OpenVMS AXP
オペレーティング・システム

OpenVMS AXP オペレーティング・システムへの移行：再コンパイルと再リンク

AA-PU8LC-TE

ソフトウェア・バージョン: OpenVMS AXP V6.1

日本ディジタル イクイップメント株式会社

1994 年 7 月

ZK5780

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

目次

まえがき xi

1 **はじめに**

1.1 **概要** 1–1

1.2 **ネイティブな** AXP **コンパイラによるアプリケーションの再コンパイル** 1–2

1.3 **アプリケーションにおける** VAX **アーキテクチャに依存する部分の識別** 1–2

1.4 AXP **システムでのアプリケーションの再リンク** 1–4

1.5 VAX **システムと** AXP **システムの算術演算ライブラリ間の互換性** 1–7

1.6 **ホスト・アーキテクチャの判断** 1–8

2 **ページ・サイズの拡大に対するアプリケーションの対応**

2.1 **概要** 2–1
2.1.1 互換性のある機能 2–2
2.1.2 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 2–3

2.2 **メモリ割り当てルーチンの確認** 2–10
2.2.1 拡張された仮想アドレス空間でのメモリの割り当て 2–12
2.2.2 既存の仮想アドレス空間でのメモリの割り当て 2–15
2.2.3 仮想メモリの削除 2–16

2.3 **メモリ・マッピング・ルーチンの確認** 2–17
2.3.1 拡張した仮想アドレス空間へのマッピング 2–18
2.3.2 特定の位置への単一ページのマッピング 2–21
2.3.3 定義されたアドレス範囲へのマッピング 2–22
2.3.4 オフセットによるセクション・ファイルのマッピング 2–32

2.4 **ページ・サイズの実行時確認** 2–35

2.5 **メモリをワーキング・セットとしてロックする操作** 2–36

3 共有データの整合性の維持

3.1 **概要** 3–1
3.1.1 不可分性を保証する VAX アーキテクチャの機能 3–2
3.1.2 OpenVMS AXP の互換性機能 3–4

3.2 **アプリケーションにおける不可分性への依存の検出** 3–5
3.2.1 明示的に共有されるデータの保護 3–7
3.2.2 無意識に共有されるデータの保護 3–13

3.3 **読み込み/書き込み操作の同期** 3–15

3.4 **トランスレートされたイメージの不可分性の保証** 3–17

4 アプリケーション・データ宣言の移植性の確認

4.1 **概要** 4–1

4.2 VAX **データ型への依存の確認** 4–2

4.3 **データ型の選択に関する仮定の確認** 4–6
4.3.1 データ型の選択がコード・サイズに与える影響 4–6
4.3.2 データ型の選択が性能に与える影響 4–6

5 アプリケーション内の条件処理コードの確認

5.1 **概要** 5–1

5.2 **条件処理ルーチンがアーキテクチャ固有の機能に依存しているかどうかの確認** 5–2

5.3 **例外条件の識別** 5–7
5.3.1 AXP システムでの算術演算例外のテスト 5–10
5.3.2 データ・アラインメント・トラップのテスト 5–13

5.4 **条件処理に関連する他の作業の実行** 5–14

6 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

6.1 **概要** 6–1
6.1.1 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 6–2
6.1.2 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 6–3

6.2 **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** 6–3

6.3 **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** 6–7
6.3.1 シンボル・ベクタ・レイアウトの制御 6–10
6.3.2 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 6–14

A OpenVMS AXP **コンパイラ**

A.1 DEC Ada **の** AXP **システムと** VAX **システム間の互換性** A–1
A.1.1 データ表現とアラインメントにおける相違点 A–2
A.1.2 タスクに関する相違点 A–3
A.1.3 プラグマに関する相違点 A–3
A.1.4 SYSTEM パッケージの相違点 A–4
A.1.5 他の言語パッケージでの相違点 A–5
A.1.6 あらかじめ定義されている具現に対する変更 A–5

A.2 DEC C for OpenVMS AXP **システムと**VAX C**との互換性** A–6
A.2.1 言語モード A–6
A.2.2 DEC C for OpenVMS AXP システムのデータ型のマッピング A–7
A.2.2.1 浮動小数点マッピングの指定 A–8
A.2.3 AXP システム固有の機能 A–8
A.2.3.1 Alpha AXP 命令のアクセス A–9
A.2.3.2 Alpha AXP 特権付きアーキテクチャ・ライブラリ (PALcode) 命令のアクセス A–9
A.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 A–10
A.2.4 VAX Cと DEC C for OpenVMS AXP システムのコンパイラの相違点 A–11
A.2.4.1 データ・アラインメントの制御 A–11
A.2.4.2 引数リストのアクセス A–11
A.2.4.3 例外の同期化 A–11
A.2.5 /STANDARD=VAXC モードでサポートされないVAX Cの機能 A–12

A.3 DEC COBOL **と**VAX COBOL **の互換性** A–13
A.3.1 コマンド・ライン修飾子 A–14
A.3.1.1 DEC COBOL とVAX COBOL が共有する修飾子 . . . A–15
A.3.1.2 VAX COBOL で使えない DEC COBOL 修飾子 A–16
A.3.1.3 DEC COBOL で使用できないVAX COBOL 修飾子 A–17

A.3.2 動作の相違点 A–18
A.3.2.1 DEC COBOL の/ALIGNMENT 修飾子とアラインメント指示文による数値項目に対するアラインメントの指定 A–18
A.3.2.1.1 /ALIGNMENT 修飾子の使用 A–19
A.3.2.1.2 アラインメント指示文の使用 A–20
A.3.2.2 DEC COBOL の/CHECK=NODECIMAL オプションによる数値データの確認 A–20
A.3.2.3 先行する空白文字をゼロに変換する DEC COBOL の/CONVERT=LEADING_BLANKS オプション A–21
A.3.2.4 DEC COBOL の/FLOAT 修飾子に浮動小数点データ型を指定する A–21
A.3.2.5 /OPTIMIZE 修飾子でコードを最適化する A–22
A.3.2.6 特殊な予約語をチェックする/RESERVED_WORDS 修飾子 A–22
A.3.2.7 COBOL ANSI 規格の言語拡張機能を呼び出す/STANDARD 修飾子 A–23
A.3.2.7.1 /STANDARD=V3 オプション A–23
A.3.2.7.2 /STANDARD 修飾子と/WARNINGS 修飾子 A–27
A.3.2.8 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呼ぶ /TIE 修飾子 A–28
A.3.2.9 VAX COBOL から DEC COBOL へのプログラムの変換 A–28
A.3.2.10 プログラムの構造 A–29
A.3.2.11 COPY 文と REPLACE 文 A–30
A.3.2.12 MOVE 文 A–35
A.3.2.13 ACCEPT 文と DISPLAY 文 A–36
A.3.2.14 LINAGE 文 A–37
A.3.2.15 ファイル状態の違い A–37
A.3.2.16 システム・サービス・コールからの戻り値 A–37
A.3.2.17 倍精度データ項目における記憶領域の違い A–39
A.3.2.18 RMS スペシャル・レジスタ A–40
A.4 DEC Fortran for OpenVMS AXP **と**VAX FORTRAN**との互換性** A–40
A.4.1 言語機能 A–40
A.4.1.1 DEC Fortran 固有の言語機能 A–41
A.4.1.2 VAX FORTRAN固有の言語機能 A–43
A.4.1.3 解釈方法の相違 A–44
A.4.1.4 DEC Fortran の制約事項 A–45

A.4.2 コマンド・ライン修飾子 A–46
A.4.2.1 共有される修飾子 A–46
A.4.2.2 DEC Fortran 固有の修飾子 A–48
A.4.2.3 VAX FORTRAN固有の修飾子 A–49
A.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 A–51
A.4.4 VAX FORTRANデータの移植 A–52
A.5 DEC Pascal for OpenVMS AXP **システムと**VAX Pascal**の互換性** A–53
A.5.1 DEC Pascal の新機能 A–53
A.5.2 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 A–55
A.5.3 あらかじめ宣言されている名前の使用法 A–55
A.5.4 プラットフォームに依存する機能 A–56
A.5.5 古い機能 A–57
A.5.5.1 /OLD_VERSION 修飾子 A–57
A.5.5.2 /G_FLOATING 修飾子 A–57
A.5.5.3 OVERLAID 属性 A–57

索引

例

1–1 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 1–9
2–1 仮想アドレス空間の拡張によるメモリの割り当て 2–14
2–2 既存のアドレス空間でのメモリの割り当て 2–16
2–3 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 2–20
2–4 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 2–25
2–5 例 2–4 を AXP システムで実行するのに必要なソース・コードの変更 2–29
2–6 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 2–35
3–1 AST スレッドを含む VAX プログラムにおける不可分な処理への依存 3–8
3–2 例 3–1 の同期バージョン 3–11
4–1 VAX Cコードでのデータ型に関する仮定 4–5
5–1 条件処理ルーチン 5–6
5–2 条件処理プログラムの例 5–18
6–1 メイン・プログラム (MYMAIN.C) のソース・コード 6–4

6–2 共有可能イメージ (MYMATH.C) のソース・コード 6–4

図

2–1 仮想アドレスのレイアウト 2–12
2–2 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 2–34
3–1 同期に関する判断 3–7
3–2 例 3–1 での不可分性の仮定 3–10
3–3 AXP システムでの読み込み/書き込み操作の順序 3–16
4–1 VAX Cの使用による mystruct のアラインメント 4–10
4–2 DEC C for OpenVMS AXP システムの使用による mystruct のアラインメント 4–10
5–1 VAX システムと AXP システムでのシグナル・アレイ 5–2
5–2 VAX システムと AXP システムでのメカニズム・アレイ 5–4
5–3 SS$_HPARITH 例外シグナル・アレイ 5–11
5–4 SS$_ALIGN 例外のシグナル・アレイ 5–14

表

1–1 AXP システム固有のリンカ修飾子 1–6
1–2 AXP システム固有のオプション 1–7
1–3 ホスト・アーキテクチャを指定する$GETSYI アイテム・コード 1–9
2–1 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 2–4
2–2 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 2–10
4–1 VAX と AXP のネイティブなデータ型の比較 4–3
5–1 シグナル・アレイ内の引数 5–3
5–2 メカイズム・アレイの引数 5–5
5–3 アーキテクチャ固有のハードウェア例外 5–9
5–4 例外サマリ引数のフィールド 5–12
5–5 ランタイム・ライブラリ条件処理サポート・ルーチン 5–15
A–1 DEC C for OpenVMS AXP コンパイラの操作モード A–6
A–2 DEC C for OpenVMS AXP コンパイラでの算術演算データ型のサイズ A–7
A–3 DEC C for OpenVMS AXP コンパイラの浮動小数点マッピング A–8

A–4 OpenVMS AXP システム固有の DEC C コンパイラ機能 A–9
A–5 DEC C for OpenVMS AXP コンパイラの不可分性を保証する組み込み機能 A–10
A–6 DEC COBOL とVAX COBOL が共有する修飾子 A–15
A–7 VAX COBOL で使えない DEC COBOL 修飾子 A–16
A–8 DEC COBOL で使えないVAX COBOL 修飾子 A–17
A–9 /STANDARD 修飾子の I-O ファイル状態コード A–25
A–10 DEC Fortran とVAX FORTRANで共有される修飾子 A–47
A–11 VAX FORTRANでサポートされない DEC Fortran 修飾子 A–48
A–12 DEC Fortran でサポートされないVAX FORTRANオプション A–49
A–13 VAX システムと AXPVMS システムでの浮動小数点データ A–52
A–14 DEC Pascal の新機能 A–53
A–15 あらかじめ宣言されている名前の使用法 A–55

まえがき

本書の目的

本書は，開発者が OpenVMS VAX アプリケーションを OpenVMS AXP システムに移行する際に役立つように構成されています。

対象読者

本書は経験の豊富なソフトウェア技術者を対象にしており，特に C や FORTRAN などの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コードを移植する際の責任者を対象にしています。

本書の構成

本書は 6 つの章と 1 つの付録から構成されています。

第 1 章　アプリケーションが， VAX アーキテクチャに依存している部分を調べる方法について，その概要を示します。

第 2 章　アプリケーションが VAX のページ・サイズに依存しているときの対処方法を示します。

第 3 章　複数のプロセスによるデータ・アクセスに関して，アプリケーションが VAX アーキテクチャの同期方式に依存しているときの対処方法について説明します。

第 4 章　アラインメントに関する問題も含めて， AXP システムでのデータ宣言の影響について説明します。

第 5 章　アプリケーションが VAX の条件処理機能に依存しているときの問題への対処方法について説明します。

第 6 章　トランスレートされた VAX イメージを呼び出したり，これらのイメージから呼び出すことができるネイティブな AXP イメージの作成方法について説明します。

付録 A　OpenVMS AXP システムで，Ada，C，Cobol，FORTRAN および Pascal プログラミング言語によってサポートされる，新しい機能と変更された機能を簡単にまとめます。

参考文献

本書は OpenVMS VAX アプリケーションを OpenVMS AXP システムに移行する際のさまざまな問題を説明したマニュアル・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準備されています。

- 『OpenVMS AXP オペレーティング・システムへの移行：システム移行の手引き』

  VAX システムから Alpha AXP システムへの移行処理の概要を示し，移行計画の作成に役立つ情報も示します。移行計画を作成するときに判断しなければならない事項について説明し，それらの判断を下すのに必要な情報の入手方法も示します。さらに，さまざまな移行方法について説明し，各方法で必要な作業量を見積もるのに必要な情報と，各アプリケーションにとって最適な移行方法を選択する方法についても説明します。

- 『Migrating to an OpenVMS AXP System: Porting VAX MACRO Code』

  このマニュアルでは，MACRO-32 Compiler for OpenVMS AXP を使用してVAX MACROコードを AXP システムに移植する方法およびコンパイラの機能，移植不可能なコーディング様式の識別法，このようなコーディング様式のかわりとなる適切な方法などについて説明します。このマニュアルではまた，コンパイラの修飾子，ディレクティブ，および組み込み機能と，AXP システムに移植するために作成されたシステム・マクロの詳細な説明を示したリファレンスも記載されています。

また，VAX Environment Software Translator (VEST) に関する次のマニュアルがあります。

- 『DECmigrate for OpenVMS AXP Systems Translating Images』

このマニュアルでは，VAX Environment Software Translator (VEST) ユーティリティについて説明します。このマニュアルは，オプションとして提供されるレイヤード・プロダクトである DECmigrate for OpenVMS AXP に添付されており，このレイヤード・プロダクトは，OpenVMS VAX アプリケーションを OpenVMS AXP 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・モードの OpenVMS VAX イメージを，OpenVMS AXP 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させる方法，VEST を使用して，VAX イメージの中で AXP と互換性のない部分を元のソース・ファイルまでトレースする方法，および VEST を使用して，ネイティブなランタイム・ライブラリとトランスレートされたランタイム・ライブラリの間で互換性をサポートする方法について説明します。また，このマニュアルには完全な VEST コマンド・リファレンスも記載されています。

表記法

本書では次の表記法を使用します。

表記法	意味
Return	四角形で囲まれたこの記号は，キーボードのキーを押すことを示します。たとえば，Return は Return キーを押すことを示します。
Ctrl/x	Ctrl/x の記号は，Ctrl キーを押しながら，同時にあるキーを押すことを示します。たとえば，Ctrl/c は Ctrl キーと c 文字キーを同時に押します。
. . .	例の中で水平反復記号は，次のいずれかを示します。 • 文中のオプショナル引数が省略されていること • 前の項目 (1 つ以上) を 1 回以上繰り返すことができること • 追加パラメータ，値，あるいは他の情報を入力でること

表記法	意味
. . .	垂直反復記号は，コード例やコマンド形式から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ないからです。
( )	括弧は，複数のオプションを選択するときに，選択項目を括弧で囲まなければならないことを示す。
[ ]	大括弧は，項目が省略可能であることを示します (しかし，VMS ファイル指定のディレクトリ名の構文や，代入文の部分文字列指定の構文では，大括弧は省略可能ではありません)。
{ }	中括弧は，必ず 1 つを選択しなければならない項目を囲むために使用します。
太字	太字のテキストは，新しい用語を導入する場合や，引数，属性，条件の名前を示すために使用します。 また，マニュアルのオンライン・バージョンでユーザ入力を示す場合も，太字のテキストを使用します。
イタリック体	イタリック体のテキストは，システム・メッセージやコマンド・ラインの中で，変化する可能性のある情報を表現します。
英大文字	英大文字は，コマンド，修飾子，パラメータ，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。
—	コード例で使用されているハイフンは，要求に対する追加引数が後続の行に指定されることを示します。
数字	特に示した場合を除き，説明文の内部で使用している数字はすべて 10 進数です。数値が 10 進数以外 (2 進数， 8 進数， 16 進数) の場合には，そのことが明記されます。
「 」	かぎ括弧は，この製品のドキュメント構成に含まれるマニュアル名を示します。
『 』	二重かぎ括弧は，この製品のドキュメント構成に含まれないマニュアル名または，別の製品のマニュアル名を示します。

1
はじめに

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 AXP 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。特に，この章では次の内容について説明します。

