenVMS Linker Utility Manual』を参照してください)。

VAX イメージを作成した後， VEST を使用してそれらのイメージをトランスレートします。その場合，最初に共有可能イメージをトランスレートしなければなりません (VEST コマンド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)。次の例では，MYMATH_TV.EXE と MYMAIN_TV.EXE という名前のトランスレートされたファイルを作成します (VEST はイメージ・ファイルの名前の最後に“_TV”という文字を追加します)。

```
$ VEST  MYMATH.EXE
$ VEST  MYMAIN.EXE
```

トランスレートされた実行可能なメイン・イメージ MYMAIN_TV.EXE をネイティブ・バージョンと置き換えるには， Alpha コードを生成するコンパイラを使用して例 10–1 に示したソース・モジュールをコンパイルします。そのとき，コンパイル・コマンド・ラインに/TIE 修飾子を指定します。その後，ネイティブ・オブジェクト・モジュール MYMAIN.OBJ をリンクして，ネイティブな Alpha イメージを作成します。次の例に示すように，他の共有可能イメージをリンクするときと同様に，トランスレートされた共有可能イメージをリンカ・オプション・ファイルに指定してください。

```
$ LINK/NONATIVE_ONLY  MYMAIN.OBJ,SYS$INPUT/OPT
MYMATH_TV.EXE/SHAREABLE
Ctrl/Z
```

ネイティブなメイン・イメージは他の Alpha イメージと同様に実行できます。トランスレートされた共有可能イメージの名前 (MYMATH_TV) を，トランスレートされた共有可能イメージの位置を示す論理名として定義し (論理名 SYS$SHARE によって示されるディレクトリに登録されていない場合)， RUN コマンドを実行します。次の例を参照してください。

```
$ DEFINE MYMATH_TV YOUR$DISK:[YOUR_DIR]MYMATH_TV.EXE
$ RUN MYMAIN
```

10.3 トランスレートされたイメージから呼び出すことができるネイティブ・イメージの作成

トランスレートされた VAX イメージから呼び出すことができるネイティブな Alpha 共有可能イメージを作成するには，次の操作を実行します。

1. VAX **共有可能イメージをトランスレートします**

   最終的には，共有可能イメージの VAX バージョンをネイティブ・バージョンに置き換えますが，この時点では VEST インターフェイス情報ファイル (IIF) を作成するために共有可能イメージをトランスレートしなければなりません。 VEST は共有可能イメージを呼び出すイメージをトランスレートするときに，共有可能イメージに関連する IIF を必要とします。 IIF ファイルについての説明と，VAX イメージをトランスレートするために VEST を使用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 (トランスレートされた共有可能イメージ内でシンボル・ベクタのレイアウトを制御するには，この処理を繰り返さなければなりません。詳しくは第 10.3.1 項を参照してください)。

2. **共有可能イメージを呼び出す** VAX **実行可能イメージをトランスレートします**

3. **共有可能イメージのネイティブな** Alpha **バージョンを作成します**

   Alpha コードを生成するコンパイラを使用してソース・モジュールをコンパイルします。そのとき，コマンド・ラインに/TIE 修飾子を指定します。

4. **オブジェクト・モジュールをリンクして，ネイティブな** Alpha **共有可能イメージを作成します**

   共有可能イメージでユニバーサル・シンボルを宣言するために， SYMBOL_VECTOR オプションを使用します。互換性を維持するために， VAX 共有可能イメージで宣言した順序と同じ順序で， SYMBOL_VECTOR オプションにユニバーサル・シンボルを宣言します。

注意

トランスレートされた VAX **共有可能イメージと置換するために，ネイティブな** Alpha **共有可能イメージを作成する場合には，** SYMBOL_VECTOR **オプションの最初のエントリとして** SPARE **キーワードを指定することにより，シンボル・ベクタの最初のエントリを空にしておかなければなりません。** VEST

は，トランスレートされた VAX イメージ内で最初のシンボル・ベクタ・エントリを独自の目的で使用します。

次の例では，例 10–2 のソース・モジュールからネイティブな共有可能イメージを作成します。

```
$ LINK/SHAREABLE  MYMATH.OBJ,SYS$INPUT/OPT
GSMATCH=LEQUAL,1,1000  1
SYMBOL_VECTOR=(SPARE,-
               myadd=procedure,-  2
               mysub=procedure,-
               mydiv=procedure,-
               mymul=procedure)
Ctrl/Z
```

1 共有可能イメージのメジャー識別番号とマイナー識別番号を指定します。これらの識別番号の値は， VAX 共有可能イメージに指定した値と一致しなければなりません (GSMATCH オプションの使い方についての説明は『OpenVMS Linker Utility Manual』を参照してください)。

2 共有可能イメージでユニバーサル・シンボルを指定します。

5. **ネイティブな Alpha イメージのシンボル・ベクタのレイアウトは，トランスレートされた VAX イメージのシンボル・ベクタのレイアウトと一致しなければなりません**

   これらのシンボル・ベクタでのシンボルのオフセットを決定する方法についてと，レイアウトを一致するように制御する方法については，第 10.3.1 項を参照してください。

トランスレートされたメイン・イメージ (MYMAIN_TV.EXE) は，トランスレートされた VAX 共有可能イメージ， MYMATH_TV.EXE と組み合わせて実行でき，また，ネイティブな Alpha 共有可能イメージ， MYMATH.EXE と組み合わせて実行することもできます。省略時の設定では，トランスレートされた実行可能イメージはトランスレートされた共有可能イメージを呼び出します (トランスレートされた実行可能イメージには，トランスレートされた共有可能イメージを示すグローバル・イメージ・セクション・ディスクリプタ[GISD]が含まれています。イメージ・アクティベータは，そのイメージがリンクされている共有可能イメージを起動します)。

トランスレートされたメイン・イメージをネイティブな共有可能イメージと組み合わせて実行するには，共有可能イメージ MYMATH_TV の名前を，ネイティブな Alpha 共有可能イメージ， MYMATH.EXE の位置を指す論理名として定義します。次の例を参照してください。

```
$ DEFINE MYMATH_TV YOUR_DISK:[YOUR_DIR]MYMATH.EXE
$ RUN MYMAIN_TV
```

10.3.1 シンボル・ベクタ・レイアウトの制御

段階的なシステム移行に際して，トランスレートされた VAX 共有可能イメージに代わって用いるネイティブな Alpha 共有可能イメージを作成するには，共有可能イメージ内のユニバーサル・シンボルが両方のイメージでシンボル・ベクタ内の同じオフセットに設定されるようにしなければなりません。 VAX 共有可能イメージをトランスレートする場合， VEST は元の VAX 共有可能イメージで宣言されたユニバーサル・シンボルを含むシンボル・ベクタをイメージに対して作成します (トランスレートされたイメージは実際には， VEST が作成する Alpha イメージであり，他の Alpha 共有可能イメージと同様にシンボル・ベクタにユニバーサル・シンボルが登録されています)。トランスレートされた共有可能イメージと互換性のあるネイティブな共有可能イメージを作成するには，ネイティブな Alpha 共有可能イメージと，それに対応するトランスレートされた VAX 共有可能イメージの両方で，同じシンボルがシンボル・ベクタ内の同じオフセットに登録されていなければなりません。

VEST がトランスレートされた VAX イメージ内で作成するシンボル・ベクタをレイアウトする方法を制御するには，シンボル・インフォメーション・ファイル (SIF) を作成し，トランスレーション操作をそのファイルを参照しながら行います。 SIF ファイルはテキスト・ファイルであり，このファイルを使用すれば，トランスレートされたイメージに対して VEST が作成するシンボル・ベクタ内のエントリのレイアウトを指定でき，どのシンボルを，トランスレートされた共有可能イメージのグローバル・シンボル・テーブル (GST) に登録しなければならないかも指定できます。シンボル・ベクタのレイアウトを指定しなかった場合には， VEST は共有可能イメージの再トランスレーションでレイアウトを変更する可能性があります。 VEST は独自の目的で使用するために，最初のシンボル・ベクタ・エントリを確保します。シンボル・インフォメーション・ファイル (SIF) 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。

ネイティブな共有可能イメージ内でシンボル・ベクタのレイアウトを制御するには，SYMBOL_VECTOR オプションを指定します。リンカは SYMBOL_VECTOR オプション文にシンボルが指定されている順序と同じ順序でシンボル・ベクタ内にエントリをレイアウトします。 SYMBOL_VECTOR オプションにシンボルを指定する場合には， VAX 共有可能イメージを作成するために使用した転送ベクタ内でのシンボルの順序と同じ順序になるようにしてください。 SYMBOL_VECTOR オプションの使い方についての詳しい説明は，『OpenVMS Linker Utility Manual』を参照してください。

トランスレートされた共有可能イメージのシンボル・ベクタがネイティブな共有可能イメージのシンボル・ベクタと一致するかどうかを確認するには，次の操作を実行します。

1. **/SIF 修飾子を指定して VAX 共有可能イメージをトランスレートします**

   /SIF 修飾子を指定した場合には， VEST はシンボル・ベクタの内容をリストとして登録した SIF ファイルを作成します (SIF ファイルの作成と解釈方法については，『『DECmigrate for OpenVMS AXP Systems Translating Images』』を参照してください)。次の例は，共有可能イメージ MYMATH.EXE に対して，VEST が作成した SIF ファイルを示しています。エントリがシンボル・ベクタ内の番目の位置から始まっていることに注意してください (オフセットは 16 進数の 10)。

```
! .SIF Generated by VEST (V1.0) on
! Image "MYMATH", "V1.0"
MYDIV                           00000018 +S +G 00000030    00 4e
MYSUB   1                       0000000c +S +G 00000020 2  00 4e
MYADD                           00000008 +S +G 00000010    00 4e
MYMUL                           00000010 +S +G 00000040    00 4e
```

   1 ユニバーサル・シンボル MYSUB に対するエントリ

   2 トランスレートされたイメージのシンボル・ベクタ内での MYSUB に対するエントリのオフセット

2. **ネイティブな共有可能イメージ内でのシンボル・ベクタのオフセットを調べます**

   ネイティブな共有可能イメージでシンボル・ベクタ内のシンボルのオフセットを判断するには， ANALYZE/IMAGE ユーティリティを使用します。次の例は共有可能イメージ MYMATH.EXE の解析から抜粋したものであり， MYSUB というシンボルのオフセットを示しています。

```
.
.
.
4) Universal Symbol Specification (EGSD$C_SYMG)
 data type: DSC$K_DTYPE_Z (0)
 symbol flags:
  (0)  EGSY$V_WEAK      0
  (1)  EGSY$V_DEF       1
  (2)  EGSY$V_UNI       1
  (3)  EGSY$V_REL       1
  (4)  EGSY$V_COMM      0
  (5)  EGSY$V_VECEP     0
  (6)  EGSY$V_NORM      1
 psect: 0
 value: 16 (%X'00000020')
 symbol vector entry (procedure)
  %X'00000000 00010000'
  %X'00000000 00000050'
 symbol: "MYSUB"
  .
  .
  .
```

3. **必要に応じて，SIF ファイルに登録されているオフセットを変更します**

   SIF ファイルに登録されているオフセットがネイティブな共有可能イメージのオフセットと一致しない場合には，テキスト・エディタを使用して，これらのオフセットを修正しなければなりません。シンボル・ベクタの最初のエントリは VEST ユーティリティが使用するために確保されています。

4. **VAX 共有可能イメージを再トランスレートします**
   トランスレーション操作では，SIF ファイルを参照してください

   このトランスレーション操作で，VEST は SIF ファイルに指定されたオフセットを使用して，トランスレートされたイメージにシンボル・ベクタを作成します。VEST はデフォルトのデバイスおよびディレクトリから SIF ファイルを検索します (VEST ユーティリティの使い方については，『DECmigrate for OpenVMS AXP Systems Translating Images』を参照してください)。

10.3.2 特殊なトランスレートされたイメージ (ジャケット・イメージ) と代用イメージの作成

場合によっては，VAX 共有可能イメージを Alpha 共有可能イメージに完全に置き換えることができない場合があります。たとえば，VAX 共有可能イメージで VAX アーキテクチャ固有の機能を使用している場合などです。このような場合には，元の VAX 共有可能イメージの機能を実行できるように，トランスレートされたイメージとネイティブ・イメージの両方を作成しなければなりません。また，場合によっては，トランスレートされた VAX 共有可能イメージと新しい Alpha 共有可能イメージとの間に関係を定義しなければならないことがあります。どちらの場合にも，トランスレートされた VAX イメージは**ジャケット・イメージ**として作成されなければなりません。

10.3 トランスレートされたイメージから呼び出すことができるネイティブ・イメージの作成

ジャケット・イメージを作成するには，VAX システムで新しい Alpha イメージの代用バージョンを作成します。その後，変更された VAX 共有可能イメージを作成し，/JACKET=shrimg修飾子を指定して，この共有可能イメージをトランスレートします。ただし，shrimgは新しい Alpha 共有可能イメージの名前です。代用イメージのトランスレーションは前もって実行しておかなければなりません。これは，代用イメージを記述する IIF ファイルが必要だからです。代用イメージの作成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。

11

OpenVMS Alpha コンパイラ

この章では，ネイティブな OpenVMS Alpha 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 Alpha 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。

- DEC Ada (第 11.1 節)
- DEC C (第 11.2 節)
- DEC COBOL (第 11.3 節)
- Digital Fortran (第 11.4 節)
- DEC Pascal (第 11.5 節)

コンパイラの相違点が発生するのは，次の 2 つの理由によります。それは，OpenVMS VAX で動作するコンパイラの以前のバージョンと現在のバージョンとの間に違いがあるためと， VAX コンピュータと Alpha コンピュータで動作するバージョンに違いがあるためです。 OpenVMS Alpha コンパイラは， OpenVMS VAX 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。

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をサポートします。コンパイラは OpenVMS VAX システムの場合と同じ省略時のファイル・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 Alpha システムで提供されません。

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

11.1 DEC Ada の Alpha システムと VAX システム間の互換性

DEC Ada は，VAX Adaに含まれる標準的および拡張された Ada 言語機能を，ほとんどすべて含んでいます。これらの機能は次の解説書で説明されています。

- 『DEC Ada Language Reference Manual』

- 『Developing Ada Programs on OpenVMS Systems』
- 『DEC Ada Run-Time Reference Manual for OpenVMS Systems』