- ネイティブなコンパイラやリンカなど， VAX プログラミング環境のツールの AXP 版の使用について
- アプリケーションで VAX アーキテクチャ固有の機能に依存している部分の識別

1.1 概要

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 AXP 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， AXP 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， AXP 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 AXP イメージを作成できます。

しかし，アプリケーションが高級言語で作成されている場合でも，アーキテクチャ固有の機能に依存している可能性があります。この後の節では， AXP システム上のプログラミング環境について説明し，アプリケーション・ソース・ファイルの中で，変更しなければ AXP システムに移行できないコードを識別するためのガイドラインを示します。

1.2 ネイティブな AXP 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，AXP 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。AXP システムでどのプログラミング言語を使用できるかについての詳しい説明は，『OpenVMS AXP オペレーティング・システムへの移行：システム移行の手引き』を参照してください。

AXP 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 (compatibility) 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，AXP 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，AXP 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 AXP 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 AXP 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。付録 A は，VAX システムと AXP システムで使用できるコンパイラの機能を示しています。

1.3 アプリケーションにおける VAX アーキテクチャに依存する部分の識別

ネイティブな AXP コードで生成された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た場合でも，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。オペレーティング・システムは，高いレベルの互換性を維持するように設計されています。しかし，2 つのアーキテクチャには基本的な相違があるため，ある種の矛盾をどうしても避けることができません。次のリストはアプリケーションの中で調べなければならない部分を示しています。本書のこの後の各章では，これらの各項目について詳しく説明します。

- アプリケーションに含まれるデータ宣言を確認してください。 VAX システムでデータを表現するために選択した高級言語のデータ型 (data type) が AXP システムでは最適なデータ型でない可能性があります。特に次のことを考慮してください。
  - **データのパック**—VAX システムのアプリケーションでは通常，メモリ資源を効率よく使用するために，可能な限り最小のデータ型を使用してデータを表現します。しかし， AXP システムでは，さまざまな理由から (第 4 章を参照)，大きなデータ型を使用した方が効率を向上できることがあります。
  - **データ型の選択**—Alpha AXP アーキテクチャは，大部分の VAX ネイティブ・データ型をサポートします。しかし， H 浮動小数点データ型など，特定の VAX データ型はサポートされません。アプリケーションがハードウェア固有のデータ型のビット表現やサイズに依存していないかどうかを確認してください。 Alpha AXP アーキテクチャでサポートされるすべてのデータ型のリストは，第 4 章を参照してください。
  - **データの共有アクセス**— 複数の実行スレッドによってアクセスされるデータを確認してください。 VAX アーキテクチャには，変数のインクリメントなどのように，他の実行スレッドからは割り込み不可能な 1 つの操作として見える，複雑な操作を実行できる命令があります。しかし， Alpha AXP アーキテクチャはロード・ストア・アーキテクチャであり，それ以外の不可分な (atomic) メモリ操作はサポートされません。この問題についての詳しい説明は，第 3 章を参照してください。

    さらに， VAX アーキテクチャでは，バイト・サイズおよびワード・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。 Alpha AXP アーキテクチャはアラインされたロングワード・サイズ，またはアラインされたクォドワード・サイズのデータに対してのみ，割り込み不可能なアクセスをサポートします。この問題がアプリケーションにどのような影響を与えるかについては，第 3 章と第 4 章を参照してください。
  - **バッファ・サイズ**— アプリケーションによっては， VAX ページ・サイズをもとにデータ・バッファのサイズを決定している可能性があります。 Alpha AXP アーキテクチャを実現した各システムでは， 8K バイト， 16K バイト， 32K バイト，または 64K バイトのページをサポートします。 VAX ページ・サイズに依存する部分を検出するには，“512”や“511”(または 16 進数の“200”) というテキスト文字列をアプリケーションから検索してください。

このページ・サイズの変更に対してアプリケーションでどのように対処するかについては，第 2 章を参照してください。

- アプリケーションに含まれる条件ハンドラを確認してください。 AXP システムの条件処理機能は， VAX システムの条件処理機能と同じ機能を実行しますが，メカニズム・アレイの形式など，部分的に変更されている箇所があります。さらに，算術演算例外の報告方法が変更されました。この問題についての詳しい説明は第 5 章を参照してください。
- AST パラメータ・リストに依存する部分を確認してください。 AXP システムの AST パラメータ・リストは VAX システムと同じ形式ですが， AST パラメータ・フィールドだけが使用されます。 AST パラメータ・リスト内の他のフィールド (R0， R1，プログラム・カウンタ[PC]，およびプロセッサ・ステータス[PS]の内容) は互換性を維持するためだけに提供され， AST プロシージャが終了した後は使用されません。

1.4 AXP 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 AXP 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。

AXP 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** (shareable image) **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。 AXP 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。AXP 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**—AXP 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。AXP 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 AXP システム固有の修飾子とオプションをサポートします。これらは表 1–1，表 1–2 に示すとおりです。この表にはまた，VAX システムでサポートされ，AXP システムのリンカでサポートされないリンカ修飾子も示されています。

表 1–1 AXP システム固有のリンカ修飾子

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。
/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	作成中のイメージ内で，コンパイラが作成したプロシージャ・シグナチャ・ブロック (PSB) 情報を渡さないことをリンカに要求する。省略時の設定では，リンク操作に対する入力ファイルとして指定したオブジェクト・モジュールに PSB 情報が含まれている場合には，リンカはイメージに PSB 情報を含む。イメージ・アクティベータはこの情報を使用して，ジャケット・ルーチンを作成する。ネイティブな AXP イメージがトランスレートされた VAX イメージと協調動作するには，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)。
/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。

表 1–2 AXP システム固有のオプション

オプション	説明
BASE オプション	AXP システムではサポートされない。
DZRO_MIN オプション	AXP システムではサポートされない。
ISD_MAX オプション	AXP システムではサポートされない。
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	AXP 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。
UNIVERSAL オプション	AXP システムではサポートされない。

1.5 VAX システムと AXP システムの算術演算ライブラリ間の互換性

OpenVMS Mathematics (MTH$) ランタイム・ライブラリに対して標準的な VMS 呼び出しインターフェイスを使用する算術演算アプリケーションを，AXP システムに移行するときには，MTH$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$ルーチンを Digital Portable Mathematics Library (DPML) for AXP システムの対応する math$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 FORTRAN や C 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。

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

例 1–1 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 2.4 節を参照してください。その例では，AXP 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 1–1 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_AXP
$ ON_VAX:
$ !
$ ! Do VAX-specific processing
$ !
$ exit
$ ON_ALPHA_AXP:
$ !
$ ! Do Alpha AXP-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 1–3 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 1–3 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。 AXP システムでは 2 を戻す。 AXP 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， AXP マシンでは“Alpha”というテキスト文字列を戻す。 AXP 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 AXP システムを示す。

(次ページに続く)

表 1–3 (続き) 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
CPU	CPU を識別する整数を戻す。 128 という値は AXP システムを識別する。

2

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。

2.1 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 AXP 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。

2.1.1 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，AXP 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 AXP 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

AXP 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。**

2.1.2 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 AXP 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 AXP 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。 AXP 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 2–1 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 AXP 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 2–1 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		AXP システムの動作
Adjust Working Set Limit ($ADJWSL)		
pagcnt	現在のワーキング・セット・リミットに加算される (またはワーキング・セット・リミットから減算される) ページ数を指定する。	値をページレットとして解釈し，CPU 固有のサイズのページを表現するために切り上げるか，または切り捨てる。
wsetlm	現在のワーキング・セット・リミットの値を指定する。	値をページレットとして解釈し，CPU 固有のサイズのページを表現するために切り上げるか，または切り捨てる。
Create Process ($CREPRC)		
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)		
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。

(次ページに続く)

表 2–1 (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		AXP システムの動作
Create and Map Section ($CRMPSC)		
inadr	再びマッピングされる領域を定義する先頭アドレスと末尾アドレスを指定する。末尾アドレスが先頭アドレスと同じである場合には，1 ページがマッピングされる。ただし，SEC$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$M_EXPREG フラグが設定されていない場合)。切り上げや切り捨ては実行されない (マッピングについての詳しい説明は第 2.3 節を参照)。
retadr	呼び出しの影響を受けるメモリの実際の先頭アドレスと末尾アドレスを指定する。	使用可能なアドレス範囲の先頭アドレスと末尾アドレスを戻す。これはマッピングされた合計サイズと異なる可能性がある。relpag 引数を指定した場合には，この引数も指定しなければならない。
flags	作成またはマッピングされるセクションのタイプと属性を指定する。	フラグ・ビット SEC$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 2–1 (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		AXP システムの動作
Create and Map Section ($CRMPSC)		
relpag	セクション・ファイルのマッピングを開始するページ・オフセットを指定する。	セクション・ファイルへのインデックスとして解釈され，単位はページレットであると解釈される。
pagcnt	マッピングされるファイル内のページ数 (ブロック数) を指定する。	ページレットとして解釈される。切り上げや切り捨ては実行されない。フラグ・ビット SEC$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， AXP 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 ($DELTVA)		
inadr	割り当てが解除されるメモリの先頭アドレスと末尾アドレスを指定する。	CPU 固有のページ境界にアラインされるように，アドレスは切り上げられるか，または切り捨てられる。
retadr	削除されたメモリの実際の先頭アドレスと末尾アドレスを指定する。	変更されない。
Expand Program/Control Region ($EXPREG)		
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。

(次ページに続く)

表 2–1 (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		AXP システムの動作
Get Job/Process Information ($GETJPI)		
itmlst	プロセスに関して戻される情報を指定する。	JPI$_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 ($GETQUI)		
itmlst	func 引数によって指定された関数を実行するときに使用される情報を指定する。	いくつかの項目はページレット単位の値として解釈される。詳しくは『OpenVMS System Services Reference Manual』を参照。
Get Systemwide Information ($GETSYI)		
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)		
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)		
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。

(次ページに続く)

表 2–1 (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		AXP システムの動作
Lock Working Set ($LKWSET)		
inadr	ロックされるメモリの先頭アドレスと末尾アドレスを指定する。	CPU 固有のページ境界にアラインされるように，アドレスは切り上げられるか，または切り捨てられる。
retadr	実際にロックされたメモリの先頭アドレスと末尾アドレスを指定する。	変更されない。
Map Global Section ($MGBLSC)		
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 2.3 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 ($PURGWS)		
inadr	パージされるメモリの先頭アドレスと末尾アドレスを指定する。	CPU 固有のページ境界にアラインされるように，アドレスは切り上げられるか，または切り捨てられる。
Set Protection ($SETPRT)		
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。

(次ページに続く)

表 2–1 (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		AXP システムの動作
Set User Authorization File ($SETUAI)		
itmlst	ユーザのユーザ登録ファイルからどの情報を設定するかを指定する。	いくつかの項目はページレット単位の値として解釈される。詳しくは『OpenVMS System Services Reference Manual』を参照。
Send to Job Controller ($SNDJBC)		
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 ($ULKPAG)		
inadr	アンロックされるメモリの先頭アドレスと末尾アドレスを指定する。	CPU 固有のページ境界にアラインされるように，アドレスは切り上げられるか，または切り捨てられる。
retadr	アンロックされたメモリの実際の先頭アドレスと末尾アドレスを指定する。	変更されない。
Unlock Working Set ($ULWSET)		
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。

(次ページに続く)

表 2–1 (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		AXP システムの動作
Update Section ($UPDSEC)		
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 2–2 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 2–2 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

アドレス範囲引数の機能	OpenVMS AXP 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。

2.2 メモリ割り当てルーチンの確認

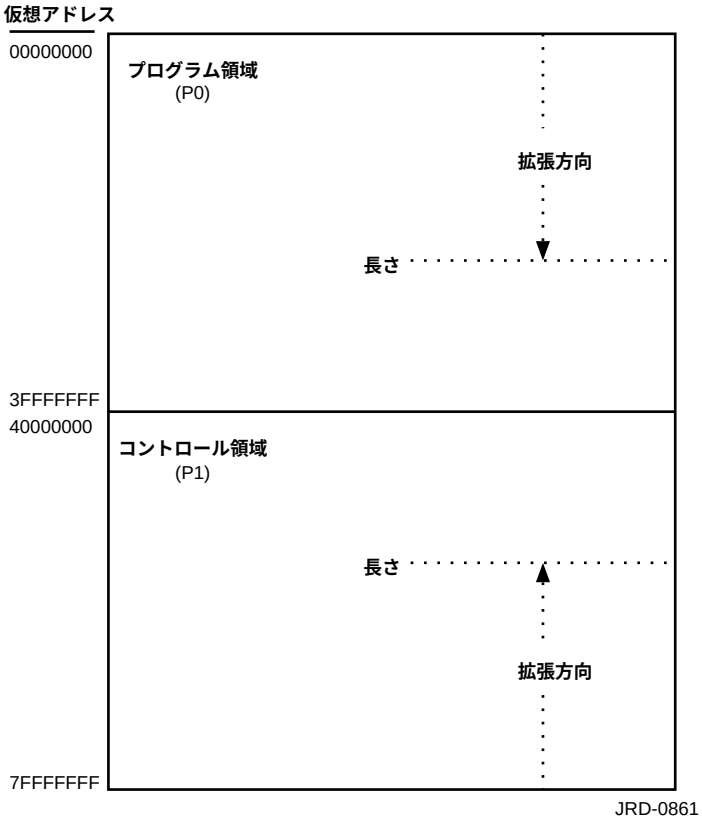
アプリケーションが実行するメモリ割り当てを変更しなければならないかどうかを判断するには，メモリがどこで割り当てられるかを確認しなければなりません。メ

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 AXP 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 2–1 はこのレイアウトを示しています。

図 2–1 仮想アドレスのレイアウト



2.2.1 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 AXP 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。

- AXP 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。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 AXP 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，$EXPREG システム・サービスが戻すメモリ・サイズは， Alpha AXP アーキテクチャを実現した各システムで異なる可能性があるため，システムが割り当てた正確なメモリ境界を確認しておくことが適切でしょう。正確なメモリ境界を確認するには，$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 2–1 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 AXP 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 AXP 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 2–1 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 2–1 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。

2.2.2 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。 AXP 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， AXP 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 2–2 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 2–2 既存のアドレス空間でのメモリの割り当て

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

char _align(page) buffer[1024];

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

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

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

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

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

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

2.2.3 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。

2.3 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 AXP 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 2.3.2 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。

2.3.1 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 AXP 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，AXP 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** 2.3.4 **項を参照してください。**

例 2–3 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 2–3 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;

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

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

     status = sys$create( &fab );

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

     fileChannel = fab.fab$l_stv;
```

(次ページに続く)

例 2–3 (続き) 拡張された仮想アドレス空間へのセクションのマッピング

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

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

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

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

}
```

2.3.2 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，AXP 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。AXP 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 AXP 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 2.3.1 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 2.3.3 項を参照してください。

2.3.3 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， AXP 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。 AXP 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，AXP 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。 AXP 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 2.3.4 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。 AXP 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。 AXP 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。 AXP 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 (AXP 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 2–4 に示す VAX プログラムは，第 2.3.1 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 2–4 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 2–4 (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]);
```

(次ページに続く)

例 2–4 (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

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

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

例 2–4 に示したプログラムを AXP 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 AXP 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 AXP 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。
- AXP 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 2.4 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 AXP 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 2–5 は， AXP システムで正しく実行するために例 2–4 に対して必要なソースの変更を示しています。

例 2–5 例 2–4 を AXP 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 2–5 (続き) 例 2–4 を AXP 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 );
     }
```

(次ページに続く)

例 2–5 (続き) 例 2–4 を AXP システムで実行するのに必要なソース・コードの変更

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

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

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

     status = SYS$CRMPSC(&inadr, /* inadr=address target for map */
                         &retadr, /* retadr= what was actually mapped */
                               0, /* acmode  */
                               0, /* noflags 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 2–5 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 AXP システムで実行するまで判断できないため，DEC C for OpenVMS AXP コンパイラは，__align(page) が指定されているときに，データを AXP 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。

2.3.4 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。

$CRMPSC システム・サービスは互換性を維持するために， VAX システムと AXP システムの両方のシステムにおいて，relpag 引数の値を 512 バイト単位で解釈します。しかし， AXP システムの CPU 固有のページ・サイズは 512 バイトより大きいため，relpag 引数にオフセットとして指定する値はおそらく CPU 固有のページ境界にアラインされません。$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， AXP 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。

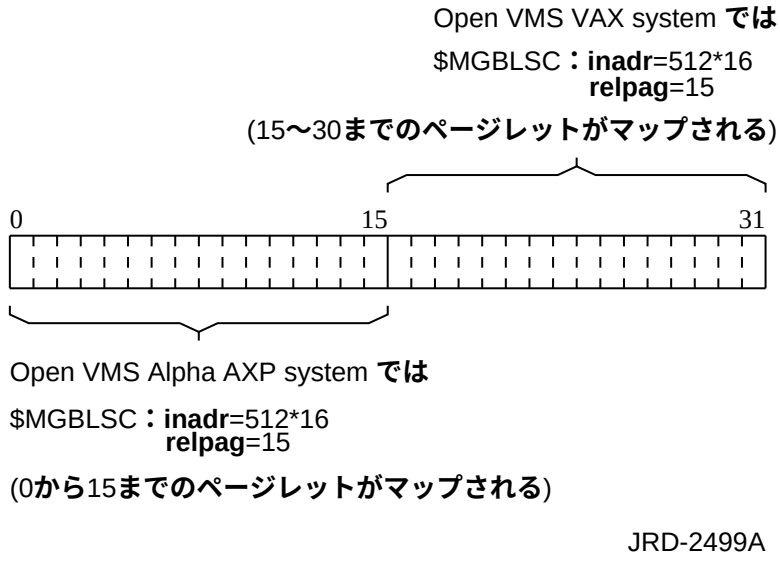
注意

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， AXP 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 AXP 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 AXP 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 2–2 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。 AXP システムでは，$CRMPSC システム・サービスはセクション・ファイルのマッピングを CPU 固有のページ境界から開始するため， 16 番目から 30 番目までのブロックを正しくマッピングできません。マッピングを正しく実行するには， AXP システムで$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 2–2 はこの状況を示しています。

図 2–2 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。

2.4 ページ・サイズの実行時確認

AXP 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 2–6 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 2–6 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  */
```

(次ページに続く)

例 2–6 (続き) CPU 固有のページ・サイズを確認するための$GETSYI システム・サービスの使用

```
    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 );
    }
}
```

2.5 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 AXP 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 AXP 命令は完全な仮想アドレスを指定できないため，AXP 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

AXP 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。

3

共有データの整合性の維持

この章では，共有データ (shared data) 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 (atomicity) などについて説明します。

3.1 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， AXP 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 3.1.1 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 AXP 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。 AXP 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 AXP アーキテクチャの機能については，第 3.1.2 項を参照してください。

3.1.1 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 AXP 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， AXP 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， AXP 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 AXP 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， AXP 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 3.1.2 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** (granularity)—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 AXP 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。 AXP 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， AXP 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。 AXP 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 3.1.2 項で説明する Alpha AXP 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。

- **リード/ライト (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。 AXP 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， AXP 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 AXP 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 3.1.2 項を参照してください。この同期をとるために Alpha AXP 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， AXP 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 3.3 節を参照してください。

3.1.2 OpenVMS AXP 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 AXP 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 AXP 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，AXP 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，AXP 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 AXP 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 AXP 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。

3.2 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。

AXP 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(unaligned)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

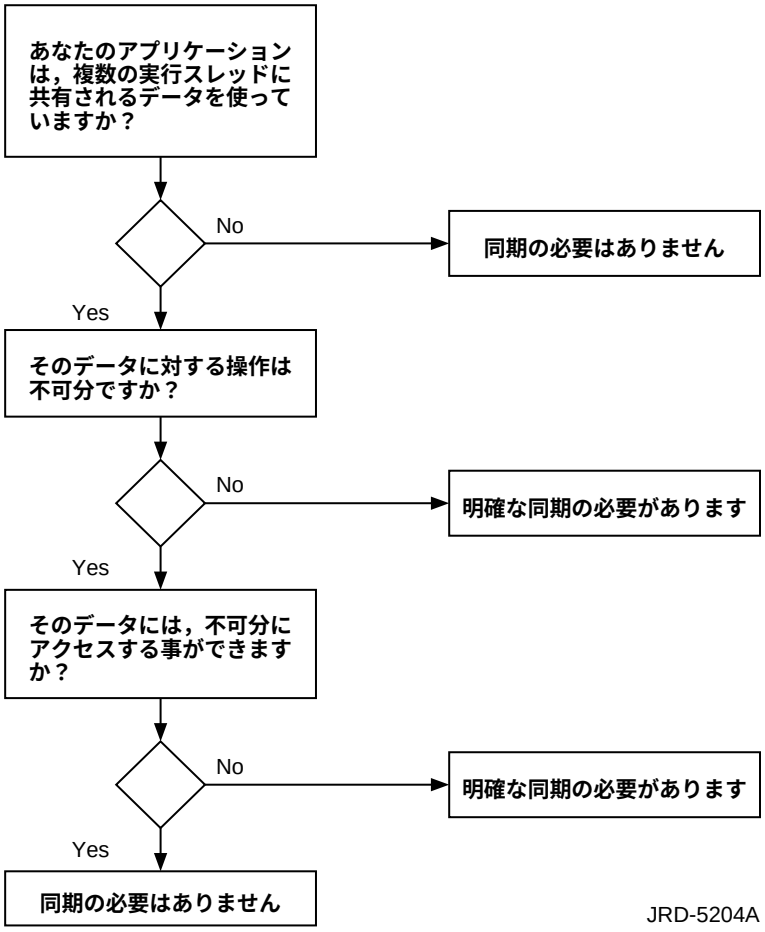
は，データは自然にアラインされています(naturally aligned)。詳しくは第 4 章を参照してください。コンパイラは省略時の設定により，宣言されたデータを自然な境界にアラインします)。

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 3–1 はこの判断を下す処理を示しています。

図 3–1 同期に関する判断



3.2.1 明示的に共有されるデータの保護

例 3–1 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 3–1 AST スレッドを含む VAX 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)

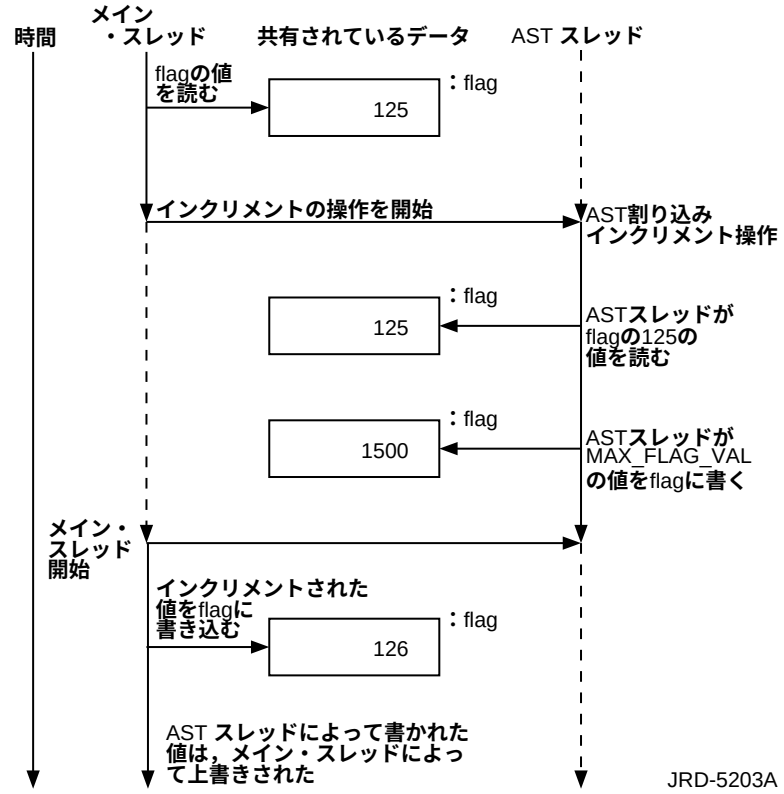
例 3–1 (続き) AST スレッドを含む VAX プログラムにおける不可分な処理への依存

```
    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 3–1 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。

AXP 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 3–2 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 3–2 例 3–1 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 AXP 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 3–1 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 AXP 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 3–2 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。 AXP 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 3–2 は，例 3–1 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 3–2 例 3–1 の同期バージョン

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

(次ページに続く)

例 3–2 (続き) 例 3–1 の同期バージョン

```
#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 );
     }

     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");
}
```

(次ページに続く)

例 3–2 (続き) 例 3–1 の同期バージョン

```
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 3–2 に示した番号に対応しています。

1. DEC C for OpenVMS AXP 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。

3.2.2 無意識に共有されるデータの保護

例 3–1 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，AXP 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。AXP 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 4 章を参照してください)。

この問題を示すために，例 3–1 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，AXP 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 AXP 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。

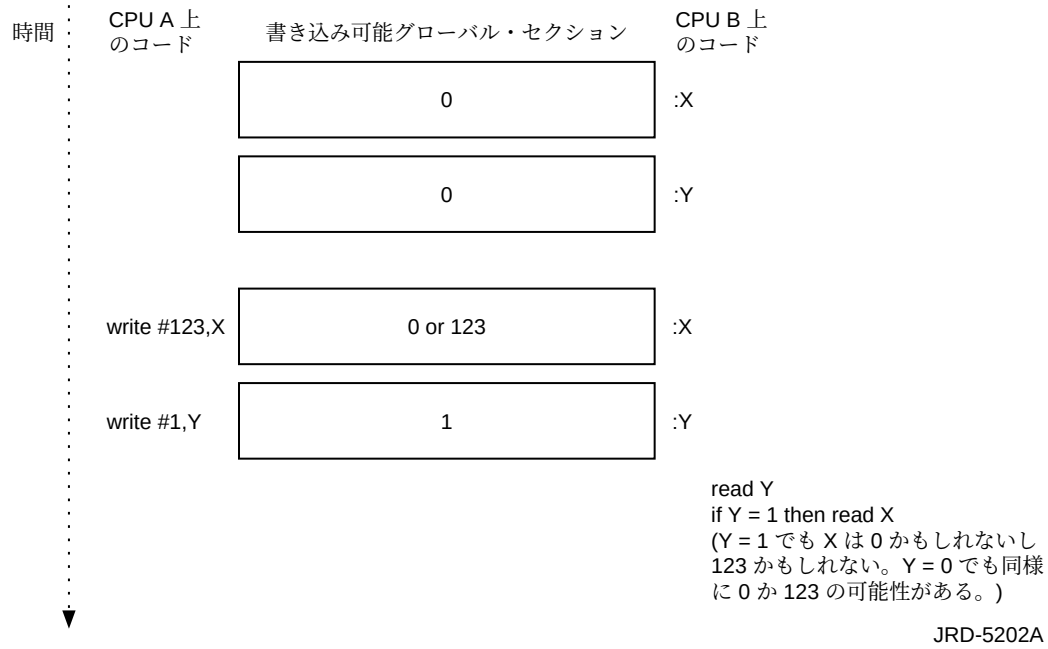
3.3 読み込み/書き込み操作の同期

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 3–3 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 3–3 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。

AXP 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 3–3 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 3–3 AXP 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，AXP 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 AXP 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 AXP 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。
- PPL$ランタイム・ライブラリ提供されるメモリ・インターロックを使用するか，または LIB$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。

3.4 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， AXP 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。

4

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 AXP 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。

4.1 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 (portability) 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 AXP 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 AXP 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。 AXP 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 4.2 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 AXP 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。 AXP 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 4.3 節では，この問題について詳しく説明します。

4.2 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 AXP 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 4–1 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 4–1 VAX と AXP のネイティブなデータ型の比較

VAX データ型	AXP 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 (packed decimal) 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， AXP 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 AXP 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 AXP データ型にマッピングします (AXP 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 AXP 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，付録 A 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** (D-floating) **データ**— 大部分の AXP 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 AXP 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 AXP 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 AXP 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 AXP 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。 AXP 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。 AXP 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 4–1 を参照してください。

例 4–1 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 4–1 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 AXP 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;
```

4.3 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 AXP 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 AXP 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 AXP 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。

4.3.1 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。

AXP 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 AXP 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， AXP 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， AXP 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。

4.3.2 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 (data alignment) 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 AXP 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。 AXP システムではこのようなハードウェアの支援はありません。したがって，アラインされていないデータを参照すると，フォルトが発生し，オペレーティング・システムのアラインメント・フォルト・ハンドラ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， AXP システムではきわめて大きくなります。

AXP 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・

† Alpha AXP アーキテクチャは，アラインされていないデータを操作する命令を持っていません。もしコード中にアラインされていないデータに対する参照が存在し，それが実行されたときは，フォルトが発生します。このフォルトを**アラインメント・フォルト**とよびます。

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** (Interoperability) **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。

- AXP 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。たとえば，多くの AXP システムのコンパイラは/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 Debugger Manual』を参照してください。
- 相互操作性の問題がない場合には， AXP 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。 AXP 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。

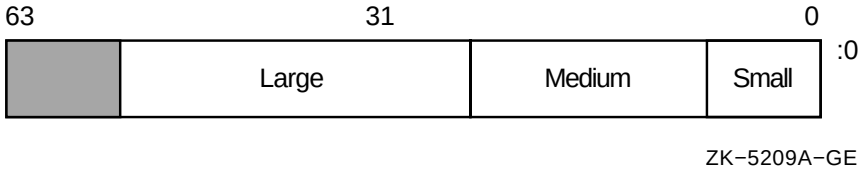
  AXP システムのコンパイラは， VAX システムと同様のバイト・アラインメントの使用を要求することを許可する修飾子や言語プログラムをサポートしています。たとえば， DEC C for OpenVMS AXP システムのコンパイラは/NOMEMBER_ALIGNMENT 修飾子をサポートし，また，それに対応した，データ・アラインメントの制御を許可するプラグマもサポートします。詳しくは， DEC C のコンパイラ解説書を参照してください。

例 4–1 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 ;
```

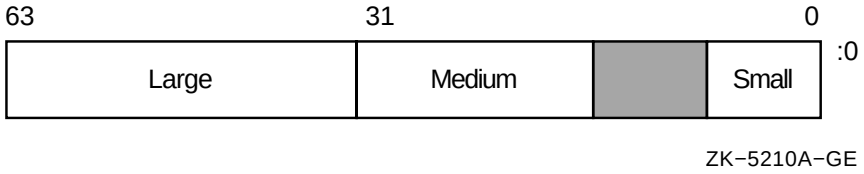
VAX Cを使用してコンパイルした場合には，この構造体は図 4–1 に示すようにメモリにレイアウトされます。

図 4–1 VAX Cの使用による mystruct のアラインメント



DEC C for OpenVMS AXP 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 4–2 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 4–2 DEC C for OpenVMS AXP 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。

5

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 AXP 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。

5.1 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 AXP 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 AXP 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

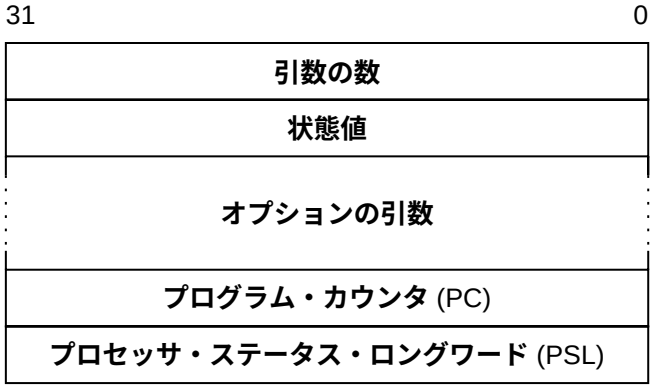
この後の節では，これらの変更について詳しく説明し，ソース・コードの変更が必要かどうかを判断するのに役立つガイドラインも示します。

5.2 条件処理ルーチンがアーキテクチャ固有の機能に依存しているかどうかの確認

ユーザ作成条件処理ルーチンの呼び出しシーケンスは，AXP 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。 AXP 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 5–1 を参照してください。

図 5–1 VAX システムと AXP システムでのシグナル・アレイ



JRD-5208A

表 5–1 はシグナル・アレイ内の各引数を示しています。

表 5–1 シグナル・アレイ内の引数

引数	説明
引数の数	AXP 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。
状態値	AXP 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 AXP 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，AXP 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。 AXP 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 5.3 節を参照。
オプションの引数	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。 AXP 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 (AXP 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。 AXP システムでは，この引数には AXP 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。

AXP 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。 AXP 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， AXP 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 5–2 は VAX システムと AXP 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 5–2 VAX システムと AXP システムでのメカニズム・アレイ

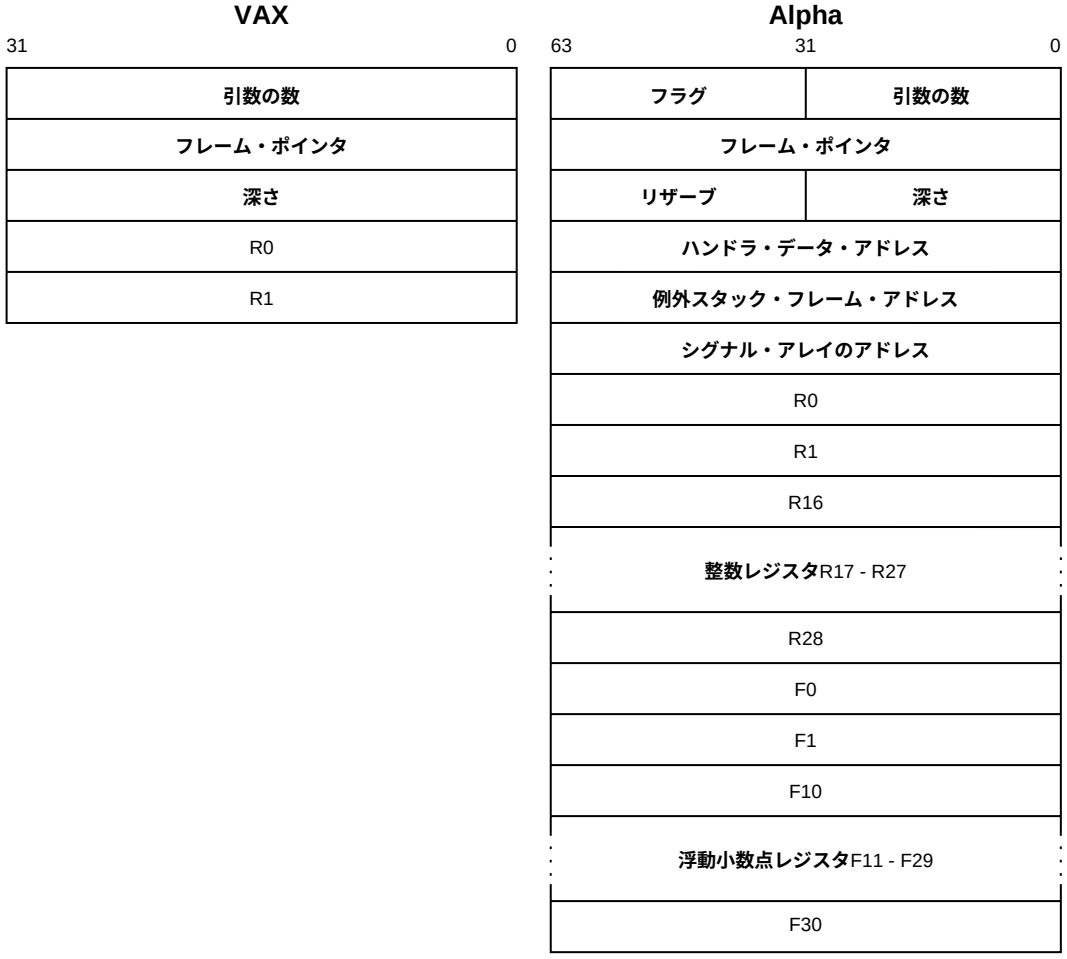


表 5–2 はメカニズム・アレイ内の各引数を示しています。

表 5–2 メカイズム・アレイの引数

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。 AXP 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 (AXP システムでは常に 43) を示すわけではない。
フラグ	AXP 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 AXP 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。
深さ	VAX システムでも AXP 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	AXP 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	AXP 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	AXP 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 AXP 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。 AXP 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

シグナル・アレイは AXP 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， AXP 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 5–1 は C で作成した条件処理ルーチンを示しています。

例 5–1 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  */
```