しかし，プラットフォーム・ハードウェアの違いにより，いくつかの機能はサポートされておらす， VAX システムと Alpha システムでは異なる機能もあります。あるシステムから別のシステムへのプログラムの移行を助けるため，移行の節ではこれらの違いを説明します。

注意

すべてのシステムの各リリースごとに，これらの機能のすべてがサポートされるわけではありません。詳しくは，DEC Ada のリリース・ノートを参照してください。

11.1.1 データ表現とアラインメントにおける相違点

概して， DEC Ada はすべてのプラットフォームで同じデータ・タイプをサポートします。しかし，以下の違いに注意してください。

- H 浮動小数点データ

  VAX システムではサポートされているが， Alpha システムではサポートされていない

- IEEE 浮動小数点データ型

  Alpha システムではサポートされているが， VAX システムではサポートされていない

- 自然なアラインメント

  Alpha システムでは， DEC Ada は省略時設定としてレコードとアレイの構成要素を自然な境界にアラインします。 VAX システムでは， DEC Ada はレコードとアレイの構成要素をバイト境界にアラインします。アラインメントは COMPONENT_ALIGNMENT プラグマで指定できます。レコード表現節の最大アラインメントは， VAX システムでも Alpha システムでも2^9です。

11.1.2 タスクに関する相違点

タスクの優先順位とスケジューリング，およびタスク制御ブロック・サイズはアーキテクチャ固有です。詳しくは，リリース・ノートを参照してください。

11.1.3 プラグマに関する相違点

プラグマには以下のような違いがあります。

- COMPONENT_ALIGNMENT プラグマ

  Alpha システムでは， COMPONENT_SIZE が省略時の選択肢です。 VAX システムでは， STORAGE_UNIT が省略時の選択肢です。

- FLOAT_REPRESENTATION プラグマ

  Alpha システムでは，このプラグマは IEEE_FLOAT と VAX_FLOAT という 2 つの選択肢をサポートします。 VAX システムでは， VAX_FLOAT をサポートします。

- LONG_FLOAT プラグマ Alpha システムでは LONG_FLOAT プラグマは，FLOAT_REPRESENTATION プラグマの値が VAX_FLOAT であるときのみサポートされます。

- SHARED プラグマ

  システム間で異なるデータ型の制限があります。詳しくは，『DEC Ada Run-Time Reference Manual for OpenVMS Systems』を参照してください。

- MAIN_STORAGE プラグマ

  Alpha システムではサポートされません。

- SHARE_GENERIC プラグマ

  Alpha システムではサポートされません。

- TIME_SLICE プラグマ

  VAX システムと Alpha システムでのこのプラグマのサポートには，実行に関するいくつかの違いがあります。詳しくは，『DEC Ada Run-Time Reference Manual for OpenVMS Systems』を参照してください。

11.1.4 SYSTEM パッケージの相違点

SYSTEM パッケージに関しては，以下の変更があります。

- SYSTEM.IEEE_SINGLE_FLOAT と SYSTEM.IEEE_DOUBLE_FLOAT

  Alpha システムではサポートしていますが， VAX システムではサポートしていません。

- SYSTEM.H_FLOAT

  VAX システムではサポートしていますが， Alpha システムではサポートしていません。

- SYSTEM.MAX_DIGITS

  VAX システムでの値は 33， Alpha システムでの値は 15 です。

- SYSTEM.NAME

  DEC Ada を使用できる各システムで特定の列挙型がサポートされます。

- SYSTEM.SYSTEM_NAME

  Alpha システムでは， OpenVMS_Alpha という名前がサポートされます。

- SYSTEM.TICK

  Alpha システムでの値は10.0^{-3}(1 ms) です。 VAX システムでの値は10.0^{-2}(10 ms) です。

さらに， VAX システムでサポートされる以下のタイプとサブプログラムは， Alpha システムではサポートされません。

SYSTEM.READ_REGISTER
SYSTEM.WRITE_REGISTER
SYSTEM.MFPR
SYSTEM.MTPR
SYSTEM.CLEAR_INTERLOCKED
SYSTEM.SET_INTERLOCKED
SYSTEM.ALIGNED_WORD
SYSTEM.ADD_INTERLOCKED
SYSTEM.INSQ_STATUS
SYSTEM.REMQ_STATUS
SYSTEM.INSQHI
SYSTEM.REMQHI
SYSTEM.INSQTI
SYSTEM.REMQTI

11.1.5 他の言語パッケージ間での相違点

以下に他のパッケージでの違いを示します。

- CALENDAR パッケージ

  システム間で実行方法に違いがあります。詳しくは，『DEC Ada Language Reference Manual』を参照してください。

- MATH_LIB パッケージ

  システム間で実行方法に違いがあります。各パッケージの解説を参照してください。

- SYSTEM_RUNTIME_TUNING パッケージ

  このパッケージは VAX システムと，いくつかの制限事項はありますが Alpha システムでサポートされます。詳しくは，『DEC Ada Run-Time Reference Manual for OpenVMS Systems』を参照してください。

11.1.6 あらかじめ定義されている命令に対する変更

VAX システムでサポートされる以下の 2 つのあらかじめ定義されている命令は Alpha システムではサポートされません。

- LONG_LONG_FLOAT_TEXT_IO
- LONG_LONG_FLOAT_MATH_LIB

11.2 DEC C for OpenVMS Alpha システムとVAX Cとの互換性

DEC C を構成するコンパイラ群は，ANSI に準拠する基本的な C 言語を定義し，これらの言語は Alpha アーキテクチャも含めて，すべての DEC プラットフォームで使用できます。詳しくは，DEC C の解説書を参照してください。

11.2.1 言語モード

DEC C for OpenVMS Alpha システムは ANSI C 標準規格に準拠し，オプションとしてVAX Cおよび Common C (pcc) の拡張機能をサポートします。オプションとして提供されるこれらの拡張機能は**モード**と呼び，これらの拡張機能を起動するには，/STANDARD 修飾子を使用します。表 11–1 はこれらのモードと，各モードを起動するのに必要なコマンドと修飾子の構文を示しています。

表 11–1 DEC C for OpenVMS Alpha コンパイラの操作モード

モード	コマンド修飾子	説明
省略時の設定	/STANDARD=RELAXED_ANSI89	ANSI C 標準規格に準拠するが，弊社の追加キーワードや，1 文字目がアンダースコアでない事前定義マクロも使用できる。
ANSI C	/STANDARD=ANSI89	厳密に ANSI C に準拠した言語のみを受け付ける。
VAX C	/STANDARD=VAXC	ANSI C 標準規格の他にVAX Cの拡張機能も使用できる。これらの拡張機能が ANSI C 標準規格と互換性がない場合でも使用可能である。
Common C (pcc)	/STANDARD=COMMON	ANSI C標準規格の他に，Common C の拡張機能も使用できる。これらの拡張機能が ANSI C 標準規格と互換性がない場合も使用可能である。
Microsoft との互換性	/STANDARD=MS	Microsoft Visual C++ コンパイラ製品に付属する C コンパイラの言語規則に従ってソース・プログラムを解釈する。

11.2.2 DEC C for OpenVMS Alpha システムのデータ型のマッピング

DEC C for OpenVMS Alpha システムのコンパイラは，対応する VAX コンパイラとほとんど同じデータ型マッピングをサポートします。表 11–2 は， Alpha アーキテクチャでの C 言語の算術演算データ型のサイズを示しています。

表 11–2 DEC C for OpenVMS Alpha コンパイラでの算術演算データ型のサイズ

C データ型	VAX Cのマッピング	DEC C のマッピング
pointer	32	32 または 64[1]
long	32	32
int	32	32
short	16	16
char	8	8
float	32	32[2]
double	64[2]	64[2]
long double	64[2]	64[2]
__int16	NA	16
__int32	NA	32
__int64	NA	64

[1]実現されている場合には，ソース・ファイルでプラグマを使用するか，またはコマンド行修飾子を使用することにより，サイズを選択できる。

[2]コマンド行修飾子を使用することにより， Alpha で D， F， G， S，または T 浮動小数点データ型にマッピングする方法を選択できる。第 11.2.2.1 項を参照。

移植性を向上するために， DEC C for OpenVMS Alpha システムのコンパイラでは，各データ型に対してマクロを定義するヘッダ・ファイルが準備されています。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 マクロを使用します。

11.2.2.1 浮動小数点マッピングの指定

C の浮動小数点データ型と Alpha の浮動小数点データ型の間のマッピングは，コマンド行修飾子によって制御されます。 Alpha アーキテクチャでは，次の浮動小数点データ型をサポートします。

- F 浮動小数点 (OpenVMS VAX システムと同じ)
- D 浮動小数点 (53 ビットの精度)
- G 浮動小数点 (OpenVMS VAX システムと同じ)
- S 浮動小数点 (IEEE 単精度 –32 ビット)
- T 浮動小数点 (IEEE 倍精度 –64 ビット)
- X 浮動小数点 (IEEE 拡張倍精度 –128 ビット)

コマンド行修飾子を使用すれば，標準的な C データ型の float と double が Alpha のどの浮動小数点データ型にマッピングされるかを制御できます。たとえば，/FLOAT=G_FLOAT 修飾子を指定した場合には， DEC C は float データ型を Alpha の F 浮動小数点データ型にマッピングし， double データ型を Alpha の G 浮動小数点データ型にマッピングします。表 11–3 は浮動小数点オプションを示しています。各コマンド行に浮動小数点修飾子は 1 つだけ指定できます。

表 11–3 DEC C の浮動小数点マッピング

コンパイラ・オプション	Float	Double	Long Double
/FLOAT=F_GLOAT	F 浮動形式	G 浮動形式	
/FLOAT=D_FLOAT	F 浮動形式	D-53 浮動小数点	
/FLOAT=IEEE_FLOAT	S 浮動形式	T 浮動形式	
/L_DOUBLE_SIZE=128 (省略時の値)	—	—	X 浮動形式

11.2.3 Alpha 命令にアクセスする組み込み機能

DEC C には，表 11–4 に示す機能があり，これらの機能は Alpha システム固有の機能です。この後の節では，これらの機能について説明します。

表 11–4 OpenVMS Alpha システム固有の DEC C コンパイラ機能

機能	説明
一部の Alpha 命令へのアクセス	組み込み機能として使用できる
一部の VAX 命令へのアクセス	Alpha PALcode を通じて使用できる
不可分な組み込み機能	AND， OR，および ADD 演算の不可分性を保証する

11.2.3.1 Alpha 命令のアクセス

DEC C は，特にシステムおプレベルのプログラミングのための C 言語により表現できない機能を提供するために正確な Alpha 命令をサポートします。例えば，次のようなものがあります。

- TRAPB は命令パイプラインをドレインします。
- MB はメモリ・バリアとして機能します。

11.2.3.2 Alpha 特権付きアーキテクチャ・ライブラリ (PALcode) 命令のアクセス

Alpha アーキテクチャでは，特定の VAX 命令を Alpha 特権付きアーキテクチャ・ライブラリ (PALcode) 命令として実現しています。 DEC C では，次の PALcode 命令をアクセスできます。

- INSQUEx — エントリをロングワード・キューまたはクォドワード・キューに登録します。
- INSQxI — エントリをキューに登録し，インターロックします。

- REMQUEx — エントリをロングワード・キューまたはクォドワード・キューから削除します。
- REMQxI — エントリをキューから削除し，インターロックします。

11.2.3.3 複数の操作の組み合わせに対する不可分性の保証

VAX アーキテクチャでは，変数のインクリメントなど，特定の組み合わせ操作は不可分に実行されることが保証されます (つまり，途中で割り込みが発生することはありません)。 Alpha システムでこれと同じ機能を実現するために， DEC C は不可分性を保証して操作を実行できるような組み込み機能を準備しています。表 11–5 はこれらの不可分な組み込み機能を示しています。これらの組み込み機能についてん詳細は DEC C 言語の解説書を参照してください。

表 11–5 不可分性組み込み機能

不可分性組み込み機能	説明
__ADD_ATOMIC_LONG(ptr, expr, retry_count) __ADD_ATOMIC_QUAD(ptr,expr, retry_count)	ptr によって示されるデータ引数に式 expr を追加する。任意に指定できる retry_count パラメータは，操作を繰り返す回数を指定する (省略時の設定では，操作は永久に繰り返される)。
__AND_ATOMIC_LONG(ptr, expr, retry_count) __AND_ATOMIC_QUAD(ptr, expr, retry_count)	ptr によって示されるデータ・セグメントをフェッチし，式 expr との間で論理 AND 演算を実行し，結果を格納する。retry_count パラメータは，操作を繰り返す回数を指定する (省略時の設定では，操作は永久に繰り返される)。
__OR_ATOMIC_LONG(ptr, expr, retry_count) __OR_ATOMIC_QUAD(ptr, expr, retry_count)	ptr によって示されるデータ・セグメントをフェッチし，式 expr との間で論理 OR 演算を実行し，結果を格納する。retry_count パラメータは操作を繰り返す回数を指定する (省略時の設定では，操作は永久に繰り返される)。

これらの組み込み機能は，割り込みを発生させずに操作を最後まで実行することだけを保証します。同時に書き込みアクセスが実行されるような変数に対して不可分な操作を実行する場合 (たとえば， AST とメイン・ライン・コードから書き込まれる変数や 2 つの並列プロセスから書き込まれる変数など)，volatile 属性によって変数を保護しなければなりません。

さらに， DEC C for OpenVMS Alpha システムでは， VAX インターロック命令と同じ機能を実行するために次の命令をサポートします。

- TESTBITSSI
- TESTBITCCI

これらの組み込み機能は，不可分な組み込み機能と同様にretry_count パラメータを使用して，ループが永久に実行されるのを防止します。

11.2.4 VAX Cと DEC C for OpenVMS Alpha システムのコンパイラの相違点

次の機能はVAX Cでも使用できますが，DEC C for OpenVMS Alpha システムの省略時の動作とは異なります。しかし，これらの機能の一部に対しては，コマンド行修飾子とプラグマ命令を使用することにより，VAX Cと同じ動作を実行できます。

11.2.4.1 データ・アラインメントの制御

自然な境界にアラインされていないデータをアクセスすると，Alpha システムでは性能が著しく低下するため，DEC C for OpenVMS Alpha システムは省略時の設定により，データを自然な境界にアラインします。この機能を無効にし，VAX のアラインメント (パックされたアラインメント) を実行するには，ソース・ファイルに#pragma nomember_alignment プリプロセッサ命令を指定するか，/NOMEMBER_ALIGNMENT コマンド行修飾子を使用します。

11.2.4.2 引数リストのアクセス

&argv1 などの引数のアドレスを検出すると，DEC C for OpenVMS Alpha システムは，すべての引数をスタックに移動する関数に対してプロローグ・コードを生成します (homing 引数と呼ぶ) が，その結果，性能が低下します。また，引数リスト“walking”は，<varargs.h> または<stdargs.h> インクルード・ファイルで関数を使用しなければ実現できません。

11.2.4.3 例外の同期化

Alpha アーキテクチャでは，算術演算例外がただちに報告されないため，後続の例外が通知される前に静的変数への代入が実行されることを期待することはできません (volatile属性を使用した場合でも)。

11.2.4.4 動的条件ハンドラ

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

11.2.5 C プログラマのための STARLET データ構造体と定義

OpenVMS Alpha バージョン 1.0 には，SYS$STARLET_C.TLB という新しいファイルが含まれており，このファイルには STARLETSD.TLB に相当する STARLET 機能を提供するすべての.H ファイルが格納されています。現在，DEC C コンパイラには DECC$RTLDEF.TLB の他に SYS$STARLET_C.TLB ファイルが同梱されており，以前のVAX C Compiler に同梱されていた VAXCDEF.TLB の代わりに使用されます。DECC$RTLDEF.TLB には，コンパイラと RTL をサポートするすべての.H ファイル，たとえば，STDIO.H などが格納されています。

次の相違点があるために，ソースを変更しなければならない可能性があります。

- RMS 構造体

  以前は， FAB， NAM， RAB， XABALL などの RMS 構造体は，たとえば“struct RAB {...”などのように，適切な.H ファイルで定義されていました。 OpenVMS Alpha バージョン 1.0 で提供される.H ファイルでは，これらの構造体は“struct rabdef {...”として定義されています。この違いを補正するために，“#define RAB rabdef”という形式の行が追加されました。しかし，この変更のためにソースを変更しなければならない状況が 1 つだけあります。これらの構造体のいずれかを指すポインタを格納したプライベート構造体を使用しており，そのプライベート構造体が RMS 構造体を定義する前に定義されている場合には (ただし，使用されません)，次のようなコンパイル時エラーが発生します。

  ```
  %CC-E-PASNOTMEM, In this statement, "rab$b_rac" is not a member of "rab".
  ```

  このエラーを回避するには，プライベート構造体より前に RMS 構造体が定義されるように，ソース・ファイルを変更します。通常は，“#include”文を移動します。

- LIB (特権インタフェース) 構造体

  これまで歴史的には， LIB の 3 つの構造体 (NFBDEF.H， FATDEF.H，および FCHDEF.H) は.H ファイルとして提供されてきました。 OpenVMS Alpha バージョン 1.0 とバージョン 1.5 では，これらのファイルは.H ファイルとして提供されていました (新しい SYS$STARLET_C.TLB にはありません)。 OpenVMS Alpha 7.2 では，すべての LIB 構造体と定義を格納した SYS$LIB_C.TLB ファイルが追加されました。これらの 3 つの.H ファイルは.TLB の一部として提供されるようになり，個別には提供されなくなりました。これらのファイルで既存の変則的な状態を保存しようとする試みは行われていないため，ソースを変更しなければならない可能性があります。 LIB からの構造体と定義は特権インタフェースのみに対するものであり，したがって変更が必要です。

- “variant_struct”と“variant_union”の使用

  新しい.H ファイルでは，“variant_struct”と“variant_union”が常に使用されます。一方，以前は一部の構造体が“struct”と“union”を使用していました。したがって，データ構造体内のフィールドを参照するときに，中間の構造体名を指定することはできません。

  たとえば，

  ```
  AlignFaultItem.PC[0] = DataPtr->afr$r_pc_data_overlay.afr$q_fault_pc[0];
  ```

  という文は，次のようになります。

  ```
  AlignFaultItem.PC[0] = DataPtr->afr$q_fault_pc[0];
  ```

- メンバのアライメント

  SYS$STARLET_C.TLB の各.H ファイルは，“#pragma member_alignment”の状態を保存し，復元します。

- 表記規則

  SYS$STARLET_C.TLB の.H ファイルは，以前は VAXCDEF.TLB で部分的にしか準拠していなかった表記規則に準拠しています。すべての定数 (#defines) は大文字の名前を使用します。すべての識別子 (ルーチン，構造体メンバなど) は小文字の名前を使用します。 VAXCDEF.TLB との違いがある場合には，互換性を維持するために以前のシンボル名も使用できますが，今後は新しい表記法に従うようにしてください。

- .H ファイルにアクセスするための Librarian ユーティリティの使用

  OpenVMS Alpha をインストールするときに， SYS$STARLET_C.TLB の内容は別の.H ファイルに抽出されません。 DEC C コンパイラは，#include 文の形式とは無関係に， SYS$STARLET_C.TLB の内部からこれらのファイルにアクセスします。個々の.H ファイルを調べる場合には，次の例に示すように， Librarian ユーティリティを使用できます。

  ```
  $ LIBRARY /EXTRACT=AFRDEF /OUTPUT=AFRDEF.H SYS$LIBRARY:SYS$STARLET_C.
  TLB
  ```

- SYS$STARLET_C.TLB に含まれている追加の.H ファイル

  STARLET ソースから作成された.H ファイルの他に， SYS$STARLET_C.TLB には， CMA.H など， DECthreads をサポートするための.H ファイルも含まれています。

11.2.6 /STANDARD=VAXC モードでサポートされないVAX Cの機能

VAX Cでサポートされる大部分のプログラミング方式は， DEC C for OpenVMS Alpha システムでも/STANDARD=VAXC モードでサポートされますが， ANSI 標準規格と矛盾する特定のプログラミング方式はサポートされません。次のリストはこれらの相違点を示しています。詳しくは DEC C コンパイラに関する解説書を参照してください。

- 次の例に示すように#endif 文の後に指定したテキスト。

  ```
  #ifdef a
     .
     .
     .
  #endif a
  ```

  テキストを削除するか，または次の例に示すようにテキストをコメント区切り文字で囲んでください。

  ```
  #endif /* a */
  ```

- 文字列定数の変更は常に問題となるプログラミング方法ですが，VAX Cでは受け付けられていました。 DEC C for OpenVMS Alpha システムでは，すべての文字列定数は読み込み専用プログラム・セクションに格納されるため，変更できません。
- 構造体を初期化する値は中括弧 ({}) で囲まなければなりません。

```
array[SIZE] = NULL;   /* accepted by VAX C */
array[SIZE] = {NULL}; /* required by DEC C */
```

- シンボルの再定義には，警告レベルの診断メッセージが示されるようになりました。

```
#define x  a
#define x  b   /* generates a warning message in DEC C */
```

- テキスト・ライブラリの使用は望ましくありません。VAX Cではサポートされましたが，テキスト・ライブラリを移植することはできません。

```
#include  stdio
```

このような場合には，かわりに次の構文を使用してください。

```
#include <stdio.h>
```

- 外部変数は 1 回だけ宣言しなければなりません。これはこの変数の定義です。他のモジュールでは，externセマンティックを使用して宣言することにより，その変数を使用できます。
- VAX Cコードを再コンパイルする場合には，アプリケーション全体を再コンパイルする場合も， 1 つ以上のモジュールを再コンパイルする場合も，外部シンボルにとくに注意する必要があります。 1 つの外部シンボル・モデルをサポートするVAX Cコンパイラと異なり， DEC C コンパイラは 4 つのモデルをサポートします。 DEC C コンパイラで生成される省略時の外部シンボルは， 1 つのVAX C外部シンボルと同じではありません。

  さらに，このようなコードをリンクする場合には，リンカが変更されているため， C コード・モジュールを再コンパイルしたときに/SHARE 修飾子を指定しなかったときは，関連するリンカ修飾子を指定しなければなりません。

11.3 VAX COBOL と DEC COBOL の互換性と移行

DEC 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。以下の項では，両者のおもな違いの要約を示します。この情報は，両製品との互換性がある COBOL アプリケーションを開発したり，VAX COBOL アプリケーションを OpenVMS Alpha オペレーティング・システム上の DEC COBOL に移行する際に役立ちます。

11.3.1 DEC COBOL の拡張仕様と機能の違い

DEC COBOL は，VAX COBOL にはない以下の言語拡張仕様と機能を含んでいます。

- コンパイル・コマンド行を使って，またはソース・ディレクティブを使って個々のレコードに対してアラインメントを選択することができます。 Alpha データ・アラインメントを選択してパフォーマンスを最適化するか，VAX COBOL のデータ・アラインメントを選択してVAX COBOL との互換性を実現することになります。
- 単精度および倍精度データ項目に IEEE または VAX 浮動小数点データ型を選択する修飾子。
- ネイティブな OpenVMS Alpha イメージがトランスレートされた VAX イメージを，またトランスレートされた VAX イメージがネイティブな OpenVMS Alpha イメージを呼び出せるコードを生成するための修飾子。
- 『X/Open Portability Guide』で定義された新たな COBOL の予約語を認識する修飾子。
- 数値の表示項目で，すべての空白をゼロに変更する修飾子。
- COMP の同義語としての COMP-5 と COMP-X。
- READ PRIOR と START LESS。
- 他の COBOL コンパイラと DEC COBOL の間でのプログラムの移植のサポート (/RESERVED_WORDS=FOREGN_EXTENSIONS)。
- X/Open の ASSIGN TO， LINE SEQUENTIAL， RETURN-CODE， SCREEN SECTION， FILE-SHARING，および RECORD-LOCKING。

さらに， DEC COBOL には以下の機能が含まれています。

- VAX COBOL の/STANDARD=V3 修飾子の機能のサブセットをサポートします。
- VAX COBOL バージョン 5.1 と互換性のあるファイル状態値をサポートします。これは，VAX COBOL のバージョン 5.0 とそれ以前のバージョンとは異なります。

DEC COBOL は，VAX COBOL の以下の機能を含んでいません。

- DECset/LSE Program Design Facility，/DESIGN 修飾子，デザイン・コメント，疑似コード・プレースホルダ。
- DEC COBOL は VFU-CHANNEL をサポートしていないので， VFU と VFP (Vertical Forms Unit ユーティリティと Vertical Forms Printing) を直接にはサポートしません。
- COPY FROM DICTIONARY

DEC COBOL とVAX COBOL の詳細については，製品のリリース・ノートおよび解説書をご覧ください。 OpenVMS Alpha オペレーティング・システムでは，システム・プロンプトで HELP COBOL RELEASE_NOTES と入力すると，インストールされているコンパイラのリリース・ノートのオンライン・バージョンを見ることができます。

11.3.2 コマンド行修飾子

第 11.3.2.3 項と第 11.3.2.5 項では， DEC COBOL とVAX COBOL のコマンド行修飾子を比較しています。 OpenVMS Alpha オペレーティング・システムの DEC COBOL のコマンド行修飾子の詳細については， OpenVMS Alpha のシステム・プロンプトでHELP COBOLと入力して，オンライン・ヘルプを参照してください。VAX COBOL のコマンド行修飾子については，『VAX COBOL User Manual』を参照してください。

11.3.2.1 /NATIONALITY={JAPAN|US}

/NATIONALITY=JAPAN が指定されていると，円記号 (¥) が省略時の通貨記号になり，日本語サポート機能が有効になります。また，この場合は/NODIAGNOSTICS と/NOANALYSIS_DATA が暗黙のうちに指定されます。

OpenVMS Alpha 上の DEC COBOL で/NATIONALITY=JAPAN を指定した場合，Oracle CDD/Repository はサポートされません。

コンパイル・コマンド行で/NATIONALITY=US を指定すると，ドル記号 ($) が省略時の通貨記号になり，日本語サポート機能は無効になります。

11.3.2.2 /STANDARD=MIA

コンパイル・コマンド行で/STANDARD=MIA が指定されていると，コンパイラは MIA 仕様に準拠していない言語要素について診断情報を発行します。

- 基本標準 (ANSI-85， JIS-88) に対する DEC の構文拡張
- 4 つのオプション・モジュールのうちの 2 つ
- 基本標準の必須モジュールに含まれる，旧式の言語要素
- ベンダのインプリメンテーションの違いのために，基本標準の必須モジュールから省略された言語要素
- 日本語に関連する MIA 拡張要素とは異なる， DEC 固有の日本語機能

診断情報を表示するには，/WARNINGS=ALL 修飾子か /WARNING=INFORMATION 修飾子が必要です。

省略時の設定は NOMIA です。

11.3.2.3 DEC COBOL 固有の修飾子

以下のコマンド行修飾子は DEC COBOL でのみ使用できます。

- /ALIGNMENT
- /CHECK=DECIMAL
- /CONVERT=LEADING_BLANKS
- /FLOAT=D_FLOAT
- /FLOAT=G_FLOAT
- /FLOAT=IEEE_FLOAT
- /OPTIMIZE[=LEVEL=n]
- /RESERVED_WORDS=FOREIGN_EXTENSIONS
- /RESERVED_WORDS=[NO]XOPEN
- /SEPARATE_COMPILATION[1]
- /TIE
- /VFC[1]

11.3.2.4 /ALIGNMENT=PADDING

『OpenVMS Calling Standard』の呼び出し規則では，データ・フィールドが (その規則で定められている) 特定のアドレスにアラインされている必要があります。これと同じ規則に，すべてのデータ・レコードは，そのアラインメントの倍数でなくてはならないと定められています。

コンパイル・コマンド行で/ALIGNMENT=PADDING が指定されていると，COBOL のグループ・データ項目はその自然な境界上にアラインされ，これらのグループ項目はアラインメントの倍数になるようにパッドが挿入されます。 Alpha のアラインメントとパッドの挿入が行われるときの基本データ項目のアラインメントの詳細については，『DEC COBOL Reference Manual』を参照してください。