(次ページに続く)

例 5–1 (続き) 条件処理ルーチン

```
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 5–1 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。

5.3 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 5–1 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 AXP 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 AXP 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， AXP 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， AXP 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 AXP システムでもサポートされます。さらに， Alpha AXP 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 5–3 は， AXP 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 AXP 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 AXP 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 (AXP 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 5.3.1 項を参照してください)。

注意

AXP 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 5–3 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
AXP システム固有の例外	
SS$_HPARITH – 高性能算術演算例外	VAX 算術演算例外はこの例外に変更された (第 5.3.1 項を参照)。
SS$_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$_ARTRES – 予備の算術演算トラップ	AXP システムには対応する例外はない
SS$_COMPAT – 互換性フォルト	AXP システムには対応する例外はない
[1]SS$_DECOVF –10 進オーバーフロー	SS$_HPARITH に変更された (第 5.3.1 項を参照)
[1]SS$_FLTDIV –0 による浮動小数点除算 (トラップ)	SS$_HPARITH に変更された (第 5.3.1 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 5.3.1 項を参照)
[1]SS$_FLTOVF – 浮動小数点オーバーフロー (トラップ)	SS$_HPARITH に変更された (第 5.3.1 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 5.3.1 項を参照)
[1]SS$_FLTUND – 浮動小数点アンダーフロー (トラップ)	SS$_HPARITH に変更された (第 5.3.1 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 5.3.1 項を参照)
[1]SS$_INTDIV –0 による整数除算	SS$_HPARITH に変更された (第 5.3.1 項を参照)
[1]SS$_INTOVF – 整数オーバーフロー	SS$_HPARITH に変更された (第 5.3.1 項を参照)
SS$_TBIT – トレース・ペンディング	AXP システムには対応する例外はない

[1]AXP 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。

(次ページに続く)

表 5–3 (続き) 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
VAX システム固有の例外	
SS$_OPCCUS – ユーザ用に確保されているオペコード	AXP システムには対応する例外はない
SS$_RADMOD – 予備のアドレッシング・モード	AXP 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	AXP システムには対応する例外はない

5.3.1 AXP 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。

AXP 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， AXP 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。 AXP 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， AXP 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 5–3 は SS$_HPARITH 例外シグナル・アレイの形式を示しています。

図 5–3 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** (integer register write mask)，**浮動小数点レジスタ・ライト・マスク** (floating register write mask)，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 5–4 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 5–4 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 AXP 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， AXP 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 (これらのコンパイラ修飾子についての詳しい説明は付録 A を参照してください)。しかし，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。
- 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

AXP 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。

5.3.2 データ・アラインメント・トラップのテスト

AXP 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 AXP 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 4 章を参照してください)。

AXP 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$_ALIGN によって示されます。図 5–4 は，SS$_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 5–4 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。

5.4 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 AXP 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 (calling standard) 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 5–5 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， AXP 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 5–5 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	AXP システムで使用できるかどうか
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

(次ページに続く)

表 5–5 (続き) 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	AXP システムで使用できるかどうか
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 AXP 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 AXP 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$ESTABLISH を呼び出すことにより，条件処理ルーチンを指定する場合には，ソース・コードを変更する必要はありません。 AXP システムの大部分のコンパイラは互換性を維持するために， LIB$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 5–2 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 AXP 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 5–2 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

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 5–2 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 AXP 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 5–2 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
```

6

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 AXP イメージの作成方法について説明します。

6.1 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 AXP イメージに変換する方法を説明しています (DECmigrate for OpenVMS AXP 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 AXP 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 AXP 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。

6.1.1 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 AXP 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)， AXP 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 AXP 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。

6.1.2 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 AXP 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 AXP 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/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 6.3 節を参照してください。

6.2 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 AXP 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**

   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 AXP バージョンを作成します**

OpenVMS AXP 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します**

   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 6–1 と例 6–2 に示したプログラムについて考えてみましょう。例 6–1 は，例 6–2 に定義されている mysub というルーチンを呼び出します。

例 6–1 **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 6–2 **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 );
}
```

(**次ページに続く**)

例 6–2 (続き) 共有可能イメージ (MYMATH.C) のソース・コード

```
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 6–2 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 AXP 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 AXP 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，AXP コードを生成するコンパイラを使用して例 6–1 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 AXP 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
```

(/NONATIVE_ONLY は省略時の設定であるため，この修飾子を特に指定する必要はありません。)

ネイティブなメイン・イメージは他の AXP 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
```

6.3 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 AXP 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 6.3.1 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** AXP **バージョンを作成します**