11.3.2.5 VAX COBOL 固有の修飾子

表 11–6 は，VAX COBOL に固有のコマンド行修飾子と修飾子オプションの組み合わせを示しています。これらは DEC COBOL では使用できません。

表 11–6 VAX COBOL 固有の修飾子

修飾子	説明
/DESIGN	詳細設計として入力ファイルをコンパイラが処理を行かどうかを制御する。

(次ページに続く)

[1] この修飾子に対する DEC COBOL の動作は，VAX COBOL の省略時の動作に近くなるように設計されています。

表 11–6 (続き) VAX COBOL 固有の修飾子

修飾子	説明
/INSTRUCTION_SET[=option]	VAX 命令セットの異なる部分を使用して，シングル・チップ VAX プロセッサ上で実行時の性能を改良する。
/STANDARD=OPENVMS_AXP	DEC COBOL コンパイラでサポートされない言語機能に関する情報メッセージを出力する (第 11.3.2.7 項および『VAX COBOL バージョン 5.1 リリース・ノート』を参照)
/STANDARD=PDP11	COBOL–81コンパイラでサポートされない言語機能に関する情報メッセージを出力する
/WARNINGS=STANDARD	DEC による拡張機能である言語機能に関する情報メッセージを出力する。 DEC COBOL の同等な機能は/STANDARD=SYNTAX 修飾子である

11.3.2.6 /STANDARD=V3

DEC COBOL は，VAX COBOL の/STANDARD=V3 修飾子のインプリメンテーションがサポートしているいくつかの機能をサポートしていません。

- STRING， UNSTRING，および INSPECT (Format 3) 文と， DIVIDE 文の REMAINDER 句における添え字が評価されるタイミング
- STRING， UNSTRING，および INSPECT (Format 3) 文で参照変更が評価されるタイミング
- VARYING 句に関連付けられた変数の値が， PERFORM . . . VARYING . . . AFTER 文の中で増やされるタイミング (Format 4)
- 一部の移動での PIC P の数字の解釈
- MOVE 文の中で，可変長テーブル (OCCURS DEPENDING ON) のサイズが決定されるタイミング

/WARNING=ALL 修飾子を使用すると，/STANDARD=V3 の効果がわかりやすくなります。特に， DEC COBOL は，/STANDARD=V3 が指定されていると，以下の情報メッセージを生成します。

- INSPECT， STRING， UNSTRING，および DIVIDE 文の評価順序によって影響を受ける項目

```
/STANDARD=V3 evaluation order not
  supported for this construct
```

- OCCURS DEPENDING ON が MOVE 文の異なる動作を必要とするような移動先

```
/STANDARD=V3 variable length item
  rules not supported for this construct
```

/STANDARD=V3 修飾子のVAX COBOL のインプリメンテーションの詳細については，『VAX COBOL User Manual』の修飾子に関する付録を参照してください。

11.3.2.7 /STANDARD=OPENVMS_AXP

VAX COBOL バージョン 5.1(およびそれ以上) では，/STANDARD=OPENVMS_AXP 修飾子オプションを指定して，既存のVAX COBOL プログラムの中の， OpenVMS Alpha システム上の DEC COBOL では使用できない言語要素を識別する新しいフラグ付けシステムを利用することができます。

/STANDARD=OPENVMS_AXP を指定すると，VAX COBOL コンパイラは DEC COBOL では使用できない言語構成体を通知する情報メッセージを生成します (これらのメッセージを表示するためには，同時に/WARNINGS=ALL または /WARNINGS=INFORMATIONAL も指定する必要があります)。この情報を使用して， DEC COBOL を使用する前にプログラムを変更することができます。

省略時の/STANDARD=NOOPENVMS_AXP では，これらの情報メッセージは表示されません。

11.3.3 DEC COBOL とVAX COBOL の動作の違い

この項では，VAX COBOL と DEC COBOL の動作の違いと， DEC COBOL に固有の動作について説明します。

11.3.3.1 プログラム構造メッセージ

DEC COBOL コンパイラは，到達不能コードやその他のロジック・エラーに関して，VAX COBOL コンパイラよりも詳細なメッセージを生成することがあります。

次の例に，サンプル・プログラムと， DEC COBOL コンパイラが発行するメッセージを示します。

ソース・ファイル:

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. T1.
       ENVIRONMENT DIVISION.
       PROCEDURE DIVISION.
       P0.
           GO TO P1.
       P2.
           DISPLAY "This is unreachable code".
       P1.
           STOP RUN.
```

OpenVMS VAX **システム上でのメッセージ**:

```
$ COBOL /ANSI/WARNINGS=ALL T1.COB
$
```

プログラムはコンパイルされます。VAX COBOL コンパイラはメッセージを出力しません。

OpenVMS Alpha システム上でのメッセージ:

```
$ COBOL/ANSI/OPTIMIZE/WARNINGS=ALL T1.COB
       P2.
.......^
%COBOL-I-UNREACH, code can never be executed at label P2
at line number 7 in file DISK$YOURDISK:[TESTDIR]T1.COB;1
```

DEC COBOL は，どちらのオペレーティング・システム上でも最適化を行います。最適化の 1 つの用途として，呼び出されていないルーチンや到達不能な段落の分析を行うことができます。コンパイラは，/NOOPTIMIZE を含むすべての最適化レベルで，到達不能コードの分析を実行します (省略時には完全な最適化が行われるので，上の例のようにコマンド行で修飾子やフラグを指定する必要はありません)。VAX COBOL には/OPTIMIZE 修飾子はありません。

11.3.3.2 プログラム・リスティングの違い

VAX COBOL コンパイラと， OpenVMS Alpha システム上の DEC COBOL コンパイラには，出力されるプログラム・リスティングに違いがあります。

11.3.3.2.1 マシン・コード DEC COBOL では，/NOOBJECT 修飾子を指定すると，コンパイラはコード生成を抑止します。このため，リスティングでもオブジェクト・モジュールでも，マシン・コードは生成されません。

VAX COBOL では，/NOOBJECT は.OBJ の作成を抑止するだけです。この場合でも，VAX COBOL はオブジェクト・コードを生成するための作業を実行しますので，その結果をリスティングに出力することができます。

プログラム・リスティングにマシン・コードを出力したい場合は，/NOOBJECT を使用しないでください。

11.3.3.2.2 モジュール名 DEC COBOL では，最初のプログラムの名前がコンパイル全体でのモジュール名になります。VAX COBOL では，モジュール名は個々のプログラムごとに変わります。

11.3.3.2.3 COPY 文と REPLACE 文 DEC COBOL コンパイラとVAX COBOL コンパイラは， COBOL プログラムの COPY 文の注釈を，若干異なる形式で出力します。

次の 2 つのコンパイラ・リスティングは， DEC COBOL とVAX COBOL での，COBOL プログラムのリスティングにおける注釈の位置の違い (“L”という文字) を示しています。

DEC COBOL における COPY 文のリスティング・ファイル:

```
                1 IDENTIFICATION DIVISION.
                2 PROGRAM-ID. DCOP1B.
                3 *
                4 *       This program tests the copy library file.
                5 *       with a comment in the middle of it.
                6 *       It should not produce any diagnostics.
                7         COPY
                8 *       this is the comment in the middle
                9                 LCOP1A.
L              10 ENVIRONMENT DIVISION.
L              11 INPUT-OUTPUT SECTION.
L              12 FILE-CONTROL.
L              13 SELECT FILE-1
L              14         ASSIGN TO "FILE1.TMP".
               15 DATA DIVISION.
               16 FILE SECTION.
               17 FD      FILE-1.
               18 01      FILE1-REC       PIC X.
               19 WORKING-STORAGE SECTION.
               20 PROCEDURE DIVISION.
               21 PE.     DISPLAY "***END***"
               22         STOP RUN.
```

VAX COBOL における COPY 文のリスティング・ファイル:

```
    1          IDENTIFICATION DIVISION.
    2          PROGRAM-ID. DCOP1B.
    3         *
    4         *    This program tests the copy library file.
    5         *    with a comment in the middle of it.
    6         *    It should not produce any diagnostics.
    7              COPY
    8         *    this is the comment in the middle
    9                      LCOP1A.
   10L         ENVIRONMENT DIVISION.
   11L         INPUT-OUTPUT SECTION.
   12L         FILE-CONTROL.
   13L         SELECT FILE-1
   14L             ASSIGN TO "FILE1.TMP".
   15          DATA DIVISION.
   16          FILE SECTION.
   17          FD  FILE-1.
   18          01  FILE1-REC       PIC X.
   19          WORKING-STORAGE SECTION.
   20          PROCEDURE DIVISION.
   21          PE. DISPLAY "***END***"
   22              STOP RUN.
```

11.3.3.2.4 複数の COPY 文

DEC COBOL コンパイラとVAX COBOL コンパイラは，複数の COPY 文が同じ行に含まれている COBOL プログラムのリスティングでも，若干異なる形式を使用します。

次の 2 つのコンパイラ・リスティングは， DEC COBOL とVAX COBOL での，COBOL プログラムの 1 つの行に複数の COPY 文があったときの注釈の位置の違い (“L”という文字) を示しています。

DEC COBOL における複数の COPY 文のリスティング・ファイル:

```
                1 IDENTIFICATION DIVISION.
                2 PROGRAM-ID. DCOP1J.
                3 *
                4 *       Tests copy with three copy statements on 1 line.
                5 *
                6 ENVIRONMENT DIVISION.
                7 DATA DIVISION.
                8 PROCEDURE DIVISION.
                9 THE.
               10         COPY LCOP1J. COPY LCOP1J. COPY LCOP1J.
L              11         DISPLAY "POIUYTREWQ".
L              12         DISPLAY "POIUYTREWQ".
L              13         DISPLAY "POIUYTREWQ".
               14         STOP RUN.
```

VAX COBOL における複数の COPY 文のリスティング・ファイル:

```
    1          IDENTIFICATION DIVISION.
    2          PROGRAM-ID. DCOP1J.
    3         *
    4         *    Tests copy with three copy statements on 1 line.
    5         *
    6          ENVIRONMENT DIVISION.
    7          DATA DIVISION.
    8          PROCEDURE DIVISION.
    9          THE.
   10              COPY LCOP1J.
   11L             DISPLAY "POIUYTREWQ".
   12C                         COPY LCOP1J.
   13L             DISPLAY "POIUYTREWQ".
   14C                                     COPY LCOP1J.
   15L             DISPLAY "POIUYTREWQ".
   16              STOP RUN.
```

11.3.3.2.5 COPY 挿入文

COPY 文が行の途中に文を挿入するときのコンパイラ・リスティング・ファイルは，DEC COBOL プログラムとVAX COBOL プログラムで異なります。

次の 2 つのコンパイラ・リスティングで，LCOP5D.LIB は“O”というテキストを含んでいます。DEC COBOL コンパイラは行をそのまま残し，COPY ファイルの内容をソース行の下に出力します。VAX COBOL コンパイラは，元のソース行を 2 つの部分に分割します。

DEC COBOL における COPY 文のリスティング・ファイル:

```
------------------------------------------------------------
   13 P0.     MOVE COPY LCOP5D. TO ALPHA.
L  14              "O"
```

VAX COBOL における COPY 文のリスティング・ファイル:

```
------------------------------------------------------------
13         P0. MOVE COPY LCOP5D.
14L                "O"
15C                              TO ALPHA.
```

11.3.3.2.6 REPLACE 文

COBOL ソース文の REPLACE と DATE-COMPILED に対する診断メッセージにより，コンパイラ・リスティング・ファイルにソース行の複数のインスタンスが含まれることになります。

DEC COBOL プログラムの REPLACE 文では，DEC COBOL コンパイラが置換テキストに関するメッセージを発行する場合，そのメッセージは次のコンパイラ・リスティング・ファイルに示すように，プログラムの元のテキストに対応するものになります。

DEC COBOL における REPLACE 文のリスティング・ファイル:

```
                18 P0.     REPLACE ==xyzpdqnothere==
                19                         BY ==nothere==.
                20
                21         copy "drep3hlib".
L               22         display xyzpdqnothere.
                ...................1
     %COBOL-F-UNDEFSYM, (1) Undefined name

LR              22     display nothere.
```

VAX COBOL プログラムでは，コンパイラ・メッセージは次のコンパイラ・リスティング・ファイルに示すように，置換後のテキストに対応するものになります。

VAX COBOL における REPLACE 文のリスティング・ファイル:

```
        18         P0. REPLACE ==xyzpdqnothere==
        19                              BY ==nothere==.
        20
        21             copy "drep3hlib".
        22LR           display nothere.
                              1
             %COBOL-F-ERROR  349, (1) Undefined name
```

11.3.3.2.7 DATE COMPILED 文　次の 2 つのコンパイラ・リスティング・ファイルは，DEC COBOL とVAX COBOL で DATE-COMPILED 文を使用した場合の違いを示しています。

DEC COBOL における DATE-COMPILED 文のリスティング・ファイル:

```
            33 *
            34 date-compiled
               ............1
%COBOL-E-NODOT, (1) Missing period is assumed
            34 date-compiled  16-Jul-1992.
            35 security. none.
```

VAX COBOL における DATE-COMPILED 文のリスティング・ファイル:

```
   33        *
   34         date-compiled   16-Jul-1992.
                         1
%COBOL-E-ERROR   65, (1) Missing period is assumed
   35         security. none.
```

REPLACE 文と COPY REPLACING 文を使用したとき，コンパイラ・リスティング・ファイルにおける行番号は DEC COBOL とVAX COBOL で異なります。 DEC COBOL は，置換後の行番号を元のソース・テキストの行番号に対応させます。このため，それ以降の行番号は食い違うことになります。VAX COBOL は行番号を連続的に付けます。

次のソース・プログラムは，プログラムを DEC COBOL とVAX COBOL のどちらでコンパイルするかによって，コンパイラ・リスティング・ファイルの最後の行番号が変わります。

ソース・ファイル:

```
              REPLACE ==A VERY LONG STATEMENT== by ==EXIT PROGRAM==.
              A
              VERY
              LONG
              STATEMENT.
              DISPLAY "To REPLACE or not to REPLACE".
```

DEC COBOL における REPLACE 文のリスティング・ファイル:

```
------------------------------------------------------------
        1 REPLACE ==A VERY LONG STATEMENT== by ==EXIT PROGRAM==.
        2 EXIT PROGRAM.
        6 DISPLAY "To REPLACE or not to REPLACE".
```

VAX COBOL における REPLACE 文のリスティング・ファイル:

```
------------------------------------------------------------
        1 REPLACE ==A VERY LONG STATEMENT== by ==EXIT PROGRAM==.
        2 EXIT PROGRAM.
        3 DISPLAY "To REPLACE or not to REPLACE".
```

11.3.3.2.8 コンパイラ・リスティングと分割コンパイル

/SEPARATE_COMPILATION 修飾子を指定すると，リスティングが個別に生成されます。/SEPARATE_COMPILATION を**指定せずに**コンパイルされた分割コンパイル・プログラム (SCP) では，リスティングは以下のようになります。

- PROGRAM_1 のソース・リスティング
- PROGRAM_2 のソース・リスティング
- PROGRAM_3 のソース・リスティング
- PROGRAM_1 のマシン・コード・リスティング
- PROGRAM_2 のマシン・コード・リスティング
- PROGRAM_3 のマシン・コード・リスティング

/SEPARATE_COMPILATION を指定したときの結果は以下のようになります。

- PROGRAM_1 のソース・リスティング
- PROGRAM_1 のマシン・コード・リスティング
- PROGRAM_2 のソース・リスティング
- PROGRAM_2 のマシン・コード・リスティング
- PROGRAM_3 のソース・リスティング
- PROGRAM_3 のマシン・コード・リスティング

これはVAX COBOL が生成するリスティングと同じであることに注意してください。

11.3.3.3 出力のフォーマッティング

VFU-CHANNEL

DEC COBOL は VFU-CHANNEL をサポートしていないので， VFU と VFP (Vertical Forms Unit ユーティリティと Vertical Forms Printing) を直接にはサポートしません。

制御バイト・シーケンス
DEC COBOL とVAX COBOL は，似たような出力ファイル・フォーマッティングを実現するために， VFC ファイルの中で異なる制御バイト・シーケンスを使用しなければならないことがあります。

画面のフォーマッティング
DEC COBOL とVAX COBOL は，似たような画面フォーマッティングを実現するために， ACCEPT と DISPLAY で異なるエスケープ・シーケンスを使用しなければならないことがあります。

VFC ファイル
VFC フォーマットの REPORT WRITER または LINAGE ファイルは，通常は TYPE コマンドを使用するか，プリントアウトして表示します。電子メールで送信したり，エディタで読み込んだりするためには，コンパイル・コマンド行で/NOVFC を指定してコンパイルを行います。

1 つの.COB ソース・ファイルでオープンされるすべての REPORT WRITER および LINAGE ファイルは，同じフォーマットを持つことになります (VFC または NOVFC)。省略時は VFC です。/NOVFC 修飾子が指定されていると，各ソース・ファイルに対して NOVFC 条件が設定されます。次に例を示します。

```
$ COBOL A/NOVFC,B/VFC,C/NOVFC,D
```

この例で，ソース・ファイル B と D は VFC フォーマットでレポートを生成します (ソース・ファイル・リスト項目がプラス(+)記号で区切られている場合には動作が異なります)。

11.3.3.4 DEC COBOL とVAX COBOL の文の違い

以下の COBOL 文は， DEC COBOL とVAX COBOL で異なる動作をします。

- ACCEPT
- DISPLAY
- EXIT PROGRAM
- LINAGE
- MOVE
- SEARCH

11.3.3.4.1 ACCEPT および DISPLAY 文

プログラムの中で ACCEPT または DISPLAY の拡張機能を使用していると， DEC COBOL とVAX COBOL の動作に違いが生じることがあります。この場合， DEC COBOL は以下のように動作します。

- 1 つのプログラムの中で ANSI ACCEPT 文と拡張 ACCEPT 文を混在させると，拡張 ACCEPT 文が使用する編集キーが， ANSI ACCEPT 文でも使用されます
- VAX COBOL は，画面管理での VT52 端末をサポートしていますが， DEC COBOL は**サポートしていません**。

- 端末がラップなしモードに設定されているときに，文字が画面の終わりを越える項目を表示すると，一番右の列以降の文字はすべて切り捨てられます。たとえば， 80 カラムの画面の 79 カラム目に“1234”を表示するように指定すると， DEC COBOL は“12”と表示します。一方，VAX COBOL は一番右のカラムの文字を上書きします。この例では，VAX COBOL は“14”と表示します。
- アプリケーションが ACCEPT または DISPLAY 文の日本 DEC による拡張機能を使用している場合， DEC COBOL は最初の ACCEPT または DISPLAY 文を実行する前に，画面の左上のコーナーにカーソルを置きます。

  この違いは，最初の ACCEPT または DISPLAY 文が LINE 句と COLUMN 句を含んでいないときに明確になります。この場合， DEC COBOL はカーソルを画面の一番上に移動してから ACCEPT または DISPLAY を実行しますが，VAX COBOL はカーソルを移動しません。

11.3.3.4.2 EXIT PROGRAM 句　呼び出されたプログラムの中の EXIT PROGRAM は PERFORM 範囲をリセットしません。VAX COBOL では， EXIT PROGRAM で脱出した後にプログラムに再び入ったとき，以前の実行におけるすべての PERFORM 範囲が満たされています。

11.3.3.4.3 LINAGE 句　DEC COBOL とVAX COBOL では， LINAGE 文で大きな値を扱ったときの動作が異なります。 WRITE 文の ADVANCING 句の行カウントが 127 よりも大きいと， DEC COBOL は 1 行進みますが，VAX COBOL での結果は定義されていません。

11.3.3.4.4 MOVE 文　符号なし計算フィールドは，符号付き計算フィールドよりも大きな値を含むことができます。 ANSI COBOL 規格に従い，符号なし項目の値は必ず正の値として扱われなくてはなりません。しかし， DEC COBOL は符号なし項目を正の値として扱いますが，VAX COBOL はこれを符号付き項目として扱います。このため，稀ではありますが， MOVE 文や算術文の中で符号なしデータ項目と符号付きデータ項目を混在させたときに，VAX COBOL と DEC COBOL で異なる結果が得られることがあります。

例 11–1 では，VAX COBOL と DEC COBOL で異なる値が得られます。

VAX COBOL の結果:

```
B1 =  -1
B2 =  -1
```

DEC COBOL の結果:

```
B1 =  65535
B2 =  65535
```

例 11–1 符号付きと符号なしの違い

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SHOW-DIFF.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 A2       PIC 99    COMP.
01 B1       PIC S9(5) COMP.
01 B2       PIC 9(5)  COMP.
PROCEDURE DIVISION.
TEST-1.
    MOVE 65535 TO A2.
    MOVE A2 TO B1.
    DISPLAY B1 WITH CONVERSION.
    MOVE A2 TO B2.
    DISPLAY B2 WITH CONVERSION.
    STOP RUN.
```

11.3.3.4.5 SEARCH 文

DEC COBOL とVAX COBOL バージョン 5.0 以上では，SEARCH 文の中で END-SEARCH 句と NEXT SENTENCE 句を同時に使用することはできません。一方を使用すると，もう一方は使用できません。この規則は ANSI COBOL 規格に則ったものですが，バージョン 5.0 よりも前のVAX COBOL には適用されません。

11.3.3.5 システムの戻りコード

例 11–2 は， DEC COBOL とVAX COBOL で異なる動作をする，規則に反するコーディングを示しています。この動作の違いは， VAX アーキテクチャと Alpha アーキテクチャの，レジスタ・セットにおけるアーキテクチャ上の違いが原因です。 Alpha には浮動小数点データ型専用のレジスタのセットがあります。

例 11–2 に示されているようなコーディング違反は， Alpha がサポートしているどの浮動小数点データ型にも影響を及ぼします。

例 11–2 戻り値の誤ったコーディング

```
IDENTIFICATION DIVISION.
PROGRAM-ID. BADCODING.
ENVIRONMENT DIVISION.

DATA DIVISION.
FILE SECTION.

WORKING-STORAGE SECTION.

    01  FIELDS-NEEDED.
        05 CYCLE-LOGICAL      PIC X(14) VALUE 'A_LOGICAL_NAME'.

    01  EDIT-PARM.
        05  EDIT-YR           PIC X(4).
        05  EDIT-MO           PIC XX.

    01  CMR-RETURN-CODE       COMP-1 VALUE 0.


LINKAGE SECTION.

       01 PARM-REC.
          05 CYCLE-PARM        PIC X(6).
          05 RETURN-CODE       COMP-1 VALUE 0.

PROCEDURE DIVISION USING PARM-REC GIVING CMR-RETURN-CODE.

P0-CONTROL.

    CALL 'LIB$SYS_TRNLOG' USING BY DESCRIPTOR CYCLE-LOGICAL,
                                OMITTED,
                                BY DESCRIPTOR CYCLE-PARM
                                GIVING RETURN-CODE.

    IF  RETURN-CODE  GREATER 0
      THEN
        MOVE RETURN-CODE TO CMR-RETURN-CODE
        GO TO P0-EXIT.

    MOVE  CYCLE-PARM  TO  EDIT-PARM.

    IF  EDIT-YR  NOT  NUMERIC
      THEN
        MOVE  4  TO CMR-RETURN-CODE, RETURN-CODE.

    IF  EDIT-MO  NOT NUMERIC
      THEN
        MOVE  4  TO  CMR-RETURN-CODE, RETURN-CODE.


    IF  CMR-RETURN-CODE GREATER 0
                OR
        RETURN-CODE GREATER 0
      THEN
        DISPLAY "**************************"
        DISPLAY "** BADCODING.COB  **"
        DISPLAY "** A_LOGICAL_NAME> ", CYCLE-PARM, "   **"
        DISPLAY "**************************".

P0-EXIT.
```

(次ページに続く)

例 11–2 (続き) 戻り値の誤ったコーディング

```
    EXIT PROGRAM.
```

例 11–2 では，システム・サービス呼び出しの戻り値が，実際にはバイナリ (COMP) でなければならないのに誤って F 浮動小数点値と定義されています。このプログラマは，ルーチンからのすべての戻り値はレジスタ R0 に返されるという VAX アーキテクチャの動作を利用していました。 VAX アーキテクチャでは，整数と浮動小数点数のレジスタが別々に存在しているわけではありません。一方， Alpha アーキテクチャは，浮動小数点データとバイナリ・データについて，別々のレジスタ・セットを定義しています。浮動小数点値を返すルーチンはレジスタ F0 を使用し，バイナリ値を返すルーチンはレジスタ R0 を使用します。

DEC COBOL コンパイラには，外部ルーチンが返すデータ型を知る手段はありません。プログラマは， CALL 文の GIVING-VALUE 項目に対して，正しいデータ型を指定する必要があります。 Alpha アーキテクチャでは，浮動小数点データ項目に異なるレジスタのセットが使用されるため，生成されるコードは R0 ではなく F0 をテストします。

サンプル・プログラムのコード・シーケンスにおける F0 の値は予測できません。このコーディングでも期待される動作が得られることがありますが，ほとんどの場合は失敗します。

11.3.3.6 診断メッセージ

プログラミングに使用しているプラットフォームによって，いくつかの診断メッセージは異なる意味と結果を持っています。

- DEC COBOL は，次の診断の受信時に，VAX COBOL と同じランタイム・エラー回復動作を行いません。

  ```
  %COBOL-E-EXITDECL, EXIT PROGRAM statement invalid in
    GLOBAL DECLARATIVE
  ```

- VAX COBOL は GLOBAL USE プロシージャの中の EXIT PROGRAM をつねに無視します。 DEC COBOL は， GLOBAL USE が現在のプログラム単位とは別のプログラム単位から呼び出されている場合にのみ EXIT PROGRAM を無視します。

  VAX COBOL と同じ動作をさせるためには，診断の原因となった問題を修正してください。

- 比較に使われているオペランドのいずれかが不正であると，VAX COBOL と DEC COBOL はどちらもエラー・メッセージを発行します。VAX COBOL はこの条件を含んでいる文の分析を続けますが， DEC COBOL は次の文に進みます (このため，その文の中のエラーはこれ以上発見されません)。

- ソース文に複数の除算が含まれており，分母がリテラルのゼロ，数値としてのゼロ，または値がゼロの変数である場合， DEC COBOL はゼロによる除算のランタイム診断を 1 つだけ発行しますが，VAX COBOL は文の中のゼロによる除算すべてについて同じ診断を発行します。たとえば，次のコードでは，VAX COBOL では 3 つの診断が， DEC COBOL では 1 つの診断が発行されます。

  ```
  DIVIDE 0 INTO A, B, C.
  ```

  ANSI COBOL 規格に従い，どちらのコンパイラも予期しない結果を生じないで実行を続けることができます。

11.3.3.7 倍精度データ項目の記憶形式

VAX アーキテクチャと Alpha アーキテクチャでの D 浮動小数点データの記憶形式の違いにより，実行結果の評価の際に，若干異なる答えが得られます。この差は，最終的な結果を出力するまで何回 D-float の演算を行うかに依存します。これは，COMP-2 型のデータをファイルに出力し， OpenVMS Alpha システム上で動作するプログラムが生成する出力を OpenVMS VAX システムが生成する出力と比較しようとした場合に問題になることがあります。

浮動小数点データ型の記憶形式の詳細については，『Alpha Architecture Handbook』を参照してください。

11.3.3.8 データ項目のハイオーダー切り捨て

DEC COBOL はVAX COBOL よりも，データ項目のハイオーダー切り捨てが生じる可能性に敏感に反応します。次の例を/WARNINGS=ALL修飾子を指定してコンパイルした場合を考えます。

```
WORKING-STORAGE SECTION.
01 K4 PIC 9(9) COMP.

PROCEDURE DIVISION.
01-MAIN-SECTION SECTION.
01-MAIN.
    DISPLAY K4 WITH CONVERSION.
```

DEC COBOL は，VAX COBOL とは異なり，次のメッセージを出力します。

```
    DISPLAY K4 WITH CONVERSION.
    ............^
    Possible high-order truncation ...
```