   AXP 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** AXP **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** AXP **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 6–2 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** AXP **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 6.3.1 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 AXP 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 AXP 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
```

6.3.1 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 AXP 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 AXP イメージであり，他の AXP 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 AXP 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                           0000000c +S +G 00000020 1  00 4e
MYADD 2                         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)。

6.3.2 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 AXP 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 AXP 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。

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 AXP 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 AXP 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。

A

OpenVMS AXP コンパイラ

この付録では，ネイティブな OpenVMS AXP コンパイラ固有の機能について説明します。さらに，この付録では， OpenVMS VAX コンパイラの機能のうち，OpenVMS AXP 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。

以下に付録 A で説明するコンパイラと，その節番号をアルファベット順に示します。

- DEC Ada (第 A.1 節)
- DEC C (第 A.2 節)
- DEC COBOL (第 A.3 節)
- DEC Fortran (第 A.4 節)
- DEC Pascal (第 A.5 節)

A.1 DEC Ada の AXP 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。

- 『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 AXP 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。**

A.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， AXP 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

  AXP 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

AXP 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 AXP システムでも2^9です。

A.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。

A.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

  AXP 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

  AXP 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

  AXP 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

  AXP システムではサポートされません。

- SHARE_GENERIC プラグマ

  AXP 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 AXP 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。

A.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

  AXP 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， AXP 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， AXP 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

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

- SYSTEM.TICK

  AXP 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， AXP 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_INTERRLOCKED
- SYSTEM.INSQ_STATUS
- SYSTEM.REMQ_STATUS
- SYSTEM.INSQHI
- SYSTEM.REMQHI
- SYSTEM.INSQTI
- SYSTEM.REMQTI

A.1.5 他の言語パッケージでの相違点

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 AXP 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。

A.1.6 あらかじめ定義されている具現に対する変更

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具現は AXP 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

A.2 DEC C for OpenVMS AXP システムとVAX Cとの互換性

Alpha AXP アーキテクチャをサポートするために，DEC C と総称する C コンパイラ群に 1 つのコンパイラが追加されました。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 AXP 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。

A.2.1 言語モード

DEC C for OpenVMS AXP 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 A–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 A–1 DEC C for OpenVMS AXP **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，DEC 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。
VAX CとCommon Cの組み合わせ	/STANDARD=(VAXC,COMMON)	ANSI C 標準規格の他に，VAX CとCommon C の両方の拡張機能を使用できる。これらの拡張機能が ANSI 標準規格と互換性がない場合でも使用できる。

A.2.2 DEC C for OpenVMS AXP システムのデータ型のマッピング

DEC C for OpenVMS AXP 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 A–2 は， Alpha AXP 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 A–2 DEC C for OpenVMS AXP 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・ライン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・ライン修飾子を使用することにより， AXP 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 A.2.2.1 項を参照。

移植性を向上するために， DEC C for OpenVMS AXP 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。これらのマクロは， int64 などの汎用データ型名を，–64などのマシン固有のデータ型にマッピングします。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。

A.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 AXP の浮動小数点データ型の間のマッピングは，コマンド・ライン修飾子によって制御されます。 Alpha AXP 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)
- T 浮動小数点 (IEEE 倍精度)

コマンド・ライン修飾子を使用すれば，標準的な C データ型の float と double が AXP 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 AXP の F 浮動小数点データ型にマッピングし， double データ型を AXP の G 浮動小数点データ型にマッピングします。表 A–3 は浮動小数点オプションを示しています。各コマンド・ラインに浮動小数点修飾子は 1 つだけ指定できます。

表 A–3 DEC C for OpenVMS AXP コンパイラの浮動小数点マッピング

コンパイラ・オプション	Float	Double
/FLOAT=F_FLOAT	F 浮動小数点フォーマット	G 浮動小数点フォーマット
/FLOAT=D_FLOAT	F 浮動小数点フォーマット	D-53 浮動小数点
/FLOAT=IEEE_FLOAT	S 浮動小数点フォーマット	T 浮動小数点フォーマット

A.2.3 AXP システム固有の機能

DEC C には，表 A–4 に示す機能があり，これらの機能は AXP 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 A–4　OpenVMS AXP **システム固有の** DEC C **コンパイラ機能**

機能	説明
一部の Alpha AXP 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	AXP 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

A.2.3.1　Alpha AXP 命令のアクセス

DEC C for OpenVMS AXP コンパイラは，C 言語で表現できない機能を提供するために，ある種の Alpha AXP 命令をサポートします。特に，システム・レベル・プログラミングの場合には，これらの命令を使用すると便利です。現在のところ，DEC C は次の Alpha AXP 命令をサポートする予定です。

- 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。

A.2.3.2　Alpha AXP 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 AXP 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 は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 はエントリをキューに登録し，インターロックします。
- REMQUEx は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 はエントリをキューから削除し，インターロックします。

しかし，VAX Cで組み込み機能としてサポートされる次の VAX 命令は， DEC C では組み込み機能としてサポートされません。

ADAWI	BBCCI	BBSSI	FFC
FFS	LDPCTX	LOCC	MFPR
MTPR	MOVC3	MOVPSL	PROBER
PROBEW	READ_GPR	SCANC	SIMPLE_READ
SKPC	SPANC	SCSVPCTX	WRITE_GPR

A.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)。 AXP 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 A–5 はこれらの不可分な組み込み機能を示しています。

表 A–5 DEC C for OpenVMS AXP コンパイラの不可分性を保証する組み込み機能

不可分性を保証する組み込み機能	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 AXP 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。

A.2.4 VAX Cと DEC C for OpenVMS AXP 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 AXP 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・ライン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。

A.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，AXP システムでは性能が著しく低下するため，DEC C for OpenVMS AXP 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 nomember_alignment プラグマを指定するか，または/NOMEMBER_ALIGNMENT コマンド・ライン修飾子を使用します。

A.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 AXP 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。

A.2.4.3 例外の同期化

Alpha AXP 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)。

A.2.5 /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 AXP 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 AXP 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。

A.3 DEC COBOL とVAX COBOL の互換性

OpenVMS AXP システムで動作する DEC COBOL バージョン 1.0 コンパイラは，OpenVMS AXP システムで動作するVAX COBOL バージョン 4.4 コンパイラに基づいており，高い互換性があります。 DEC COBOL コンパイラはVAX COBOL の機能のうち，すべてではありませんが多くをサポートしています。以下に， DEC COBOL コンパイラとVAX COBOL コンパイラのおもな相違点の要約を示します。

- パフォーマンスを最適化する Alpha AXP データ・アラインメントか，VAX COBOL レコード・アラインメントで互換性を保証するVAX COBOL データ・アラインメントかを選択する，新しいアラインメント修飾子
- 単精度および倍精度データに， IEEE または VAX 浮動小数点データ型のどちらかを指定する新しい修飾子
- ネイティブなイメージがトランスレートされたイメージを，またはトランスレートされたイメージがネイティブなイメージを呼ぶことを許可するコードを生成する，新しい修飾子
- X/Open ポータビリティ・ガイドで追加された COBOL の予約語を認識する新しい修飾子
- ACCEPT/DISPLAY のための新しいスクリーン・マネージャ
- VAX COBOL の/STANDATD=V3 修飾子オプションの最も重要な機能のみのサポート
- VAX DBMS (Database Management System) Data Manipulation Language (DML) をサポートしない
- VAX COBOL バージョン 5.0 以降に含まれる組み込み関数をサポートしない

- VAX COBOL(日本語版) バージョン 5.0 以降に含まれるマルチ・バイト文字およびその他の日本語機能をサポートしない
- 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する。これはバージョン 5.0 以前のVAX COBOL とは異なる値を返す

この節の内容は，既存の COBOL アプリケーションをVAX COBOL から DEC COBOL へ変換する手引きになるのと同様に，VAX COBOL および DEC COBOL 間で互換性を持つ COBOL アプリケーションを作成する手引きとなります。

この節では，VAX COBOL バージョン 4.4 と DEC COBOL バージョン 1.0 の相似点と相違点について説明します。 DEC COBOL とバージョン 4.4 以降のVAX COBOL との相違点は，その都度示します。

DEC COBOL コンパイラの機能と将来のバージョン・アップについての最新情報は， DEC COBOL の最新のリリース・ノートをご覧ください。VAX COBOL の機能についての情報は，VAX COBOL のリリース・ノートおよびその他の解説書をご覧ください。現在インストールされている COBOL コンパイラのリリース・ノートの概要は， DCL プロンプトで HELP COBOL RELEASE_NOTES コマンドを入力すると見ることができます。

DEC COBOL 言語の機能に関するリファレンス情報は，『DEC COBOL Reference Manual』をご覧ください。VAX COBOL 言語の機能に関するリファレンス情報は，『VAX COBOL Reference Manual』をご覧ください。DEC COBOL のコマンド・ライン修飾子に関する情報は，オペレーティング・システム・プロンプトで COBOL のオンライン・ヘルプを起動してください。VAX COBOL のコマンド・ライン修飾子に関する情報は，『VAX COBOL User Manual』をご覧ください。

A.3.1 コマンド・ライン修飾子

表 A–6，表 A–7 および表 A–8 は， DEC COBOL とVAX COBOL のコマンド・ライン修飾子の対比を示しています。

A.3.1.1 DEC COBOL とVAX COBOL が共有する修飾子

表 A–6 は， DEC COBOL とVAX COBOL が共有するコマンド・ライン修飾子を示しています。 DEC COBOL で有効なコマンド・ライン修飾子についての詳細は，表 A–7 を参照するか， DEC COBOL のオンライン・ヘルプ・システムを起動してください。VAX COBOL で有効なコマンド・ライン修飾子について詳しくは，表 A–8 か『VAX COBOL User Manual』を参照してください。

表 A–6 DEC COBOL とVAX COBOL が共有する修飾子

修飾子	説明
/ANALYSIS_DATA	等しい
/ANSI_FORMAT	等しい
/AUDIT	等しい
/CHECK	新しいオプション (/CHECK=[NO]DECIMAL) が DEC COBOL で有効です (表 A–7 および第 A.3.2.2 項を参照してください)
/CONDITIONALS	等しい
/COPY_LIST	等しい
/CROSS_REFERENCE	等しい
/DEBUG	等しい
/DEPENDENCY_DATA	等しい
/DIAGNOSTICS	等しい
/FIPS	機能的に小さな違いがあります (/FIPS=74 オプションの動作については， DEC COBOL のオンライン・ヘルプ・システムを起動してください)
/FLAGGER	等しい
/LIST	等しい
/MACHINE_CODE	等しい
/MAP	等しい
/OBJECT	等しい
/SEQUENCE_CHECK	等しい

(次ページに続く)

表 A–6 (**続き**) DEC COBOL とVAX COBOL **が共有する修飾子**

修飾子	説明
/STANDARD	いくつかのVAX COBOL オプションは， DEC COBOL でも有効です。 (/STANDARD=V3 オプションについては，第 A.3.2.7 項を参照してください)
/TRUNCATE	等しい
/WARNINGS	機能的に小さな違いがあります (/WARNINGS 修飾子の動作については，第 A.3.2.7.2 項を参照するか，DEC COBOL のオンライン・ヘルプ・システムを起動してください)

A.3.1.2 VAX COBOL で使えない DEC COBOL 修飾子

表 A–7 は， DEC COBOL 固有の修飾子とオプションを示します。これらの修飾子とオプションは，VAX COBOL では使えません。 DEC COBOL で有効なコマンド・ライン修飾子について詳しくは， DEC COBOL のオンライン・ヘルプ・システムを起動してください。

表 A–7 VAX COBOL **で使えない** DEC COBOL **修飾子**

修飾子	説明
/ALIGNMENT=([NO]BINARY, [NO]DECIMAL)	数値項目に対するアラインメントを指定する (第 A.3.2.1 項を参照)
/CHECK=[NO]DECIMAL	数値が扱われる状況で表示用数字項目を使用した場合，項目の内容が数値として妥当かどうかを検査する (第 A.3.2.2 項を参照)
/CONVERT=LEADING_ BLANKS	表示用数字項目で先行する空白文字の代りに 0(ゼロ) を使用する (第 A.3.2.3 項を参照)
/FLOAT=[D_FLOAT],[IEEE_ FLOAT]	メモリ内で使用される浮動小数点データ型の表現形式として， VAX 浮動小数点データ型式か IEEE 形式かを指定する (第 A.3.2.4 項を参照)
/OPTIMIZE	最も効率の良いコードを生成するようにプログラムを最適化してコンパイルすることをコンパイラに指示する (第 A.3.2.5 項を参照)

(**次ページに続く**)

表 A–7 (続き) VAX COBOL で使えない DEC COBOL 修飾子

修飾子	説明
/RESERVED_WORDS=[NO]XOPEN	コンパイラが X/Open COBOL で定義された語を予約語として認識するかどうかを制御する (第 A.3.2.6 項を参照)
/TIE	ネイティブなイメージがトランスレートされたイメージを，また，トランスレートされたイメージがネイティブなイメージを呼ぶことを許可するコードを生成する (第 A.3.2.8 項を参照)

A.3.1.3 DEC COBOL で使用できないVAX COBOL 修飾子

表 A–8 は，VAX COBOL 固有の修飾子とオプションを示しています。これらの修飾子とオプションは DEC COBOL では使えません。VAX COBOL コマンド・ライン修飾子について詳しくは，『VAX COBOL User Manual』を参照してください。

表 A–8 DEC COBOL で使えないVAX COBOL 修飾子

修飾子	説明
/DESIGN	Language-Sensitive Editor (LSE) などを使用して作成されたプログラムのコンパイル処理を可能にする
/INSTRUCTION_SET[=option]	通常のコンパイルでは使用しない VAX 命令セットを使用して，コードの最適化を行うかどうかを指定する
/STANDARD=[NO]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 A.3.2.9 項および『VAX COBOL バージョン 5.1 リリース・ノート』を参照)
/STANDARD=[NO]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=[NO]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=[NO]SYNTAX である (第 A.3.2.7.2 項を参照)

A.3.2 動作の相違点

この節では，DEC COBOL バージョン 1.0 固有の動作や，DEC COBOL の新しいコマンド・ライン修飾子などのVAX COBOL バージョン 4.4 と DEC COBOL バージョン 1.0 の動作の相違点について説明します。

A.3.2.1 DEC COBOL の/ALIGNMENT 修飾子とアラインメント指示文による数値項目に対するアラインメントの指定

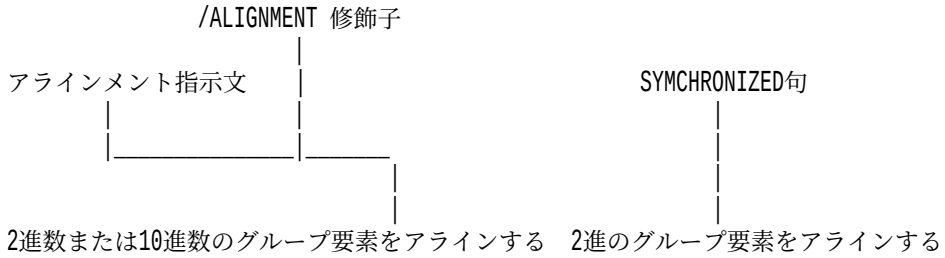
レコード内で 2 進 (バイナリ) または 10 進 (デシマル) データのアラインメントを指定するために，/ALIGNMENT 修飾子やアラインメント指示文を使用できます。アラインメントに関する詳細は『DEC COBOL Reference Manual』を参照してください。

Alpha AXP システムで COBOL アプリケーションを最適化するには，適切なデータ・アラインメントが必要です。データが自然な境界にある場合，2 進データを操作するほうが高速です。使用しているシステムでの適切な境界にデータを配置している場合には，10 進データの操作の方が高速なこともあります。

アラインメント指定の最大の特色は，パフォーマンスの高速化です。さらに，/ALIGNMENT 修飾子とアラインメント指示文は以下の特色を持っています。

- 容易な使用と変換 — 既存のソース・ファイルに最小の変更を加えればよい。いくつかの場合，DEC COBOL の起動時に/ALIGNMENT 修飾子を加えるだけですむ。
- VAX COBOL のソースの互換性 —VAX COBOL と DEC COBOL で同じソース・ファイルをコンパイルできる。DEC COBOL の指示文はVAX COBOL コンパイラではコメントとして無視される。
- 適応性 — レコード単位で VAX バイト・アラインメントまたは自然なアラインメントを指定できる。たとえば，両方のコンパイラで共有されるファイルに対してバイト・アラインメントを指定でき，DEC COBOL ファイルとレコードに対して自然なアラインメントを指定できる。

/ALIGNMENT 修飾子，アライメント指示文および SYNCHRONIZED 句は，以下に示すようなグループ (グループ要素) でデータのアラインメントに影響します。



VAX COBOL コンパイラを使えば，自然な境界にレコードの 2 進数の構成要素をアラインするために， SYNCHRONIZED 句を使用することができます。このように， DEC COBOL コンパイラのバイナリ・データの操作に対しては，SYNCHRONIZED 句，/ALIGNMENT 修飾子およびアラインメント指示文は同等の働きを示します。

A.3.2.1.1 /ALIGNMENT 修飾子の使用

/ALIGNMENT 修飾子は，プログラム内でバイナリ・データに対して自然なアラインメントを，またデシマル・データに対してより適切なアラインメントを指定することを許可します。

バイナリ・アラインメントとデシマル・アラインメントは別々のオプションです (10 進数と文字列データに別名をつけるプログラムには有効な機能ですが，そのようなプログラムはバイナリ・データのアラインメントで作成することもできます)。たとえば，/ALIGNMENT=(BINARY,NODECIMAL) (もしくは /ALIGNMENT のみ) を指定すると， DEC COBOL コンパイラはバイナリ・データを自然な境界に，またデシマル・データをバイト境界にアラインします。/ALIGNMENT を使用することで OpenVMS AXP システム上で最大限のパフォーマンスを得るようなデータ・アラインメントが行われます。

バイト・データ・アラインメントを指定するには/NOALIGNENT 修飾子 (省略時設定) を指定します (SYNCHRONIZED 句を用いてバイナリ・データをアラインするプログラムもこれに含まれます)。 VAX システムで作成されたデータ・ファイルを移行したり，互換性を得るためにも/NOALIGNMENT を使用します。

A.3.2.1.2 アラインメント指示文の使用

/ALIGNMENT 修飾子を使用した場合，アラインメント指示文で指定された箇所を除き，プログラムは指定されたアラインメントでコンパイルされます。指示文は， DEC COBOL コンパイラが解釈する特別な注釈行です (DEC COBOL 指示文はVAX COBOL コンパイラでは無視されます)。すべての指示文は“*DC”で始まります。“*” (アスタリスク) は注釈行の始まりを表わします。

アラインメントに関する指示を変更するために， COBOL ソース・プログラムの中のどこでも以下のアラインメント指示文を使用できます。

- *DC SET ALIGNMENT[=(*option*,...)] (ここで option は[NO]BINARY か[NO]DECIMAL です)

  新しいアラインメントを指定します。*DC SET ALIGNMENT は*DC SET ALIGNMENT=(BINARY,NODECIMAL) と同じです。

- *DC END-SET ALIGNMENT

  アラインメントの設定を前のものに戻します (このアラインメント指示文はオプションです)。

- *DC SET NOALIGNMENT

  バイト・アラインメントを指定します。

特定の数値データに対してアラインメントを切り替えるために，プログラム内でアラインメント指示文を入れ子にすることができます。*DC ENT-SET ALIGNMENT 指示文はオプションですが，入れ子にしたアラインメント指示文の最後には必ずこの文を指定しなければなりません。

A.3.2.2 DEC COBOL の/CHECK=NODECIMAL オプションによる数値データの確認

/CHECK=[NO]DECIMAL オプションは，数値に関する文で表示用数字項目を使用したときに数字であるかどうか検査します。無効な数字や文字に対してシステムがエラーを返すようにしたいときは，/CHECK=DECIMAL を使用してください。

この機能は主に，数値データに異なる内部表現を使用する可能性のある，他のシステムで生成されたデータを検査することを意図しています。また，この修飾子は文字列データから数値データに移送されるプログラムの論理エラーを見つけるために

も用いられます。この機能の欠点は，チェックするのに余計な命令が必要なことで，この結果，実行時間とイメージ・サイズがわずかに大きくなります。

コンパイラに表示用数字項目のなかの数字をチェックさせたくないときは，/CHECK=NODECIMAL(省略時設定) を使用してください。

A.3.2.3 先行する空白文字をゼロに変換する DEC COBOL の/CONVERT=LEADING_BLANKS オプション

/CONVERT=LEADING_BLANKS 修飾子は，表示用数字項目中の先行する空白文字をゼロに置き換え，またゼロであることを検査するコードを生成します。

この機能はおもに，実行時に先行する空白文字をゼロに置き換えることで，COBOL のプログラムを編集することなく移行することを意図しています。この機能の欠点は，データの変換に特別なコードを必要とすることです。この結果，実行時間とイメージ・サイズがわずかに大きくなります。

コンパイラに表示用数字項目中の先行する空白文字をゼロに変換させたくないときは，/NOCONVERT=LEADING_BLANKS (省略時設定) †を使用してください。

A.3.2.4 DEC COBOL の/FLOAT 修飾子に浮動小数点データ型を指定する

/FLOAT=[*option*]修飾子は，メモリ内で単精度および倍精度データに対して使用される浮動小数点データ型を指定します。ひとつのプログラムでは，/FLOAT=D_FLOAT か/FLOAT=IEEE_FLOAT のどちらかひとつを指定してください。

Alpha AXP アーキテクチャは IEEE 規格準拠ですので， DEC COBOL 上で IEEE 浮動小数点データ型を含む既存の COBOL プログラムを実行することができます。

/FLOAT=D_FLOAT (省略時設定) はコンパイル時に，単精度 (COMP-1) データに VAX の F 浮動メモリ型を，倍精度 (COMP-2) データに VAX の D 浮動メモリ型をそれぞれ指定してします。

浮動小数点演算の IEEE 規格である ANSI/IEEE 745-1985 では， 4 つの浮動小数点形式を定義しています。これには基本形式 (basic)，拡張形式 (extended) の 2 つのグループがあり，それぞれのグループ内で単精度型，倍精度型が定められています。 Alpha AXP アーキテクチャでは，このうち基本形式の単精度型，倍精度型をサポートします。

† /NOCONVERT=LEADING_BLANKS でも可。

/FLOAT=S_FLOAT(省略時設定) はコンパイル時に，単精度 (COMP-1) データに IEEE の S 浮動メモリ型を，倍精度 (COMP-2) データに IEEE の T 浮動メモリ型を，それぞれ指定します。

Alpha AXP アーキテクチャでの浮動小数点データ型の使用に関する詳細は，『Alpha Architecture Handbook』を参照してください。

A.3.2.5 /OPTIMIZE 修飾子でコードを最適化する

/OPTIMIZE 修飾子は，コンパイラがコンパイルされるプログラムをより効率的なコードを生成するように最適化するかどうかを制御します。

プログラムをより速く実行させたいときは/OPTIMIZE(省略時設定) を指定します。この修飾子を使用すると，コンパイラはこれを使用しないときより大きいサイズのオブジェクト・モジュールを生成し，コンパイルにかかる時間も長くなります。

機械語コードがソース・プログラムのプログラム行と同じ順序であることを確かめるデバッグ・セッションには，/NOOPTIMIZE を使用してください。

A.3.2.6 特殊な予約語をチェックする/RESERVED_WORDS 修飾子

/RESERVED_WORDS 修飾子はコンパイラが特定の語を予約語として認識するかどうかを制御します。

プログラムが， X/Open ポータビリティ・ガイドで COBOL の語として定義されている語を 1 つでも使用しているときは，/RESERVED_WORDS=NOXOPEN を使用してください。

プログラムに以下の X/Open COBOL の語がひとつもないときは，/RESERVED_WORDS=XOPEN (省略時設定) を使用してください。

AUTO
BACKGROUND-COLOR
BELL
BLINK
EOL
EOS
ERASE

FOREGROUND-COLOR
FULL
HIGHLIGHT
LOWLIGHT
REQUIRED
REVERSE-VIDEO
SCREEN
SECURE
UNDERLINE

A.3.2.7 COBOL ANSI 規格の言語拡張機能を呼び出す/STANDARD 修飾子

/STANDARD 修飾子はコンパイラが特定の言語機能に関する情報メッセージを出力するかどうかを制御します。このメッセージを得るには/STANDARD, /STANDARD=85 (および/WARNINGS=ALL または/WARNINGS=INFORMATIONAL) または/STANDARD=SYNTAX を指定します。

コンパイラに ANSI 1985 COBOL 規格に沿ったコードを生成させるには, /STANDARD=85 (省略時設定) を使用してください。

DEC COBOL コンパイラに, ANSI 1985 COBOL 規格に対する DEC の拡張機能に関する情報メッセージを生成させるには, /STANDARD=SYNTAX を使用してください。 NOSYNTAX (省略時設定) はこれらのメッセージを生成しません。

特殊な例で, DEC COBOL コンパイラにVAX COBOL バージョン 3.4 の形式のコードを生成させる場合は, /STANDARD=V3 を指定してください。第 A.3.2.7.1 項では, /STANDARD=V3 オプションについてさらに詳しく説明しています。

A.3.2.7.1 /STANDARD=V3 オプション

DEC COBOL バージョン 1.0 は, VAX COBOL バージョン 4.0 以降と同じように ANSI 1985 COBOL 規格に基づいています。このため DEC COBOL は, /STANDARD=85 オプションでそれを完全にサポートしています。 DEC COBOL はまた, VAX COBOL バージョン 4.0 以降で有効となった/STANDARD=V3 修飾子の機能もサポートします。

バージョン 4.0 以前VAX COBOL は ANSI 1974 COBOL 規格に基づいていました。バージョン 4.0 以降のバージョン・アップのほとんどがVAX COBOL コンパイラの以前のバージョンと互換性を持つ一方で, いくつかの相違点も存在します。それらが引き起こす結果は, 何種類かに分けられます。

既存のVAX COBOL プログラムとの矛盾を最小限にとどめるため，VAX COBOL ではバージョン 4.0 以降の規則に従ってプログラムをコンパイルすることも，また，バージョン 3.4 の規則にに従ってコンパイルすることもできます。特殊な例で，/STANDARD=V3 を指定するとVAX COBOL バージョン 3.4 の規則に沿ったコードを，VAX COBOL コンパイラに生成させることができます。これは『VAX COBOL User Manual』で説明されています。

DEC COBOL では，VAX COBOL バージョン 4.0 以降で有効な機能と比べ，/STANDARD=V3 オプションに対しては制限付きのサポートを提供しています。/STANDARD=V3 を指定すると，以下の 4 つの場合， DEC COBOL はVAX COBOL バージョン 4.0 以降とまったく同じに動作します。

- メイン・プログラムの EXIT PROGRAM 文
- I-O ファイル状態コード
- 次のレコードに対する記述がない場合
- I-O および EXTEND モードでファイルをオープンするときに，そのファイルが存在しなかった場合

以下の 4 つの項目で，この DEC COBOL の動作を詳しく説明します。

EXIT PROGRAM Statement

/STANDARD=V3 を指定した場合， EXIT PROGRAM 文はメイン・プログラムとサブプログラムの両方でプログラムの終了として扱われます。

/STANDARD=85 を指定した場合，メイン・プログラム本体にある EXIT PROGRAM 文は通過し， EXIT PROGRAM 文の後の文を実行します。そのプログラムがサブプログラムである場合には， EXIT PROGRAM 文はサブプログラムを呼んだプログラムへの戻ることになります。

I-O ファイル状態コード

/STANDARD=V3 を指定すると，表 A–9 の V3 で示された欄の値が I-O ファイル状態コードとして得られます。プログラムはこの値をもとに動作します。

/STANDARD=85 を指定すると，表 A–9 の 85 で示された欄の値が I-O ファイル状態コードとして得られます。プログラムはこの値をもとに動作します。

表 A–9 は，VAX COBOL バージョン 3.4 と DEC COBOL のファイル状態コードを説明します。

表 A–9 /STANDARD 修飾子の I-O ファイル状態コード

I-O エラー状態	ステータス・コード	
	V3	85
READ successful–record shorter than fixed file attribute.	00	04
CLOSE reel/unit attempted on nonreel/unit device.	00	07
READ fails–relative key digits exceed relative key.	00	14
WRITE fails–relative key digits exceed relative key.	00	24
OPEN I-O on file that is not mass storage.	00	37
WRITE fails–attempt to write a record of a different size than in the file description.	00	44
READ fails–no next logical record (EOF detected).	13	10
READ fails–no next logical record (EOF on OPTIONAL file).	15	10
READ fails–no valid next record (already at EOF).	16	10
READ NEXT or sequential READ–no valid next record pointer.	16[1]	46[1]
READ or START fails–optional input file not present.	25	23
READ successful–record longer than fixed file attribute.	30	04
OPEN on relative or indexed file that is not mass storage.	30	37
REWRITE fails–attempt to rewrite record of different size.	30	44
CLOSE fails–file not currently open.	93	42

[1]次のレコードに対する記述がない場合を参照。

(次ページに続く)

表 A–9 (続き) /STANDARD 修飾子の I-O ファイル状態コード

I-O エラー状態	ステータス・コード	
	V3	85
DELETE or REWRITE fails–previous I-O not successful READ.	93	43
OPEN fails–file previously closed with LOCK.	94	38
OPEN fails–file created with different organization.	94	39
OPEN fails–file created with different prime record key.	94	39
OPEN fails–file created with different alternate record keys.	94	39
OPEN fails–file currently open.	94	41
READ or START fails–file not opened INPUT or I-O.	94	47
WRITE fails–file not opened OUTPUT, EXTEND, or I-O.	94	48
DELETE or REWRITE fails–file not opened I-O.	94	49
OPEN INPUT on a nonoptional file–file not found.	97	35

次のレコードに対する記述がない場合

ここでは以下の状態のすべてが満たされた場合に，/STANDARD=V3 または /STANDARD=85 を指定してプログラムのコンパイルを行ったときに起こることを説明します。

- 次のレコードに対する条件句の記述がない
- プログラムがシーケンシャルな READ 文を実行しようとする
- READ 文に関連する AT END 文がある

/STANDARD=V3 を使用してプログラムをコンパイルすると，次の事が起こります。

- そのファイルのファイル状態コード変数がもしあれば， 16 にセットされる
- AT END 文に関連した文は実行される

- プログラムは正常に実行を続ける

/STANDARD=85 を使用してプログラムをコンパイルすると，次の事が起こります。

- そのファイルのファイル状態コード変数がもしあれば， 46 にセットされる
- AT END 文に関連した文は実行されない
- プログラムは異常終了する (USE AFTER STANDARD EXCEPTION プロシージャを実行しないかぎり)

OPEN 文の I-O モードと EXTEND モード
/STANDARD=V3 を指定すると，ファイルが存在しない場合に I-O モードまたは EXTEND モードでオープンされるファイルが作成されます。

/STANDARD=85 を指定すると，ファイルが場合に I-O モードまたは EXTEND モードでオープンされるファイルが作成されません。そのかわり，実行時エラーとなります。

A.3.2.7.2 /STANDARD 修飾子と /WARNINGS 修飾子　VAX COBOL は同じ働きをする 2 つの修飾子，/STANDARD=[NO]SYNTAX と/WARNINGS=[NO]STANDARD. を提供します。

DEC COBOL は/WARNINGS 修飾子の[NO]STANDARD オプションをサポートしません。したがって， DEC COBOL コンパイラで/WARNING=ALL を指定しても DEC の拡張機能に関する情報メッセージは出力されません。次のようなメッセージを出力するには，/STANDARD=SYNTAX を指定しなければなりません。