RELATIVE ファイル操作でも，この診断が生成されることがあります。

11.3.3.9 ファイルの状態値

ファイルを EXTEND モードでオープンし，これを REWRITE しようとすると，DEC COBOL とVAX COBOL は異なったファイル状態値を返します。この未定義の操作に対し， DEC COBOL はファイル状態 49 (互換性のないオープン・モード) を返し，VAX COBOL はファイル状態 43 (対応する READ 文がない) を返します。

11.3.3.10 参照キー

OpenVMS Alpha では，ISAM ファイルに動的にアクセスする場合，参照キーがセカンダリ・キーであると，WRITE，DELETE，または REWRITE を実行したときに，参照キーがセカンダリ・キーからプライマリ・キーに変更されます。

11.3.3.11 RMS 特殊レジスタ

プラットフォームによって，RMS 特殊レジスタの動作が変わることがあります。

ロードの違い

実行時に，DEC COBOL とVAX COBOL は，一部の入出力操作で RMS 特殊レジスタの値を違った形で更新します。DEC COBOL のランタイム・システムは，RMS 操作を試みる前に，いくつかの入出力エラー条件のチェックを行います。エラー条件が発生していると，DEC COBOL のランタイム・システムは RMS 操作を試みず，RMS 特殊レジスタは以前の値をそのまま持ち続けます。VAX COBOL のランタイム・システムは，入出力操作のチェックを行わずに，すべての RMS 操作を実行します。このため，ランタイム・システムは入出力操作のたびに，RMS 特殊レジスタの値を必ず更新します。

たとえば，ファイルのオープンに成功しなかった場合，それ以降の DEC COBOL レコード操作 (READ，WRITE，START，DELETE，REWRITE，または UNLOCK) は RMS を呼び出すことなく失敗します。このため，OPEN 操作の失敗の際に RMS 特殊レジスタに格納された値は，同じファイルに対してレコード操作の失敗を重ねても，変更されずに残っています。これと同じ操作をVAX COBOL で行うと，つねに RMS が呼び出されます。これにより，RMS は未定義の操作を試み，RMS 特殊レジスタに新しい値を返します。

RMS 特殊レジスタが DEC COBOL アプリケーションとVAX COBOL アプリケーションで異なる値を持つ場合がもう 1 つあります。DEC COBOL ファイルに対する RMS 操作が成功すると，RMS 特殊レジスタは必ず RMS 完了コードを含んでいます。VAX COBOL ファイルに対する RMS 操作が成功すると，RMS 特殊レジスタは通常は RMS 完了コードを含んでいますが，COBOL 固有の完了コードを含んでいることもあります。

VAX COBOL との違い

DEC COBOL では，VAX COBOL とは異なり，以下に示すコンパイラ生成変数をユーザ変数として宣言することはできません。

```
RMS_STS
RMS_STV
RMS_CURRENT_STS
RMS_CURRENT_STV
```

11.3.3.12 共用可能イメージの呼び出し

DEC COBOL とVAX COBOL は，共用可能イメージとしてインストールされた副プログラムを呼び出すときの動作が異なります。 DEC COBOL では， CALL 文で指定するプログラム名ではリテラルまたはデータ名が使用できます (CANCEL 文でも同じです)。VAX COBOL では， CALL 文 (または CANCEL 文) で指定するプログラム名はリテラルでなくてはなりません。また，共用可能イメージとしてインストールされたVAX COBOL プログラムは，外部ファイルを含むことができません (共用可能イメージの詳細については，『OpenVMS Linker Utility Manual』を参照してください)。

11.3.3.13 共通ブロックの共用

複数のプロセスで共通ブロックを共用する場合には， DEC COBOL プログラムをリンクするときの問題を防ぐために， PSECT 属性を SHR に設定します (省略時の値は， OpenVMS Alpha システムでは SHR， OpenVMS VAX システムでは NOSHR です)。また，次のように，共用可能イメージのリンカ・オプション・ファイルに SYMBOL_VECTOR を追加します。

```
SYMBOL_VECTOR = (psect-name = PSECT)
```

詳細については，『OpenVMS Linker Utility Manual』を参照してください。

11.3.3.14 算術演算

DEC COBOL とVAX COBOL では，プラットフォームによって，いくつかの算術演算の動作が異なります。

- 数値および整数の組み込み関数の結果は，最下位の桁がVAX COBOL と異なることがあります。また， DISPLAY 文によるフォーマットも異なることがあります。
- OpenVMS VAX と OpenVMS Alpha は COMP-2 項目を異なる方法で扱います。このため， USAGE COMP-2 データ項目の下位の桁を DISPLAY で表示したときの結果は， OpenVMS VAX 上のVAX COBOL でデータ項目の下位の桁を DISPLAY で表示したときと異なることがあります。
- DEC COBOL は，VAX COBOL よりも多くの状況で ALL_LOST(すべての桁が失われた) 警告診断を発行します。
- ANSI COBOL 規格は， SIZE ERROR 句のない算術文でオーバフローが生じたときの結果は予測不可能と定めています。VAX COBOL は，このような場合でも，一般に予想される下位の桁を返しますが， DEC COBOL は返しません。
- 中間結果の精度がVAX COBOL と DEC COBOL では異なります。これは除算が含まれた COMPUTE 操作でよく生じます。中間結果で特定の精度が必要な場合は，希望の精度を持つ一時変数を使用するようにしてください。次に例を示します。

```
COMPUTE D = (A / B) / C.
```

. . . は次のように書くことができます。

```
COMPUTE TMP1 = A / B.
COMPUTE D = TMP1 / C.
```

A/Bという計算に使用される精度は，TMP1 の宣言によって決定されます。

- VAX COBOL と DEC COBOL では，無効な 10 進データが与えられたときの数値比較の結果は定義されていません。DEC COBOL には，無効な 10 進データの詳細な分析を行うための/CHECK=DECIMAL と-check decimalがあります。これらの機能は，プログラムの DEC COBOL への移行を行うときに特に便利です。

11.3.4 言語とプラットフォームの間でのファイルの互換性

別のプログラミング言語で作成されたファイルは，言語と文字セットの互換性がないために，特殊な処理が必要な場合があります。最もよく見られる互換性上の問題は，データ型とデータ・レコードのフォーマットです。以下の点に注意してください。

- OpenVMS Alpha では，COBOL プログラムの中で，ファイル・レコード長よりも短い長さを指定した FD で固定長レコードのファイルを記述することができます。OpenVMS Alpha では，入力時に各レコードの余分なデータは無視されます。
- OpenVMS Alpha では，既存の ORGANIZATION INDEXED ファイルは，ORGANIZATION SEQUENTIAL を指定した FD では読み込むことができません。

データ型の違い

データ型はプログラミング言語およびユーティリティによって異なります。たとえば，DEC Fortran はパックされた 10 進データ型をサポートしていないので，COBOL ファイルの中の PACKED-DECIMAL データを簡単には利用できません

データ型の互換性の問題を回避するためには，以下のテクニックが利用できます。

- 複数の言語で使用することを前提としたデータ・ファイルには，ASCII 表現を使用する NATIVE 文字セットを使用します。
- 非 ASCII データを処理する必要がある場合は，(1) 環境部の SPECIAL-NAMES 段落と，(2) SELECT 文の CODE-SET 句で文字セットを指定することができます。NATIVE を除き，すべての文字セットは SPECIAL-NAMES 段落で設定する必要があります。
- 一般的な数値データ型を使用します (どのアプリケーションでも一定の数値データ型)。

次の例では，入力ファイルは EBCDIC で作成されています。この場合，OpenVMS Alpha オペレーティング・システム上では，COBOL 以外のどの言語でも処理が難しいファイルが作成されてしまいます。

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.   ALPHABET FOREIGN-CODE IS EBCDIC.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
     SELECT INPUT-FILE ASSIGN TO "INPFIL"
            CODE-SET IS FOREIGN-CODE.
            .
            .
            .
```

11.3.5 予約語

以下に，DEC COBOL には含まれているが，VAX COBOL には含まれていない X/OPEN の予約語を示します。

AUTO	EXCLUSIVE	REQUIRED
AUTOMATIC	FOREGROUND-COLOR	RETURN-CODE
BACKGROUND-COLOR	FULL	REVERSE-VIDEO
BLINK	HIGHLIGHT	SECURE
EOL	LOWLIGHT	UNDERLINE
EOS	MANUAL	

コマンド行修飾子/RESERVED_WORDS=NOXOPEN を指定すると，これらの予約語は予約語でない単語として扱われます。

11.3.6 デバッガ・サポートの違い

DEC COBOL のデバッガ・サポートは，VAX COBOL といくつかの点で異なります。

- DEC COBOL は，省略時の最適化が行われているときに，COBOL コマンド行で/DEBUG 修飾子を使用すると，次の情報メッセージを発行します。

  ```
  %COBOL-I-DEBUGOPT, /NOOPTIMIZE is recommended with /DEBUG
  ```

  このメッセージは，/DEBUG を指定したときに，最適化に関する指定を何も行っていない場合に表示されます (コンパイラの省略時の設定は/OPTIMIZE です)。省略時にはオフになっている他の情報メッセージとは異なり，DEC COBOL コンパイラでは，このメッセージは/WARN=NOINFO が有効になっているときでもつねに表示されます。メッセージをオフにするには，COBOL コマンド行上で修飾子/[NO]OPTIMIZE を何らかの形式で指定します (/NOOPTIMIZE, /OPTIMIZE，/OPTIMIZE=LEVEL=x など)。
- VAX COBOL には/OPTIMIZE 修飾子はありません。
- DEC COBOL では，VAX COBOL とは異なり，デバッガが変数名の中の下線をハイフンに，ハイフンを下線に変えることがあります。

このVAX COBOL との違いは，プログラムのデバッグに役立つことがあります。これらのメッセージは情報メッセージなので，コンパイラはリンクと実行が可能なオブジェクト・ファイルを生成します。ただし，このメッセージが表示されたときには，プログラムの構造が意図と異なるものになっていると思われるので，検出されないロジック・エラーが存在している可能性が高いといえます。

11.3.7 DECset/LSE サポートの違い

DEC COBOL は DECset/LSE Program Design Facility，/DESIGN 修飾子，デザイン・コメント，および疑似コード・プレースホルダをサポートしていません。

11.3.8 DBMS サポート

DEC COBOL の Oracle DBMS サポートには，プログラムを DEC COBOL とVAX COBOL のどちらで開発しているかに応じて，いくつかの違いがあります。

DEC COBOL でマルチストリーム Oracle DBMS DML を使用する際には，異なるソース・ファイルからは異なるスキーマまたはストリームにアクセスする必要があります。

11.4 Digital Fortran for OpenVMS Alpha と OpenVMS VAX システムとの互換性

この節では， Digital Fortran for OpenVMS Alpha システムと Digital Fortran 77 for OpenVMS VAX Systems (以前の名前はVAX FORTRAN) の互換性について，次の分野に分けて説明します。

- 言語機能 (第 11.4.1 項)
- コマンド行修飾子 (第 11.4.2 項)
- トランスレートされた共有可能イメージとの相互操作性 (第 11.4.3 項)
- Digital Fortran 77 for OpenVMS VAX Systems データの移植 (第 11.4.4 項)

11.4.1 言語機能

Digital Fortran for OpenVMS Alpha 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，Digital Fortran 77 for OpenVMS VAX Systems の拡張機能も含まれています。たとえば，次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, COMPLEX*8, COMPLEX*16
- IMPLICIT NONE 文
- INCLUDE 文
- NAMELIST 入出力
- ドル記号($)とアンダースコア(_)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 言語参照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の詳細については, Fortran 言語の参照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。

注意

Digital Fortran for OpenVMS では，DEC Fortran for OpenVMS Alpha と ISO/ANSI Fortran 90 標準でサポートされるFORTRAN–77言語の拡張機能の大部分がサポートされます。互換性については，『DEC Fortran 90 User Manual for OpenVMS Alpha Systems』を参照してください。

この節のこの後の部分では, Digital Fortran 77 for OpenVMS VAX Systems と Digital Fortran for OpenVMS Alpha 固有の言語機能，各言語で共用されるものの，異なる方法で解釈される言語機能, Digital Fortran 77 for OpenVMS VAX Systems には適用されない Digital Fortran for OpenVMS Alpha 制限事項，データを移植する場合の検討事項について説明します。

11.4.1.1 Digital Fortran for OpenVMS Alpha 固有の言語機能

次の言語機能は Digital Fortran for OpenVMS Alpha では提供されますが，Digital Fortran 77 for OpenVMS VAX Systems のバージョン 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引用符(")。これは/VMS 修飾子を指定することにより禁止できます。
- 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
- S 浮動小数点および T 浮動小数点 IEEE 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・フォーマットのサポート。ビッグ・エンディアン数値フォーマットもサポートされます。Alpha システムのネイティブな浮動小数点データ型についての説明は『Alpha Architecture Reference Manual』を参照してください。
- LIB$ESTABLISH と LIB$REVERT は，Digital Fortran 77 for OpenVMS VAX Systems 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。

  Digital Fortran は LIB$ESTABLISH を DEC Fortran RTL 固有のエントリ・ポイントにするために宣言を変換します。
- 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，平方根倍精度組み込み関数は CDSQRT または ZSQRT と指定できます。)
- 次の組み込み関数

  IMAG
  AND
  OR
  XOR
  LSHIFT
  RSHIFT
- いくつかの実行時エラーは Digital Fortran for OpenVMS Alpha 固有のエラーです。
- 大文字と小文字を区別する名前
- Digital Fortran for OpenVMS Alpha 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。Digital Fortran 77 for OpenVMS VAX Systems では，入出力ユニット番号の値は 0 ～ 99 の範囲です。

注意

Digital 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が Digital Fortran 77 では使用できません。

Digital Fortran 言語機能についての説明は，『Fortran language reference manual』を参照してください。

11.4.1.2 Digital Fortran 77 for OpenVMS VAX Systems 固有の言語機能

次の言語機能は Digital Fortran 77 for OpenVMS VAX Systems では使用できますが， Digital Fortran for OpenVMS Alpha ではサポートされません。

- FORTRAN/PARALLEL=(AUTOMATIC) の自動分析機能
- CPAR$の使用による FORTRAN/PARALLEL=(MANUAL)
  (たとえば， CPAR$ 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 (Digital Fortran for OpenVMS Alpha では DEFINE FILE 文をサポートしますが，同時に DEFINE FILE 文の変換についても考慮しなければなりません)。

  ERRSET および ERRTST のかわりに OpenVMS の条件処理を使用できます。 Digital 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。

Digital Fortran 77 for OpenVMS VAX Systems の特定の機能は， Digital Fortran for OpenVMS Alpha では使用が制限されているか，またはまったく提供されません。

- 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，常にではなく時々， 0 に初期化されます。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 (Digital Fortran 77 for OpenVMS VAX Systems 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。

次の言語機能は Digital Fortran 77 for OpenVMS VAX Systems では使用できますが， Alpha アーキテクチャと VAX アーキテクチャとの違いにより， Digital Fortran for OpenVMS Alpha 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 Alpha 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， OpenVMS VAX システムと OpenVMS Alpha システムとで異なります。正確な例外ハンドラ制御の要求には，/SYNCHRONOUS_EXCEPTIONS 修飾子を使用してください。
- 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， Alpha 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

  Alpha の命令セットでは， D 浮動小数点 REAL*8 フォーマット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 G 浮動小数点に変換され，その後， D 浮動小数点フォーマットに戻されます。したがって， VAX システムと Alpha システムとの間には， D 浮動小数点の演算に違いがあります。

  Alpha システムで最適な性能を実現するには， VAX G 浮動小数点または IEEE T 浮動小数点フォーマットでの REAL*8 データの使用を考慮する必要があり，おそらくフォーマットを指定するために/FLOAT 修飾子を使用します。 D 浮動小数点データを G 浮動小数点データまたは T 浮動小数点フォーマットに変換するための Digital Fortran for OpenVMS Alpha アプリケーション・プログラムを作成するには，『DEC Fortran Language Reference Manual』で説明す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 ディレクティブも含めてサポートされません。 Alpha プロセッサは，ベクタ化機能に類似した機能としてパイプライン機能や他の機能を提供します。

11.4.1.3 解釈方法の相違

次の言語機能は，Digital Fortran 77 for OpenVMS VAX Systems と Digital Fortran for OpenVMS Alpha とで異なる方法で解釈されます。

- 乱数ジェネレータ (RAN)

  Digital Fortran for OpenVMS Alpha の RAN 関数は，同じランダム・シードに対して Digital Fortran 77 for OpenVMS VAX Systems の場合と異なる数値パターンを生成します (RAN 関数と RANDU 関数は Digital Fortran 77 for OpenVMS VAX Systems との互換性を維持するために提供されます)。

- フォーマッティングした入出力文でのホレリス定数

  次のいずれかの場合には，Digital Fortran 77 for OpenVMS VAX Systems と Digital Fortran for OpenVMS Alpha 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)')
```