```
%COBOL-I-EXTENSION
```

注意

VAX COBOL と DEC COBOL において，/FLAGGER[(*option*,...)]を指定したときにコンパイラが出力する DEC の拡張機能に関する FIPS メッセージは，引き続き/WARNINGS=INFORMATION オプションによって制御されます。

A.3.2.8 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呼ぶ /TIE 修飾子

/TIE (Translated Image Environment) 修飾子は，ネイティブな AXP イメージがトランスレートされたイメージを呼ぶこと，および，トランスレートされたイメージがネイティブな AXP イメージを呼ぶことを許可します。この修飾子は， AXP システムでのみサポートされます。

/TIE を指定すると，コンパイルしたコードからトランスレートされたイメージを呼ぶことも，逆にトランスレートされたイメージからそのコードを呼ぶこともできるため，コンパイルしたコードをトランスレートされた共有イメージと共に使用できます。/TIE を指定したときには， LINK コマンドの/NONATIVE_ONLY 修飾子を使ってオブジェクト・モジュールをリンクしなければなりません (/NONATIVE_ONLY 修飾子に関して詳しくは，『OpenVMS Linker Utility Manual』を参照してください)。

/NOTIE(省略時設定) を指定すると，コンパイルされたコードはトランスレートされたイメージと関連を持ちません。

相互操作性について詳しくは，第 6 章を参照してください。トランスレートされたイメージについては，『DECmigrate for OpenVMS AXP Systems Translating Images』を参照してください。

A.3.2.9 VAX COBOL から DEC COBOL へのプログラムの変換

VAX COBOL バージョン 5.1 は，/STANDARD=OPNVMS_AXP オプションにより， AXP の DEC COBOL では有効でない，VAX COBOL プログラムの言語機能を判定する新しいフラグ・システムを提供します。

/STANDARD=OPENVMS_AXP (および/WARNINGS=ALL または/WARNINGS=INFORMATIONAL) を指定すると，VAX COBOL コンパイラは DEC COBOL では有効でない言語機能にフラグを立てる情報メッセージを生成します。 DEC COBOL でコンパイルする前にプログラムを変更するために，このメッセージを使用することができます。

情報メッセージを出さないときは，/STANDARD=NOOPENVMS_AXP (省略時設定) を使用してください。

A.3.2.10 プログラムの構造

いくつかの場合，DEC COBOL コンパイラは到達不能なコードまたは論理エラーについて，VAX COBOL コンパイラよりも完全なメッセージを生成します。

次の例はサンプル・プログラムと DEC COBOL コンパイラが出力するメッセージを示しています。

ソース・ファイル:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. T1.
ENVIRONMENT DIVISION.
PROCEDURE DIVISION.
P0.
    GO TO P1.
P3.
    GO TO P2.
P2.
    DISPLAY "This is unreachable code".
P1.
    STOP RUN.
IDENTIFICATION DIVISION.
PROGRAM-ID. T2.
ENVIRONMENT DIVISION.
PROCEDURE DIVISION.
P0.
    DISPLAY "This is unreachable code".
    EXIT PROGRAM.
END PROGRAM T2.
END PROGRAM T1.
```

VAX **システムの場合**:

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

AXP システムの場合:

```
$ COBOL/ANSI/OPT/WARNINGS=ALL T1.COB
        PROGRAM-ID. T2.
.................^
%COBOL-I-UNCALLED, routine T2 can never be called
at line number 14 in file DISK$YOURDISK:[TESTDIR]T1.COB;1
        P2.
........^
%COBOL-I-UNREACH, code can never be executed at label P2
at line number 9 in file DISK$YOURDISK:[TESTDIR]T1.COB;1
```

同じプログラムに対し，VAX COBOL コンパイラは到達不能なラベルと到達不能なサブプログラムの両方を検出しても，メッセージを出しません。

DEC COBOL コンパイラに呼ばれていないルーチンの分析をさせるには，/OPTIMIZE 修飾子を使用してください。コンパイラは，省略時の (簡単な) レベルの最適化をするために到達不能なコードの分析を行います。

このVAX COBOL との違いは，プログラムをデバッグするときに役立ちます。これらのメッセージは情報メッセージなので，コンパイラはオブジェクト・ファイルを生成し，それをリンクおよび実行することができます。しかし，これらのメッセージは検出されていない論理エラーを示すこともあります (その場合，プログラムはおそらく期待した動作をしません)。

A.3.2.11 COPY 文と REPLACE 文

DEC COBOL コンパイラは COBOL プログラムの COPY 文に対する注釈を表示するときに，異なった出力を生成します。

次の 2 つの例は，VAX COBOL コンパイラと DEC COBOL コンパイラでは， L という文字の位置が異なるように，注釈の表示の位置に違いがあることを示しています。

VAX COBOL ソース・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.
```

DEC COBOL ソース・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.
```

DEC COBOL コンパイラはまた，次の例に示すように複数の COPY 文を 1 行にした COBOL プログラムの表示では，異なった出力を生成します。コンパイラが置き換えられた行にメッセージを出すとき，メッセージ・ポインタは置きかわったテキストではなく，もとのテキストを呼び出します。

VAX COBOL ソース・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.
```

DEC COBOL ソース・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.
```

COBOL ソースの REPLACE 文と DATECOMPILED 文の診断を行うと，コンパイラはソース・プログラムの一部の行を複数表示します。

DEC COBOL プログラムの REPLACE 文では，置きかわったテキストにコンパイラがメッセージを出したときは，コンパイラ・メッセージがプログラム中のもとのテキストに対応します。VAX COBOL プログラムでは，しかし，コンパイラ・メッセージは置きかわったテキストに対応します。

以下の 2 つの例で示すように， COPY 文が行の中央にテキストを挿入すると，DEC COBOL プログラムとVAX COBOL プログラムに対するコンパイラの表示は異なります。

DEC COBOL **ソース・ファイル**:

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

VAX COBOL **ソース・ファイル**:

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

LCOP5D.LIB は "O"というテキストを含んでいます。 DEC COBOL コンパイラはその行をそのままにして， COPY ファイルの内容をソース行の後に挿入します。VAX COBOL コンパイラはソース行を 2 行に分けます。

REPLACE 文と COPY REPLACING 文に対しては，プログラムが表示する行番号が， DEC COBOL とVAX COBOL で異なります。 DEC COBOL では，置きかわった行番号がもとのソース・ファイルの行番号と一致しますが，その次の行番号は異なってしまいます。VAX COBOL コンパイラは行番号を連続して変更します。

次のソース・プログラムは， DEC COBOL コンパイラとVAX COBOL コンパイラのどちらでコンパイルするかによって，最終行の番号が異なった状態で表示されます。

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 **バージョン**:

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

VAX COBOL **バージョン**:

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

A.3.2.12 MOVE 文

符号なしの COMPUTIONAL (COMP) ファイルは符号つき COMPTIONAL (COMP) ファイルよりも大きな値を持つことができます。 ANSI COBOL 規格に従って，符号なし項目の値は正の値として扱われることになっています。しかし， DEC COBOL が符号なし項目を正の値として扱うのに対し，VAX COBOL は符号なし項目を符号つき項目として扱います。したがっていくつかの稀な場合，MOVE 文や演算文の中の符号なしと符号付きのデータの混在は，VAX COBOL と DEC COBOL で異なった結果をもたらします。

次のサンプル・プログラムはVAX COBOL と DEC COBOL で結果が異なります。

ソース・ファイル:

```
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.
```

VAX COBOL **の結果**:

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

DEC COBOL **の結果**:

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

A.3.2.13 ACCEPT 文と DISPLAY 文

プログラムで ACCEPT か DISPLAY の拡張機能を使用する場合， DEC COBOL コンパイラは DEC SMG(Screen Manager) を使用します。 DEC COBOL とVAX COBOL の動作の違いの目に見える部分は以下のとおりです。

- プログラムを実行する場合，最初の ACCEPT 文あるいは DISPLAY 文を見つけた時点で画面は自動的に消去されます。
- DEC SMG は，行単位の入出力としてより画面全体として．拡張された ACCEPT 文と DISPALY 文の端末入出力を管理するので， DECterm のスクロール・バーでスクロールして画面を再表示することはできない場合があります。

- DCL コマンドの RECALL は画面が使用されている間はサポートされません。
- エスケープ・シーケンスの処理は DISPLY 文字列の左端に位置するエスケープ・シーケンスの使用に限られます (『DEC COBOL User Manual』にサンプル・プログラムがあります)。
- ANSI ACCEPT 文と拡張された ACCEPT 文がひとつのプログラムにある場合，拡張された ACCEPT 文に使用される編集キーは， ANSI ACCEPT 文でも使用されます (編集キーの完全なリストは『DEC COBOL User Manual』を参照してください)。

A.3.2.14 LINAGE 文

DEC COBOL コンパイラとVAX COBOL コンパイラは， LINAGE 文で大きな値を扱ったときに，動作の違いを表わします。 WRITE 文の ADVANCING 句の行カウントが 127 より大きいと， DEC COBOL は 1 行進みますが，VAX COBOL の結果は定義されていません。

A.3.2.15 ファイル状態の違い

EXTEND モードでファイルをオープンし，それを REWRITE しようとすると，DEC COBOL コンパイラとVAX COBOL コンパイラは異なったファイル状態コードを報告します。 DEC COBOL は 49(互換性のないオープン・モード) と報告し，VAX COBOL はエラーの 43(対応する READ 文がない) と報告します。

DEC COBOL は START が失敗した後でファイル状態コードを 46 にセットします。VAX COBOL はこのような結果を生成しません。

A.3.2.16 システム・サービス・コールからの戻り値

次の例は VAX システムでは正しく動作してしまうが， AXP システムでは同じ動作にならない，規則に反するコーディングです。この動作の違いは VAX アーキテクチャと Alpha AXP アーキテクチャ間の，レジスタ・セットにおけるアーキテクチャ的な違いを示しています。つまり， AXP システムでの動作の違いは，浮動小数点データ型に使用されるレジスタ・セットの使用法の違いによるものです。

```
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.
    EXIT PROGRAM.
```

サンプル・プログラムの中では，プログラマが，バイナリ (COMP) であるべきシステム・サービスー・コールの戻り値を， F 浮動小数点と間違って定義しています。プログラマは次のような VAX のやり方に従っているのです。 VAX アーキテクチャでは，ルーチンからのすべての戻り値が R0 レジスタに返されます。 VAX アーキテクチャには整数と浮動小数点に用途を特定したレジスタはありません。 Alpha AXP アーキテクチャは浮動小数点データとバイナリ・データに別々のレジスタ・セットを定義します。特に，浮動小数点の値を返すルーチンはそれらを F0 レジスタに返し，バイナリの値を返すルーチンはそれらを R0 レジスタに返します。

DEC COBOL コンパイラは外部のルーチンが返したデータのデータ・タイプを決める方法を持っていません。 CALL 文の GIVING-VALUE 項目には正しいデータ・タイプを指定しなければなりません。 AXP システムでは，浮動小数点データ項目に使われるレジスタ・セットが異なるので，生成されたコードは R0 のかわりに F0 をテストしています。

サンプル・プログラムでは， F0 の値がコード・シーケンスのなかでは完全に不定値です。このコーディングの例は期待する動作を生成する場合もありますが，ほとんどの場合そうはなりません。

A.3.2.17 倍精度データ項目における記憶領域の違い

VAX アーキテクチャと Alpha AXP アーキテクチャでの D 浮動小数点データの記憶領域の違いにより，実行結果の評価の際，わずかに異なった答えを生成します。この差は最終的な結果を出力するまで何回 D_float の演算を行うかによって大きくなります。 COMP-2 型のデータを出力する場合， OpenVMS AXP で動作するプログラムの出力が OpenVMS VAX でも同様に出力されると仮定したプログラムを作成した場合，大きな障害になる可能性があります。

浮動小数点データ型の記憶領域については，『Alpha Architecture Handbook』を参照してください。

A.3.2.18 RMS スペシャル・レジスタ

DEC COBOL ランタイム・システムは， RMS オペレーションを試みる前に入出力エラー状態をチェックします。VAX COBOL はチェックなしで RMS 呼び出しを実行し，その結果 RMS スペシャル・レジスタが異なる値となります。 DEC COBOL ランタイム・システムが RMS オペレーションを試みない場合は，レジスタの値は以前のままになります。

たとえば，ファイルのオープンが失敗した場合，どんな DEC COBOL レコード操作 (READ， WRITE， START， DELETE， REWRITE, あるいは UNLOCK) も RMS の起動前に失敗します。

A.4 DEC Fortran for OpenVMS AXP とVAX FORTRANとの互換性

この節では， DEC Fortran for OpenVMS AXP システムと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 A.4.1 項)
- コマンド・ライン修飾子 (第 A.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 A.4.3 項)
- VAX FORTRANデータの移植 (第 A.4.4 項)

A.4.1 言語機能

DEC Fortran には， ANSI FORTRAN–77標準機能をはじめ，FORTRAN–77標準機能に対するVAX FORTRANの拡張機能も含まれています。たとえば，次の機能がサポートされます。

- RECORD 文と STRUCTURE 文
- CDEC$ディレクティブと 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 I/O
- 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
- 『DEC Fortran Language Reference Manual』に示す他の言語要素

拡張機能についての詳しい説明は『DEC Fortran Language Reference Manual』を参照してください。このマニュアルには，FORTRAN–77標準規格の拡張機能がわかりやすく示されています。

この節のこの後の部分では，VAX FORTRANおよび DEC Fortran 固有の言語機能，各言語で共有されるものの，異なる方法で解釈される言語機能，VAX FORTRANには適用されない DEC Fortran の制約事項，およびデータの移植に関する検討事項をまとめます。

言語機能についての詳しい説明は『DEC Fortran Language Reference Manual』を参照してください。

A.4.1.1 DEC Fortran 固有の言語機能

次の言語機能は， DEC Fortran では提供されますが，VAX FORTRANバージョン 5.0 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。
- AUTOMATIC 文と STATIC 文
- 再帰

- 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
- POINTER 文データ型
- 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。 AXP 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，VAX FORTRAN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。
- ′0..1′B と B′0..1′という形式のビット定数
- 8 進定数 (O′0..7′と 16 進定数 (X′0..F′または Z′0..F′) に対する MIL-STD 1753 構文
- 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 DEC Fortran 固有のエラーです。
- プラットフォーム固有の機能に対して与えられる警告メッセージ“feature not available on this platform”は， OpenVMS AXP ではサポートされません。
- 大文字と小文字を区別する名前

- DEC Fortran では，入出力ユニット番号は 0 または正の整数として指定できます。VAX FORTRANでは，入出力ユニット番号の値は 0 ～ 99 の範囲です。

DEC Fortran 言語機能についての説明は『DEC Fortran Language Reference Manual』を参照してください。

A.4.1.2 VAX FORTRAN固有の言語機能

次の言語機能はVAX FORTRANでは使用できますが， DEC Fortran ではサポートされません。

- FORTRAN/PARALLEL=(AUTOMATIC)
- CPAR$の使用による FORTRAN/PARALLEL=(MANUAL)
  (たとえば， CPAR$ DO_PARALLEL など)
- DICTIONARY 文 (Common Data Dictionary は DEC Fortran の最初のリリースではサポートされません)
- 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 (DEC Fortran 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。DEC Fortran 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。

次の言語機能はVAX FORTRANでは使用できますが， Alpha AXP アーキテクチャと VAX アーキテクチャとの違いにより， DEC Fortran ではサポートされません。

- いくつかの FORSYSDEF シンボル定義モジュールは VAX アーキテクチャまたは AXP 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 AXP システムとで異なります。
- REAL*16 (H 浮動小数点) データ型と REAL*16 Q 組み込み関数
- D 浮動小数点に対する VAX のサポート

  AXP 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 AXP 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。

  AXP 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 DEC Fortran 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

  VAX FORTRANの High-Performance Option (HPO) に関連するベクタ化は，/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。AXP 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。

A.4.1.3 解釈方法の相違

次の言語機能は，VAX FORTRANと DEC Fortran とで異なる方法で解釈されます。

- 整数定数の 8 進表現
- 乱数ジェネレータ (RAN)

DEC Fortran の RAN 関数は，同じランダム・シードに対してVAX FORTRAN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VAX FORTRAN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，VAX FORTRANと DEC Fortran は異なる動作をします。

  – 2 つの異なる入出力文がフォーマット指定子として同じ CHARACTER PARAMETER 定数を参照する場合。次の例を参照してください。

```
CHARACTER*(*) FMT2
PARAMETER (FMT2='(10Habcdefghij)')
READ (5, FMT2)
WRITE (6, FMT2)
```

  – 2 つの異なる入出力文がそれぞれのフォーマット指定子として同じ文字定数を使用する場合。次の例を参照してください。

```
READ (5, '(10Habcdefghij)')
WRITE (6, '(10Habcdefghij)')
```

  VAX FORTRAN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)， DEC Fortran では， WRITE 文の出力は "abcdefghij"になります (つまり， READ 文によって読み込まれた値は WRITE 文によって書き込まれる値に影響を与えません)。

A.4.1.4 DEC Fortran の制約事項

あるVAX FORTRANの機能は， DEC Fortran では使い方が制限されていたり，またはまったく使用できません。

- 数値ローカル変数は，常にではありませんが，使用している最適化のレベルに応じて，値が 0 に初期化されることがあります。どのような場合も，必ず値を 0 に初期化するようにするには，明示的な代入文を使用するか，または DATA 文を使用してください。
- 文字定数には文字ダミー引数を割り当てなければならず，数値ダミー引数を割り当てることはできません (VAX FORTRANでは，ダミー引数が数値の場合，'A'は参照によって渡されていました)。