  Digital Fortran 77 for OpenVMS VAX Systems では，READ 文によって読み込まれた値が大部分の出力になります。(FMT2 は無視されます)，Digital Fortran for OpenVMS Alpha では，WRITE 文の出力は "abcdefghij"になります (つまり，READ 文によって読み込まれた値は WRITE 文によって書き込まれる値に影響を与えません)。

11.4.2 コマンド行修飾子

Digital Fortran for OpenVMS Alpha と Digital Fortran 77 for OpenVMS VAX Systems 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。この節では，Digital Fortran for OpenVMS Alpha と Digital Fortran 77 for OpenVMS VAX Systems のコマンド行の修飾子の相違点を要約します。

Digital Fortran for OpenVMS Alpha のコンパイル・コマンドとオプションについての詳しい説明は。『DEC Fortran User Manual for OpenVMS AXP Systems』を参照してください。Digital Fortran 77 for OpenVMS VAX Systems のコンパイル・コマンドとオプションについての詳しい説明は『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 Alpha 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。 Alpha システムでは， F90 コマンドを使用して Digital Fortran 90 コンパイラによるコンパイルを開始します。

11.4.2.1 Digital Fortran for OpenVMS Alpha 固有の修飾子

表 11–7 は Digital Fortran for OpenVMS Alpha コンパイラ修飾子のうち， Digital Fortran 77 for OpenVMS VAX Systems のオプションがなく， Digital Fortran 77 for OpenVMS VAX Systems のバージョン 6.4 でサポートされない修飾子を示しています。

表 11–7 Digital Fortran 77 for OpenVMS VAX Systems にない Digital Fortran for OpenVMS Alpha 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 (Digital Fortran 77 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。 Digital 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。

(次ページに続く)

表 11–7 (続き) Digital Fortran 77 for OpenVMS VAX Systems にない Digital Fortran for OpenVMS Alpha 修飾子

修飾子	説明
/SEPARATE_COMPILATION	DEC Fortran コンパイラが次の処理を行うかどうかを制御する。 • Digital Fortran 77 for OpenVMS VAX Systems と同様に，個々のコンパイル・ユニット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・ユニットを 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 は省略時の設定である)。この結果，プロシージャ間の最適化が促進される。
/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	特定のキーワードは， Digital Fortran 77 for OpenVMS VAX Systems で使用できない。
/VMS	Digital Fortran for OpenVMS Alpha が特定の Digital Fortran 77 for OpenVMS VAX Systems 表記法を使用することを要求する。

11.4.2.2 Digital Fortran 77 for OpenVMS VAX Systems 固有の修飾子

この節では， Digital Fortran 77 for OpenVMS VAX Systems コンパイラのオプションのうち， Digital Fortran for OpenVMS Alpha 修飾子に対応するオプションがないものをまとめます。

表 11–8 は， Digital Fortran 77 for OpenVMS VAX Systems のバージョン 6.4 固有のコンパイル・オプションを示しています。

表 11–8 Digital Fortran for OpenVMS Alpha でサポートされない Digital Fortran 77 for OpenVMS VAX Systems 修飾子

Digital Fortran 77 for OpenVMS VAX Systems 修飾子	説明
/BLAS=(INLINE,MAPPED)	Digital Fortran 77 for OpenVMS VAX Systems 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。 Digital Fortran 77 for OpenVMS VAX Systems の場合にのみ使用できる。
/CHECK=ASSERTIONS	アサーション・チェックを許可または禁止する。 Digital Fortran 77 for OpenVMS VAX Systems に対してのみ使用できる。
/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。

(次ページに続く)

表 11–8 (続き) Digital Fortran for OpenVMS Alpha でサポートされない Digital Fortran 77 for OpenVMS VAX Systems 修飾子

Digital Fortran 77 for OpenVMS VAX Systems 修飾子	説明
/DIRECTIVES=DEPENDENCE	指定されたコンパイラ・ディレクティブがコンパイル時に使用されるかどうかを指定する。 Digital Fortran 77 for OpenVMS VAX Systems に対してのみ使用できる。
/PARALLEL=(MANUAL または AUTOMATIC)	並列処理をサポートする。
/SHOW=(DATA_DEPENDEN-CIES,DICTIONARY,LOOPS)	リスト・ファイルに次の情報を登録するかどうかを制御する。 • ベクタ化または自動分解を禁止するデータ依存性と，依存解析の対象とならないループに関する診断情報 (DATA_DEPENDENCIES)。 • インクルードした Common Data Dictionary レコードからのソース・ライン (DICTIONARY)。 • コンパイル後のループ構造に関するレポート (LOOPS)。 DATA_DEPENDENCIES および LOOPS キーワードは Digital Fortran 77 for OpenVMS VAX Systems に対してのみ使用できる。
/VECTOR	ベクタ処理を要求する。 Digital Fortran 77 for OpenVMS VAX Systems に対してのみ使用できる。
/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印刷するかどうかを制御する。 Digital Fortran 77 for OpenVMS VAX Systems に対してのみ使用できる。

要求された (手作業) 分解に関連するすべての CPAR$ディレクティブと特定の CDEC$ディレクティブ，およびそれに関連する修飾子またはキーワードも Digital Fortran 77 for OpenVMS VAX Systems 固有です。『*DEC Fortran Language Reference Manual*』を参照してください。

Digital Fortran 77 for OpenVMS VAX Systems のコンパイル・コマンドとオプションに関する詳しい説明は『DEC Fortran User Manual for OpenVMS VAX Systems』を参照してください。

11.4.3 トランスレートされた共有可能イメージとの相互操作性

Digital Fortran for OpenVMS Alpha を使用すれば，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可能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 コマンド行に/TIE 修飾子を指定します。
- LINK コマンド行に/NONATIVE_ONLY 修飾子を指定します。

作成された実行可能イメージには，最終的な実行可能イメージが共有可能イメージと相互操作できるようなコードが含まれます。たとえば，Digital Fortran 77 for OpenVMS VAX Systems RTL (FORRTL) が Digital Fortran for OpenVMS Alpha 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。ネイティブなプログラム (Digital Fortran for OpenVMS Alpha RTL) とトランスレートされたプログラム (Digital Fortran 77 for OpenVMS VAX Systems RTL) は，ファイルをオープンする 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。

プログラムでは，組み込み名を使用しなければならず (接頭辞を除く)，完全な名前 (fac$xxxx) でルーチンを呼び出すわけではありません。

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$RAB システム関数を EXTERNAL として宣言できます。ネイティブな Alpha プログラムでは，FOR$RAB を内部関数として使用しなければなりません。

11.4.4 Digital Fortran 77 for OpenVMS VAX Systems データの移植

レコード・タイプは，Digital Fortran 77 for OpenVMS VAX Systems でも Digital Fortran for OpenVMS Alpha 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 フォーマットに変換するには，/TRANSFER=BLOCK のかわりに/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，Digital Fortran 77 for OpenVMS VAX Systems 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 浮動小数点フォーマットで格納し，REAL*8 または COMPLEX*16 データを D 浮動小数点または G 浮動小数点フォーマットで格納し，REAL*16 データを H 浮動小数点フォーマットで格納することを念頭においてください。Digital 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，COMPLEX*8，および COMPLEX*16 データをそれぞれ，表 11–9 に示すフォーマットのいずれかで格納します。

表 11–9 VAX システムと Alpha システムでの浮動小数点データ

データ宣言	VAX フォーマット	Alpha フォーマット
REAL*4 と COMPLEX*8	VAX F 浮動小数点フォーマット	IEEE S 浮動小数点または VAX F 浮動小数点フォーマット
REAL*8 と COMPLEX*16	VAX D 浮動小数点または G 浮動小数点フォーマット	IEEE T 浮動小数点, VAX D 浮動小数点[1]または VAX G 浮動小数点フォーマット
REAL*16 と COMPLEX*32	VAX H 浮動小数点	Digital Fortran for OpenVMS Alpha ではサポートされない。おそらく RTL ルーチン CVT$CONVERT_FLOAT を使用して変換しなければならない。

[1]Alpha システムでは，演算を実行するときに VAX D 浮動小数点フォーマットを使用することは望ましくない。このような場合には， Digital Fortran for OpenVMS Alpha の変換ルーチンを使用する変換プログラムで D 浮動小数点フォーマットを IEEE T 浮動小数点 (または VAX G 浮動小数点) フォーマットに変換することを考慮しなければならない。

11.5 DEC Pascal for OpenVMS Alpha システムとVAX Pascalの互換性

この節では， DEC Pascal と他の DEC の Pascal コンパイラを比較し， VAX システムと Alpha システムの DEC 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。

11.5.1 DEC Pascal の新機能

表 11–10 は以前VAX Pascalでは提供されていなかった機能を示します。

表 11–10 DEC Pascal の新機能

機能	説明
OpenVMS システムのサポート	OpenVMS プラットフォームで有効なすべてのデータ・タイプを含む。
あらかじめ定義されている定数のうち再定義可能な値	MAXINT, MAXUNSIGNED, MAXREAL, MINREAL, ESPREAL の値はプラットフォームによって定義され，コンパイラが整数のサイズや浮動小数点型を指定するために変更する。
COMMON, EXTERNAL, GLOBAL. PSECT, WEAK_EXTERNAL および WEAK_GLOBAL 属性への一重引用符で囲まれたオプション・パラメータ	変更されない識別子をリンカへ渡すことを許可する。

(次ページに続く)

表 11–10 (続き) DEC Pascal の新機能

機能	説明
二重引用符で囲まれた文字列	DEC Pascal は文字列および文字の区切りとして二重引用符を使用できる。
埋め込まれた文字列の値	二重引用符で囲まれた文字列のなかでは，C プログラミング言語でラインフィード文字を表わす "/n"のような，バックスラッシュとそのすぐ後に指定された文字をサポートする。
追加されたデータ・タイプと値	DEC Pascal では次のデータ・タイプをサポートする。 ALFA, CARDINAL, CARDINAL16, CARDINAL32, INTEGER16, INTEGER32, INTEGER64, INTSET, POINTER, UNIV_PTR, UNSIGNED16, UNSIGNED32 および UNSIGNED64。
UNSIGNED 値と INTEGER 変数の割り当て	DEC Pascal は, UNSIGNED 値が割り当てにおいて INTEGER 変数およびアレイ・インデックスに互換性を持つ。
文字のアンパックされたアレイへの文字列値の割り当て	DEC Pascal は CHAR 変数の ARRAY が固定長の文字列として扱われることを許可する。
追加された文	DEC Pascal では次の文をサポートします。 BREAK, CONTINUE, EXIT, NEXT, および RETURN。
追加されたルーチン	DEC Pascal は次の機能やプロシージャをサポートします。 ADDR, ARGC, ARGV, ASSERT, BITAND, BITNOT, BITOR, BITXOR, HBOUND, LBOUND, FIRST, FIRSTOF, LAST, LASTOF, IN_RANGE, LSHIFT, RSHIFT, LSHFT, RSHFT, MESSAGE, NULL, RANDOM, SEED, REMOVE, SIZEOF, SYSCLOCK および WALLCLOCK。
RESET, REWRITE および EXTEND に対するオプションのセカンド・パラメータ	DEC Pascal は，ファイル変数に関連したファイル名に対する定数文字列表現であるセカンド・パラメータを使用できる。
コンパイラ・コマンドの切り替え	DEC Pascal は，データ・タイプに対する記憶領域およびアラインメントの割り当ての指定を変更するためのスイッチを持つ。またスイッチで最適化のレベルを変更することも可能である。 Alpha システムでは，ひとつのオプションが REAL および DOUBLE データ・タイプの省略時の意味を制御する。また，スイッチへの引数でアラインメントに関するメッセージ，および異なるプラットフォーム間でのアラインメントの互換性や特定のプラトフォームで使用できない機能を制御する。