- 保存されたダミー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
```

- ホレリス実引数は文字ダミー引数ではなく，数値ダミー (仮) 引数に対応づけなければなりません。

A.4.2 コマンド・ライン修飾子

この節では， DEC Fortran とVAX FORTRANのコマンド・ライン修飾子の違いについてまとめます。

DEC Fortran 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。VAX FORTRAN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 AXP 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。

一部のコンパイラ修飾子は各言語固有の修飾子ですが， DEC Fortran とVAX FORTRANは多くの修飾子を共有しています。

A.4.2.1 共有される修飾子

表 A–10 は， DEC Fortran とVAX FORTRANで共有されるコンパイラ修飾子を示しています。 DEC Fortran の修飾子についての詳しい説明は『DEC Fortran Language Reference Manual』を参照してください。

表 A–10 DEC Fortran とVAX FORTRANで共有される修飾子

修飾子	説明
/ASSUME	/ASSUME はVAX FORTRANバージョン 5.0 では使用できなかったが，現在は/ASSUME=([NO]ACCURACY_SENSITIVE,[NO]DUMMY_ALIAS) をVAX FORTRAN HPO で使用できる。 特定のキーワード値は DEC Fortran 固有の値である。
/ANALYSIS_DATA	同じ。
/CHECK	VAX FORTRANの/CHECK=キーワードはすべて， DEC Fortran でも使用できる。 DEC Fortran の/WARNINGS=ALIGNMENT 修飾子と VAX FORTRAN HPO の/CHECK=ALIGNMENT 修飾子は同じである。
/CROSS_REFERENCE	同じ。
/DEBUG	VAX FORTRANの/DEBUG=キーワードはすべて， DEC Fortran でも使用できる。
/D_LINES	同じ。
/DIAGNOSTICS	同じ。
/DML	同じ。
/EXTEND_SOURCE	同じ。
/F77	同じ。
/G_FLOATING	DEC Fortran では/G_FLOATING をサポートするが， DEC Fortran の/FLOAT 修飾子を使用しなければならない。第 A.4.2.2 項の/FLOAT の説明で述べるように， VAX システムと AXP システムでは D 浮動小数点の演算に違いがある。
/I4	同じ。 DEC Fortran では， INTEGER 宣言のサイズを指定するために/INTEGER_SIZE 修飾子を使用できる。
/LIBRARY	同じ。
/LIST	同じ。
/MACHINE_CODE	同じ。
/OBJECT	同じ。
/OPTIMIZE	同じであるが， DEC Fortran では，ローカルな最適化のみに対して/OPTIMIZE=LEVEL=1 などの最適化レベルの使用をサポートする (第 A.4.2.2 項を参照)。実際のコンパイラ最適化技法は異なる可能性がある。

(次ページに続く)

表 A–10 (続き) DEC Fortran とVAX FORTRANで共有される修飾子

修飾子	説明
/SHOW	VAX FORTRANの大部分の/SHOW=キーワードは DEC Fortran でも使用できる。
/STANDARD	VAX FORTRANの/STANDARD=キーワードはすべて， DEC Fortran でも使用できる。
/WARNINGS	大部分の/WARNINGS=キーワードは使用できる。しかし，特定のキーワード値は DEC Fortran 固有の値である。

VAX FORTRANソース・プログラムを移植する場合には， DEC Fortran の/VMS 修飾子 (省略時の設定) を使用することを考慮してください。

A.4.2.2 DEC Fortran 固有の修飾子

表 A–11 は， DEC Fortran コンパイラの修飾子のうち，対応するVAX FORTRANオプションがなく，VAX FORTRANバージョン 5.0 でサポートされない修飾子を示しています。

表 A–11 VAX FORTRANでサポートされない DEC Fortran 修飾子

修飾子	説明
/ALIGN	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 (VAX FORTRANでは，コモン・ブロックに対して省略時のアラインメントを使用し， CDEC$ディレクティブも認識する)。
/ASSUME	特定のキーワードはVAX FORTRANバージョン 5.0 では使用できない。たとえば， BIG_ENDIAN, CRAY， IBM， LITTLE_ENDIAN， RECURSIVE, VAXD，および VAXG はサポートされない。
/FLOAT	浮動小数点データに対して使用するフォーマット (REAL または COMPLEX) を制御し， VAX G 浮動小数点， VAX D 浮動小数点，または IEEE (S 浮動小数点と T 浮動小数点) 浮動小数点データの使用を許可する。
/INTEGER_SIZE	INTEGER 宣言のサイズを制御する。

(次ページに続く)

表 A–11 (続き) VAX FORTRANでサポートされない DEC Fortran 修飾子

修飾子	説明
/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，元のまま保存するのかを制御する。
/OPTIMIZE=LEVEL= n	最適化のレベルを /NOOPTIMIZE (/OPTIMIZE=LEVEL=0) と /OPTIMIZE (/OPTIMIZE=LEVEL=4) の間で制御する。VAX FORTRAN(および DEC Fortran) は /OPTIMIZE と/NOOPTIMIZE をサポートする。
/POINTER_SIZE	ポインタ・データのサイズ (アドレッシング可能な範囲) を制御する。
/RECURSIVE	実行時プロセス・スタックでローカル・データを割り当て，可能な再帰的実行のためにプロシージャをプリページする。
/VMS	DEC Fortran が特定のVAX FORTRAN表記法を使用することを要求する。
/WARNING=(ALIGNMENTS, TRUNCATED_SOURCE)	自然にアラインされていないデータと切り捨てられたソース行に対して警告メッセージを表示することを要求する。これらのキーワードはVAX FORTRAN HPO で使用できる。

A.4.2.3 VAX FORTRAN固有の修飾子

この節では，VAX FORTRANコンパイラのオプションのうち， DEC Fortran には対応するオプションがないものをまとめます。

表 A–12 は，VAX FORTRANバージョン 5.0 固有のコンパイル・オプションを示しています。

表 A–12 DEC Fortran でサポートされないVAX FORTRANオプション

VAX FORTRAN修飾子	説明
/BLAS=(INLINE,MAPPED)	VAX FORTRAN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。VAX FORTRAN High Performance Option (HPO) の場合にのみ使用できる。

(次ページに続く)

表 A–12 (続き) DEC Fortran でサポートされないVAX FORTRANオプション

VAX FORTRAN修飾子	説明
/CHECK=ASSERTIONS	アサーション・チェックを許可または禁止する。VAX FORTRAN HPO に対してのみ使用できる。
/CONTINUATIONS=n	1 つの文で認められる継続行の行数を指定する。 DEC Fortran では継続行の行数は最大 99 行である。
/DESIGN=[NO]COMMENTS /DESIGN=[NO]PLACEHOLDERS	設計情報を確認するためにプログラムを解析する。
/DIRECTIVES=DEPENDENCE	指定されたコンパイラ・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。VAX FORTRAN HPO に対してのみ使用できる。
/MATH_LIBRARY=(FAST または ACCURATE)	特定の算術演算組み込み関数を実現するために使用される算術演算ライブラリ・ルーチンの選択を制御する。/MATH_LIBRARY はまた， real データ型の指数演算子**に対するベクタ化された参照にも影響を与える。VAX FORTRAN HPO でのみ使用できる。
/PARALLEL=(MANUAL または AUTOMATIC)	並列処理をサポートする。
/SHOW=(DATA_DEPENDENCIES,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VAX FORTRAN HPO に対してのみ使用できる。
/VECTOR	ベクタ処理を要求する。VAX FORTRAN HPO 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。VAX FORTRAN HPO 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VAX FORTRAN固有です。『DEC Fortran Language Reference Manual』を参照してください。

VAX FORTRAN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。

A.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

DEC Fortran 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・ラインに/TIE 修飾子を指定します。
- LINK コマンド・ラインに/NONATIVE_ONLY 修飾子を指定します (省略時の設定)。

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，VAX FORTRAN RTL (FORRTL) が DEC Fortran RTL (DEC$FORTRTL) と協調動作できるようにするためのコードが含まれます。ネイティブなプログラム (DEC Fortran RTL) とトランスレートされたプログラム (VAX FORTRAN 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。

A.4.4 VAX FORTRANデータの移植

レコード・タイプは，VAX FORTRANでも DEC Fortran でも同じです。必要な場合には， EXCHANGE コマンドと/NETWORK 修飾子および /TRANSFER=BLOCK 修飾子を使用してデータを移植してください。コピー操作でファイルを Stream_LF 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，VAX FORTRANプログラム (VAX ハードウェァ) 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。 DEC Fortran プログラム (AXP 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 A–13 に示すフォーマットのいずれかで格納します。

表 A–13 VAX システムと AXPVMS 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	AXP 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	DEC Fortran 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]AXP 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， DEC Fortran 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。

VAX D 浮動小数点または G 浮動小数点 (REAL*8) の範囲外の VAX H 浮動小数点 (REAL*16) データを変換することはできません。このような場合には， 1つ

の方法として，ファイルを VAX D 浮動小数点または G 浮動小数点フォーマット (REAL*8) に変換するための変換アプリケーションを作成し，その後， DEC Fortran の変換ルーチンを使用して，データを IEEE T 浮動小数点フォーマットに変換する方法があります。

A.5 DEC Pascal for OpenVMS AXP 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 AXP 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。

A.5.1 DEC Pascal の新機能

表 A–14 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 A–14 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 A–14 (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。 AXP 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。

A.5.2 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。

A.5.3 あらかじめ宣言されている名前の使用法

互換性を維持するために DEC Pascal は，表 A–15 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， DEC 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 A–15 あらかじめ宣言されている名前の使用法

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

(次ページに続く)

表 A–15 (続き) あらかじめ宣言されている名前の使用法

識別子	望ましい使用法
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

A.5.4 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 AXP 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

A.5.5 古い機能

この節では，サポートされてはいるが使用が勧められない機能について説明します。これらは， DEC の他の Pascal コンパイラとの互換性のためにだけ提供されます。

A.5.5.1 /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。

A.5.5.2 /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 VAX システムでも AXP 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， AXP 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。

A.5.5.3 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。

索引

A

__ADD_ATOMIC_LONG 組み込み機能 A–10
__ADD_ATOMIC_QUAD 組み込み機能 A–10
$ADJWSL 2–4
/ALIGNMENT 修飾子 A–19
Alpha AXP 命令
  DEC C からのアクセス A–9
ARCH_NAME 1–9
ARCH_TYPE 1–8, 1–9

B

BASE 1–7
/BPAGE
  VAX イメージのリンク 6–6

C

CALENDAR パッケージ A–5
COMPONENT_ALIGNMENT A–3
COMPONETN_SIZE A–3
Conditional compilation directives
  DEC C のVAX Cとの相違点 A–12
CPU 1–9
$CREPRC 2–4
$CRETVA 2–4
  メモリの再割り当て 2–15
  例 2–15
$CRMPSC 2–2, 2–4
  マッピング
    拡張された仮想アドレス空間へ 2–18
      例 2–20
$CRMPSC
  マッピング (続き)
    単一ページの 2–22
    定義されたアドレス範囲へ 2–22
      例 2–25

D

DEC Ada
  AXP と VAX システム間の互換性 . . . A–1
DEC C
  64-bit 互換性 A–7
  Alpha AXP 命令へのアクセス A–9
  ANSI 準拠 A–6
  AXP システム固有の機能 A–8
  pcc モード A–6
  VAX Cモード A–6, A–12
  VAX Cとの互換性 A–6
  VAX 命令へのアクセス A–9
  サポートされるデータ型サイズ A–7
  データ・アラインメントの制御 A–11
  データ型サイズ移植用マクロ A–7
  不可分な組み込み機能 A–10
  浮動小数点形式の指定 A–8
DEC COBOL
  VAX COBOL との互換性 A–13
  コマンド・ライン修飾子 A–14
DECmigrate for OpenVMS AXP xiii
DECmigrate ユーティリティ
  VEST コマンド/PRESERVE 修飾子 3–17
DEC Fortran
  VAX FORTRAN
    アーキテクチャの違い A–43
    インタプリタの違い A–44
    共有される修飾子 A–46

DEC Fortran
VAX FORTRAN (続き)
言語の特徴 A–40
コマンド・ライン修飾子 A–46
固有の修飾子 A–49
制約事項 A–45
で使用できない修飾子 A–48
データの移植 A–52
との互換性 A–40
との違い A–40
組み込み名
接頭辞 A–51
相互操作性の考慮 A–51
入出力の実行 A–51
浮動小数点データ型のサポート A–52
DEC Pascal
VAX Pascalとの互換性 A–53
新機能 A–53
$DELTVA 2–6
割り当てたメモリの解除 2–16
/DEMAND_ZERO 1–6
DPML (Digital Portable Mathematics Library)
互換性 1–7
ルーチン 1–7
DZRO_MIN 1–7

E

$EXPREG 2–6, 2–12
例 2–14

F

/FLOAT A–21
浮動小数点形式の指定 A–8
FLOAT_REPRESENTATION A–3
free ルーチン
メモリの割り当て 2–2

G

$GETJPI 2–6
$GETQUI 2–7
$GETSYI 1–8, 1–9, 2–7, 2–31
システム・ページ・サイズの確認 . . . 2–35
$GETUAI 2–7
/GST 1–6

H

HW_MODEL 1–9

I

IEEE_FLOAT A–3
IEEE 浮動小数点
DEC C での指定 A–8
/INFORMATIONALS 1–6
INSQUEx 命令 (VAX)
DEC C からのアクセス A–9
ISD_MAX 1–7

L

$LCKPAG 2–7
LIB$DEC_OVER 5–15
LIB$DECODE_FAULT 5–15
LIB$ESTABLISH 5–16
AXP システムでのサポート 5–17
LIB$FIXUP_FLT 5–15
LIB$FLT_UNDER 5–15
LIB$FREE_VM_PAGE 2–10
LIB$GET_VM_PAGE 2–10
LIB$INT_OVER 5–15
LIB$MATCH_COND 5–8, 5–16
LIB$REVERT 5–16
LIB$SIG_TO_RET 5–16
LIB$SIG_TO_STOP 5–16
LIB$SIGNAL 5–16
LIB$SIM_TRAP 5–16
LIB$STOP 5–16
$LKWSET 2–7, 2–36
Load locked 命令 (LDxL) 3–4

LONG_FLOAT A–3
LONG_LONG_FLOAT_MATH_LIB . . . A–5
LONG_LONG_FLOAT_TEXT_IO A–5

M

MAIN_STORAGE A–3
malloc ルーチン
メモリの割り当て 2–2
MATH_LIB パッケージ A–5
MB 命令
DEC C からのアクセス A–9
/MEMBER_ALIGNMENT
データ・アラインメントの制御 A–11
$MGBLSC 2–8
MTH$ルーチン
互換性 1–7

N

/NATIVE_ONLY 1–6, 6–7
相互操作性 6–3
ネイティブ・イメージの作成 A–51
/NOINFORMATIONALS 1–6

O

OpenVMS エグゼクティブ 1–5
/OPTIMIZE A–22
__OR_ATOMIC_LONG 組み込み機能 A–10
__OR_ATOMIC_QUAD 組み込み機能 A–10
OTS$CALL_PROC RTL
コールバックを可能にする 6–2

P

PAGE_COUNT 2–14
pcc
DEC C 互換モード A–6
/PRESERVE 5–12
PSB の作成 6–2
$PURGWS 2–8

R

REMQUEx 命令 (VAX) A–9
/REPLACE 1–6
/RESERVED_WORD A–22
retadr 引数
$CRETVA 2–15
$CRMPSC 2–19
$EXPREG 2–13

S

/SECTION_BINDING 1–6
$SETPRT 2–8
$SETUAI 2–8
SHARE_GENERIC A–3
SHARED A–3
SIF (シンボル・インフォメーション・ファイル)
形式 6–12
使用方法 6–10
/SIF 6–12
$SNDJBC 2–9
SS$_ALIGN 5–9
シグナル・アレイ 5–13
SS$_HPARITH 5–9
シグナル・アレイ 5–11
/STANDARD A–23, A–27
DEC C 互換モード A–6
STORAGE_UNIT A–3
Store conditional 命令 (STxC) 3–4
SYMBOL_TABLE 1–7
SYMBOL_VECTOR 1–4, 1–7
相互操作性の考慮 6–11
SYS$UNWIND 5–6
SYS.STB 1–5
/SYSEXE 1–5, 1–6
SYSTEM.ADD_INTERRLOCKED . . . A–5
SYSTEM.ALIGNED_WORD A–4
SYSTEM.CLEAR_INTERLOCKED . . . A–4
SYSTEM.H_FLOAT A–4
SYSTEM.IEEE_DOUBLE_FLOAT . . . A–4
SYSTEM.IEEE_SINGLE_FLOAT A–4

SYSTEM.INSQ_STATUS A–5
SYSTEM.INSQHI A–5
SYSTEM.INSQTI A–5
SYSTEM.MAX_DIGITS A–4
SYSTEM.MFPR A–4
SYSTEM.MTPR A–4
SYSTEM.NAME A–4
SYSTEM.READ_REGISTER A–4
SYSTEM.REMQ_STATUS A–5
SYSTEM.REMQHI A–5
SYSTEM.REMQTI A–5
SYSTEM.SET_INTERLOCKED A–4
SYSTEM.SYSTEM_NAME A–4
SYSTEM.TICK A–4
SYSTEM.WRITE_REGISTER A–4
SYSTEM_RUNTIME_TUNING パッケージ A–5

T

TESTBITCCI 命令 A–11
TESTBITSSI 命令 A–11
/TIE
  DEC Fortran がサポートする A–51
  コンパイラ相互操作性修飾子 6–2
TIME_SLICE A–3
TRAPB 命令 A–9

U

$ULKPAG 2–9
$ULWSET 2–9
UNIVERSAL 1–7
$UPDSEC 2–9

V

VAX C
  DEC C 互換モード A–6
VAX_FLOAT A–3
VAX 命令
  DEC C からのアクセス A–9
  インターロック命令
    DEC C でのサポート A–11
VAX FORTRAN
  DEC Fortran との互換性 A–40
VAX FORTRAN (続き)
  データの移植 A–52
VEST (VAX Environment Software Translator) xiii
  修飾子
    /PRESERVE 3–17, 5–12
  使用方法 6–6
  シンボル・インフォメーション・ファイル (SIF) の使用 6–10
  代用イメージの作成 6–14
  相互操作性 6–1
VMS Mathematics (MTH$) ランタイム・ライブラリ
  互換性 1–7
Volatile 属性 A–11
  共有データ
    移植 3–14
    保護 3–5

W

/WARNINGS A–27

ア

アイテム・コード
  ARCH_TYPE 1–8
  $GETSYI 1–9
アプリケーション
  VAX に依存する部分のチェック・リスト 1–2

イ

移行プロセスの概要 1–1
一般的な条件処理サポート・ルーチン
  LIB$DECODE_FAULT 5–15
  LIB$ESTABLISH 5–16
  LIB$MATCH_COND 5–16
  LIB$REVERT 5–16
  LIB$SIG_TO_RET 5–16
  LIB$SIG_TO_STOP 5–16
  LIB$SIGNAL 5–16
  LIB$SIM_TRAP 5–16
  LIB$STOP 5–16

イメージの作成 1–4
インターロック命令 A–11

オ

オーバーフローの通知 5–16
オプションの引数
　シグナル・アレイ 5–3

カ

仮想アドレスのレイアウト 2–11
仮想メモリの割り当て 2–1

キ

共有可能イメージ (shareable image)
　トランスレートされたイメージとネイティブ・イメージの置換 6–9
共有データ (shared data)
　無意識に共有される 3–13
キーワード
　ARCH_NAME 1–9
　ARCH_TYPE 1–9
　CPU 1–9
　HW_MODEL 1–9

ク

グローバル・シンボル・テーブル (GST) 1–6

コ

コンパイラ
　AXP システム 1–2
　互換性 . . . . . A–1〜A–58
　相互操作性修飾子
　　/TIE 6–2

サ

参考文献 . . . . . xii〜xiii
算術演算
　ライブラリ
　　互換性 1–7
　例外 5–10
　　サポート・ルーチン
　　LIB$DEC_OVER 5–15
　　LIB$FIXUP_FLT 5–15
　　LIB$FLT_UNDER 5–15
　　LIB$INT_OVER 5–15

シ

シグナル・アレイ
　SS$_ALIGN 例外 5–13
　SS$_HPARITH 例外 5–11
　アドレス
　　メカニズム・アレイ 5–5
　オプションの引数 5–3
　形式 5–2
　状態値 5–3
　引数の数 5–3
　プログラム・カウンタ (PC) 5–3
　プロセッサ・ステータス・ロングワード (PSL) 5–3
実行スレッド
　同期に関する影響 3–1
システム・サービス
　$ADJWSL 2–4
　$LCKPAG 2–7
　$CREPRC 2–4
　$CRETVA 2–4
　$CRMPSC 2–2, 2–4
　$DELTVA 2–6
　$EXPREG 2–6
　　メモリ割り当て 2–12
　$GETJPI 2–6
　$GETQUI 2–7
　$GETSYI 1–8, 2–7, 2–31
　$GETUAI 2–7
　$LKWSET 2–7, 2–36
　$MGBLSC 2–8

システム・サービス (続き)
$PURGWS 2–8
$SETPRT 2–8
$SETUAI 2–8
$SNDJBC 2–9
$ULKPAG 2–9
$ULWSET 2–9
$UPDSEC 2–9
メモリ管理機能 2–3
ジャケット・ルーチン
代用イメージの作成 6–14
条件コード
識別 5–7
条件処理
AXP システム 5–1
アラインメント・フォルトの報告 . . . 5–13
オーバーフローの通知 5–16
算術演算例外 5–10
シグナル・アレイ形式 5–2
条件コード 5–7
条件ハンドラ
作成 5–2
指定 5–17
ハードウェア例外 5–8
巻き戻し 5–6
メカニズム・アレイ 5–3
ランタイム・ライブラリ・サポート・ルーチン 5–15
状態値
シグナル・アレイ 5–3
シンボル
PAGE_COUNT 2–14
再定義
DEC C のVAX Cとの互換性 A–12
シンボル・インフォメーション・ファイル (SIF)
形式 6–12
使用方法 6–10
シンボル・テーブル・ファイル (SYS.STB) 1–5
シンボル・ベクタ
レイアウトの制御 6–10

セ

セクションのマッピング 2–1
セグメントの保護 2–1

ソ

相互操作性 (interoperability)
/BPAGE 6–6
コンパイル時に考慮すること 6–2
シンボル・ベクタ・レイアウトの制御 6–10
代用イメージの作成 6–14
トランスレートされたイメージとネイティブなイメージ 6–1
ネイティブ・イメージの作成
トランスレートされたイメージから呼び出せる 6–7
トランスレートされたイメージを呼び出す 6–3
ネイティブな AXP イメージ
コンパイル 6–2
リンク 6–3

タ

代用イメージ
作成 6–14

テ

テキスト・ライブラリ
移植 A–12
データ
VAX FORTRAN から DEC Fortran への移植 A–52
アラインメント A–2
表現 A–2
無意識に共有される 3–13
データ・アラインメント
DEC C で無効な機能 A–11
例外報告 5–13
データ型 (data type)
Alpha AXP アーキテクチャ 4–2

データ型 (data type) (続き)
DEC Fortran とVAX FORTRANの違い A–52
VAX アーキテクチャ 4–2
移植性 (portability) 4–1
サイズ
DEC C でのサポート A–7
DEC C 移植用マクロ A–7
共有データの保護 3–14
データ構造体の初期化
DEC C のVAX Cとの相違点 A–12
転送ベクタ・ファイル 1–4

ト

同期 . . . . . 3–1〜3–17
OpenVMS AXP 互換性機能 3–4
VAX アーキテクチャの機能 3–2
VAX アプリケーションで仮定された保証の検出 3–5
トランスレートされたイメージの . . . 3–17
プログラム例 3–8
トランスレートされたイメージ
コールバック 6–2
作成 6–1
ネイティブな AXP イメージとの置換 6–9
不可分性の保証 3–17
リンカ・オプションでの使用 6–7

ハ

バイト粒度 (byte granularity)
同期に関する影響 3–4
ハンドラ・データ・アドレス
メカニズム・アレイ 5–5

ヒ

引数の数
シグナル・アレイ 5–3
メカニズム・アレイ 5–5
引数リスト
OpenVMS AXP コンパイラの DEC C A–11
表記法 xiii

フ

ファイル・タイプ
AXP システムの 1–4
深さ
メカニズム・アレイ 5–5
不可分性 (atomicity)
DEC C のサポート A–10
トランスレートされたイメージでの保証 3–17
不可分な (atomic) 命令
同期に関する影響 3–3
浮動小数点データ型
CVT$CONVERT_FLOAT RTL ルーチン A–52
DEC C での形式の指定 A–8
H 浮動小数点データの変換 A–52
VAX FORTRAN
と DEC Fortran の違い A–52
VAX 型と AXP 型 A–52
VAX リトル・エンディアン形式 A–52
フラグ
メカニズム・アレイ 5–5
プラグマ A–3
COMPONENT_ALIGNMENT A–3
FLOAT_REPRESENTATION A–3
LONG_FLOAT A–3
MAIN_STORAGE A–3
SHARE_GENERIC A–3
SHARED A–3
TIME_SLICE A–3
フレーム・ポインタ
メカニズム・アレイ 5–5
プログラム・カウンタ (PC)
シグナル・アレイ 5–3
プロシージャ・シグナチャ・ブロック (PSB)
作成 6–2
プロセッサ・ステータス・ロングワード (PSL)
シグナル・アレイ 5–3

へ

ページ・サイズ 1–8
  $GETSYI の使用 2–35
  AXP システム 2–2
  OpenVMS VAX との互換性 2–2
  VAX に依存する 2–1
ページレット
  定義 2–2

ホ

ホスト・アーキテクチャ 1–8

マ

巻き戻し
  条件処理 5–6
マッピング
  オフセットによるセクションファイル 2–32
  拡張した仮想アドレス空間 2–18
  単一ページ 2–21
  定義されたアドレス範囲 2–22
  ページ・サイズへの依存 2–17

メ

メカニズム・アレイ
  depth 引数 5–6
  形式 5–3
  シグナル・アレイのアドレス 5–5
  ハンドラ・データ・アドレス 5–5
  引数の数 5–5
  深さ 5–5
  フラグ 5–5
  フレーム・ポインタ 5–5
  リザーブ 5–5
  例外スタック・フレーム・アドレス 5–5
  レジスタ 5–5
メモリ
  管理機能 2–1
  管理ルーチン . . . . . 2–3〜2–10
メモリ (続き)
  マッピング
    $CRMPSC 2–20
    拡張した仮想アドレス空間 2–18
    単一ページ 2–21
    定義されたアドレス範囲 2–22
    必要な変更 2–27
    ページ・サイズへの依存 2–17
  ロック 2–1
  ワーキング・セットとしてロックする操作 2–36
  割り当て
    $CRETVA による 2–15
    $EXPREG による 2–13
    アドレス領域の指定 2–15
    解除 2–16
    拡張された仮想アドレス空間 . . . 2–12
    既存の仮想アドレス空間 2–15
    ルーチン 2–11

モ

文字列定数
  変更 A–12

ユ

ユニバーサル・シンボル 1–4

ヨ

読み込み/書き込み操作の順序 3–15
  同期に関する影響 3–4

ラ

ランタイム・ライブラリ 2–10
ランタイム・ライブラリ・ルーチン
  LIB$FREE_VM_PAGE 2–10
  LIB$GET_VM_PAGE 2–10

リ

リザーブ
  メカニズム・アレイ 5–5
リンカ
  AXP システム 1–6
  オプション
    BASE 1–7
    DZRO_MIN 1–7
    ISD_MAX 1–7
    /NATIVE_ONLY 6–7
    SYMBOL_VECTOR 1–4, 1–7, 6–11
    SYMBOL_TABLE 1–7
    UNIVERSAL 1–7
  修飾子
    /BPAGE 6–6
    /DEMAND_ZERO 1–6
    /GST 1–6
    /INFORMATIONALS 1–6
    /NATIVE_ONLY 1–6
    /NOINFORMATIONALS 1–6
    /REPLACE 1–6
    /SECTION_BINDING 1–6
    /SIF 6–12
    /SYSEXE 1–6
リンク
  ネイティブ・イメージの作成 . . . 1–4, 6–3

ル

ルーチン
  LIB$MATCH_COND 5–8
  SYS$UNWIND 5–6

レ

例外
  SS$_ALIGN 5–9
    シグナル・アレイ 5–13
  SS$_HPARITH 5–9
    シグナル・アレイ 5–11
例外処理ハンドラ
  巻き戻し 5–6
例外スタック・フレーム・アドレス
  メカニズム・アレイ 5–5
レジスタ
  メカニズム・アレイ 5–5

OpenVMS AXP **オペレーティング・システム**
OpenVMS AXP **オペレーティング・システムへの移行：再コンパイルと再リンク**

1994 年 7 月　発行

日本ディジタル イクイップメント株式会社

〒 167 東京都杉並区上荻 1-2-1

電話　(03)5349-7111 (大代表)

AA-PU8LC-TE