11.5.2 動的条件ハンドラの設定

DEC Pascal では， LIB$ESTABLISH の代わりに使用するために， ESTABLISH と REVERT という組み込みルーチンが提供されます。 LIB$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されます。

11.5.3 レコード・ファイルに対する省略時のアラインメント規則の変更

DEC Pascal では，フィールド・アラインメントの位置や， POS， ALIGNED, DATA 属性およびデータ・コンパイラ切り替えの位置を上書きすることが可能です。

11.5.4 あらかじめ宣言されている名前の使用方法

互換性を維持するために DEC Pascal は，表 11–11 に示されるあらかじめ宣言されている名前を含むプログラムをコンパイルすることができますが，弊社では以下のように識別子を置き換えて使用されることをお勧めします。

表 11–11 あらかじめ宣言されている名前の使用方法

識別子	望ましい使用法
ADDR	ADDRESS 機能をご使用ください
ALFA	TYPE ALFA = PACKED ARRAY [1..10]OF CHAR と同じ
BITAND	UAND 文と同じ
BITNOT	UNOT 文と同じ
BITOR	UOR 文と同じ
BITXOR	UXOR 文と同じ
EXIT	BREAK 文と同じ
FIRST,FIRSTOF	LOWER 機能と同じ
HBOUND	UPPER 機能と同じ
IN_RANGE	サブレンジ・チェックができない場合のみ有効。 IN_RANGE(X) は (X≥LOWER(X))AND(X≤UPPER(X)) と同じ
INTSET	TYPE INTSET = SET OF 0 .. 255; と同じ
LAST, LASTOF	UPPER 機能と同じ
LBOUND	LOWER 機能と同じ
LSHFT	LSHIFT 機能と同じ
MESSAGE	WRITELN(ERR,expression) と同じ
NEXT	CONTINUE 文と同じ
NULL	empty 文と同じ
REMOVE	DELETE_FILE プロシージャと同じ
RSHFT	RSHIFT 機能と同じ
SIZEOF	SIZE 機能と同じ
STLIMIT	コンパイルするが，エラーを返さない
UNIV_PTR	TYPE UNIV_PTR = POINTER; と同じ

11.5.5 プラットフォームに依存する機能

DEC Pascal はコンパイルされたプラットフォームと同じプラットフォーム (オペレーティング・システムとハードウェアの組み合わせ) でのみ環境ファイルを使用できます。

さらに， VAX システムでのみサポートされる DEC Pascal の機能を以下に示します。

- QUADRUPLE データ型
- H 浮動小数点データ型

- VAX Pascalバージョン 1.0 ダイナミック・アレイ
- MFPR と MTPR という前もって宣言されたルーチン
- [OVERLAID]属性
- Table of contents in listing
- ルーチン上の最適化属性

以下に Alpha システムでのみサポートされる DEC Pascal の機能を示します。

- 列挙されたデータ型を読むときの省略形
- 索引ファイル編成
- 相対ファイル編成

11.5.6 古い機能

この節では，サポートされてはいても使用が勧められない機能について説明します。これらは，弊社の他の Pascal コンパイラとの互換性のためにのみ提供されます。

11.5.6.1 /OLD_VERSION 修飾子

/OLD_VERSION 修飾子は，コンパイラにVAX Pascalバージョン 1.0 の言語の定義を使用して，VAX Pascalバージョン 1.0 とその後のバージョンの違いを解決させます。この修飾子を指定すれば，既存のプログラムを引き続き使えます。

11.5.6.2 /G_FLOATING 修飾子

/G_FLOATING 修飾子は，コンパイラに DOUBLE 型の値に対して G_floating の表現と命令を使用することを指定します。[[NO]G_FLOATING]属性は OpenVMS VAX システムでも OpenVMS Alpha システムでも指定できます。

/G_FLOATING 修飾子の使用が，ソース・プログラムやモジュール内で指定された倍精度属性と矛盾するときは，エラーとなります。倍精度数の受け渡しを行うルーチンやコンパイル単位間で複数の浮動小数点フォーマットを混在させることはできません。すべての OpenVMS VAX プロセッサが G 浮動小数点データ型をサポートするわけではありません。

これより前に，コンパイラが浮動小数点データ型を指定する方法だった，/FLOAT 修飾子の説明も参照してください。/FLOAT 修飾子は， Alpha システムのみでサポートされる IEEE 浮動小数点データ型も選択できます。

11.5.6.3 OVERLAID 属性

OVERLAID 属性はコンパイル単位で宣言された変数へ，記憶領域がどのように割り当てられるかを指定します。コンパイル・ユニットで OVERLAID が指定されると，プログラムやモジュール・レベルで宣言された変数は (STATIC または PSECT 属性を持たない限り)，すべての他の上書きされたコンパイル・ユニットで，静的変数の記憶領域を上書きします。

この属性は，VAX Pascalバージョン 1.0 で提供された分割コンパイル機能を用いているプログラムでの使用のみを対象としています。

A

アプリケーション評価チェックリスト

このチェックリストの例は， OpenVMS Alpha に移行する対象としてアプリケーションを評価するために弊社が使用したチェックリストをもとにしています。

質問の後の大括弧で囲んだコメントは，その質問の目的を正確に示しています。

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 Alpha 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 Alpha 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考慮してください。 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 Alpha 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，アプリケーションの異なるバージョンが，同じデータベースをアクセスできる場合には，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設定するために，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設定するために，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? インストレーション検証プロシージャ，リグレッション・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 Alpha 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 Alpha 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 7.1 で実行されていますか? ☐ YES ☐ NO

   c. アプリケーションは OpenVMS Alpha で有効でない機能を使用しますか? ☐ YES ☐ NO

   [OpenVMS Alpha への移行の基礎となるのは， OpenVMS VAX バージョン 7.1 です。 c に対する回答が "YES"の場合には，アプリケーションは OpenVMS Alpha でまだサポートされていない機能を使用する可能性があり，また OpenVMS Alpha の現在のバージョンと互換性のない OpenVMS RTL，または他の共有可能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. サード・パーティ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 Alpha 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，サード・パーティ・プロダクトの業者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?

   モジュール数は? _____

   コードの行数またはキロバイト数は? _____

   必要なディスク空間? _____

   [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，または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コードの機能を各高級の言語コンパイラ，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

[Alpha アプリケーションで，VAX MACROや MACRO–64 Assembler for OpenVMS Alpha を使用することは，望ましくありません。アプリケーションを最初に開発した段階では，まだ提供されていなかった OpenVMS システム・サービスを呼び出すことにより，特定のユーザ・モード・アプリケーションで，アセンブリ言語コードを他のコードに置換することができます。]

10. a. アプリケーションを構成する，すべてのソース・ファイルをアクセスできますか? ☐ YES ☐ NO

    b. 弊社サービスの使用を検討している場合には，弊社がこれらのソース・ファイルとビルド・プロシージャをアクセスできますか? ☐ YES ☐ NO

    [a に対する回答が "YES"の場合には，ソース・ファイルを入手できない部分の移行はトランスレーションによって実行しなければなりません。 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。]

11. a. アプリケーションをテストするために必要なリグレッション・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，これらのリグレッション・テストの移行を考慮しなければなりません。 OpenVMS Alpha の初期リリースでは， DEC Test Manager は提供されません。リグレッション・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 浮動小数点フォーマット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， Alpha システムが準備している 53 ビットの精度の D-floatin フォーマットを使用するか，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フォーマット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，Alpha 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有されるグローバル・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有される場合には，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，データ構造がパックされていること，自然なアラインメントにされていること，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，Alpha 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。コンパイラ・スイッチによって，アラインメントは強制的に設定されるため，データ・アラインメントに関するコンパイラの省略時の設定はさまざまです。通常，VAX システムでは，省略時のデータ・アラインメントはパック形式のアラインメントですが，Alpha コンパイラの省略時の設定は，可能な限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，Alpha 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。Alpha のページ・サイズは 512 バイトよりはるかに大きく，各システムで異なります。したがって，ページ・サイズを明示的に参照することは避け，可能な限り，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有するために，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，明示的な同期を使用しなければならないのかどうかを判断し，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，プロセス空間でデータ・アクセスを共有するかどうかを判断しなければなりません。共有する場合には，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

[Alpha 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するときに，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，ネイティブな Alpha 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

[OpenVMS Alpha の初期リリースでは，ユーザ作成デバイス・ドライバはサポートされません。この機能が必要な場合には，弊社の担当者にご連絡ください。]

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? _____

[接続/割り込みは， OpenVMS Alpha システムでサポートされません。この機能が必要な場合には，弊社の担当者にご連絡ください。]

24. アプリケーションで，機械語を直接作成または変更しますか? ☐ YES ☐ NO

[OpenVMS Alpha 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もっと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，お客様にとってもっと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

CISC
複雑命令セット・コンピュータを参照。

IIF
VAX イメージ間のインターフェイスに関する情報を記述した ASCII ファイル。VEST は IIF を使って他のイメージに対する参照を解決し，適切なリンクを作成する。

PALcode
特権付きアーキテクチャ・ライブラリを参照。

RISC
縮小命令セット・コンピュータを参照。

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な Alpha 共有可能イメージ。 TIE はトランスレートされたイメージとネイティブな 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。

VAX Environment Software Translator (VEST)
ソフトウェア移行用のツールであり， VAX の実行可能イメージと共有可能イメージを， 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変換する。**トランスレートされたイメージ**を参照。

VEST
VAX Environment Software Translatorを参照。

アラインされたデータ
アラインメントの用件を満たすデータを，**アラインされたデータ**と呼ぶ。

アラインメント
自然なアラインメントを参照。

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，たとえば読み込み専用アクセス属性，読み込み/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。

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) や共有可能イメージ・セクション。

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。アドレスを 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。

クロス開発

1 つのシステムで実行されるツールを使用して，別のシステムを対象としたソフトウェアを作成する処理。たとえば， VAX システムで実行されるツールを使用して，Alpha システムのためのコードを作成する処理。

互換性

あるコンピュータ・システム (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 Alpha) で実行できる能力。

自然なアラインメント

データ・アドレスをデータ・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，その構造も**自然にアラインされる**という。

ジャケット・ルーチン

1 つの呼び出し規則から別の規則にプロシージャ呼び出しを変換するプロシージャ。たとえば，トランスレートされた VAX イメージ (VAX 呼び出し規則を使用するイメージ) とネイティブな Alpha イメージ (Alpha 呼び出し規則を使用するイメージ) との間で呼び出しを変換する。

縮小命令セット・コンピュータ (RISC)

複雑さが削減された命令セットを使用するコンピュータ。ただし，命令の数が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。

同期
マルチプロセシング環境や共有データを使用するユニプロセシング環境で操作するときに，きちんと定義された予測可能な結果が得られるように，一部の共有資源に対するアクセスを制御する方法。

同時実行/並列処理
複数のエージェントが共有オブジェクトに対して操作を同時に実行すること。

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。特殊な命令がルーチンを呼び出し，これらは中断せずに実行される。

トランスレーション
VAX バイナリ・イメージを Alpha 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。変換は静的処理であり，できるだけ多くの VAX コードがネイティブな Alpha 命令に変換される。実行時に変換されなかった VAX コードに対しては， TIE が最終的に解釈する。

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 Alpha 上の実行可能イメージまたは共有可能イメージ。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もとになる 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。

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 Alpha 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 Alpha コード
- Translated Image Environment (TIE)の呼び出し

ネイティブなイメージ
OpenVMS Alpha コンパイラ， OpenVMS Alpha リンカを使用して作られた OpenVMS Alpha 上の実行可能イメージ，または共有可能イメージを，**トランスレートされたイメージ**に対してこう呼ぶ。

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。

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た後，その操作は中断されずに，必ず最後まで終了する。

読み込み/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。

不可分な命令 (atomic instruction)
単一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。

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，複雑な操作を実行する命令を取り扱うコンピュータ。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(**縮小命令セット・コンピュータ**) コンピュータの反対語である。

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。

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含む部分。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確認できる。Alpha システムでは，プログラム・カウンタはレジスタではないので注意が必要。

プロセッサ・ステータス (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，オーバーフロー，0，負)，演算トラップ・イネーブル・ビット (整数オーバーフロー，10 進オーバーフロー，浮動小数点アンダーフロー)，およびトレース・イネーブル・ビットが含まれる。

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，共有，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

Alpha 環境で，512 バイトのメモリ・サイズを指す表現。Alpha システムでは，特定の 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ーフェイスは VAX システムの外部インターフェイスと互換性を維持するが，OpenVMS Alpha は内部的には CPU メモリのページ・サイズの整数倍でのみ，メモリを管理する。

読み込み/書き込みの順序

1 つの CPU のメモリに対する読み込み，書き込みなどの操作を実行エージェント (密接に結合されたシステム内の別の CPU または装置) から確認できるようになる順序。

読み込み/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込み，変更し，書き込むハードウェア操作。

粒度

1 つの命令によって読み込むか，または書き込むことができるデータ・サイズ，つまり，個別に読み込みまたは書き込み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 バイト以上の粒度である。

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。

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。

索引

A

B

C

D

E

F

G

T

U

V

W

ア

イ

エ

オ

カ

キ

ク

ケ

コ

サ

シ

ス

セ

ソ

タ

テ

ト

ナ

ニ

ネ

ハ

ヒ

フ

へ

ホ

マ

メ

モ

ユ

ヨ

ラ

リ

レ

ロ

ワ

OpenVMS Alpha **オペレーティング・システム**
OpenVMS VAX **から** OpenVMS Alpha **へのアプリケーションの移行**

1999 年 4 月　発行

コンパックコンピュータ株式会社

〒 140-8641 東京都品川区東品川 2-2-24 天王洲セントラルタワー

電話　(03)5463-6600 (大代表)

AA-R1E8B-TE