

# HP OpenVMS

---

## OpenVMS Alpha から OpenVMS I64 への アプリケーション・ポーティング・ガイド

注文番号: AA-RX0JA-TE

2005 年 4 月

このドキュメントは、HP OpenVMS Alpha から HP OpenVMS Industry Standard 64 for Integrity Servers へアプリケーションを移行しようとしているアプリケーション開発者に対して、ポーティング作業の概要を説明しています。

改訂情報:	新規ドキュメントです。
ソフトウェア・バージョン:	OpenVMS I64 V8.2 OpenVMS Alpha V8.2

---

© Copyright 2005 Hewlett-Packard Development Company, L.P.

本書の著作権は Hewlett-Packard Development Company, L.P. が保有しており、本書中の解説および図、表は Hewlett-Packard Development Company, L.P. の文書による許可なしに、その全体または一部を、いかなる場合にも再版あるいは複製することを禁じます。

また、本書に記載されている事項は、予告なく変更されることがありますので、あらかじめご承知おきください。万一、本書の記述に誤りがあった場合でも、弊社は一切その責任を負いかねます。

本書で解説するソフトウェア (対象ソフトウェア) は、所定のライセンス契約が締結された場合に限り、その使用あるいは複製が許可されます。

日本ヒューレット・パカードは、弊社または弊社の指定する会社から納入された機器以外の機器で対象ソフトウェアを使用した場合、その性能あるいは信頼性について一切責任を負いかねます。

Intel®, Xeon™, および Itanium®は、米国およびその他の国における Intel Corporation またはその子会社の商標または登録商標です。

UNIX®は、The Open Group の登録商標です。

Java™は、米国における Sun Microsystems, Inc. の商標です。

原典：Porting Applications from HP OpenVMS Alpha to OpenVMS Industry  
Standard 64 for Integrity Servers

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	ix
1 はじめに	
1.1 OpenVMS Industry Standard 64 for Integrity Servers	1-1
1.2 ポーティング・プロセスの概要	1-1
1.2.1 アプリケーションの評価	1-2
2 基本的な相違点	
2.1 呼び出し規則	2-1
2.1.1 OpenVMS の呼び出し規則の変更点	2-1
2.1.2 OpenVMS 呼び出し規則の拡張	2-3
2.2 浮動小数点演算	2-4
2.2.1 概要	2-4
2.2.2 OpenVMS アプリケーションへの影響	2-5
2.3 オブジェクト・ファイルの形式	2-5
3 ポーティングが必要なアプリケーションの調査	
3.1 概要	3-1
3.2 アプリケーションの評価	3-2
3.2.1 ポーティング方法の選択	3-3
3.2.2 ポーティングが必要なアプリケーションの洗い出し	3-3
3.2.3 依存性の評価	3-6
3.2.3.1 ソフトウェアへの依存性	3-6
3.2.3.2 開発環境	3-7
3.2.3.3 オペレーティング環境	3-7
3.2.4 運用タスク	3-8
3.3 HP から提供されるポーティング・リソース	3-9
3.4 その他の検討事項	3-10
4 ソース・モジュールの移行	
4.1 移行環境の用意	4-2
4.1.1 ハードウェア	4-2
4.1.2 ソフトウェア	4-3
4.1.2.1 HP OpenVMS Migration Software for Alpha to Integrity Servers と Translated Image Environment (TIE)	4-4
4.1.3 変換イメージとの共存	4-5
4.2 最新バージョンのコンパイラによる Alpha でのアプリケーションのコンパイル	4-7

4.3	移行するアプリケーションの Alpha 上での動作を確認するためのテスト .....	4-8
4.4	I64 システムでの再コンパイルと再リンク .....	4-8
4.5	移行したアプリケーションのデバッグ .....	4-8
4.5.1	デバッグ .....	4-9
4.5.2	システム・クラッシュの分析 .....	4-9
4.5.2.1	System Dump Analyzer .....	4-9
4.5.2.2	Crash Log Utility Extractor .....	4-10
4.6	移行したアプリケーションのテスト .....	4-10
4.6.1	I64 への Alpha テスト・ツールのポーティング .....	4-10
4.6.2	新しい I64 テスト .....	4-11
4.6.3	潜在的なバグの検出 .....	4-11
4.7	移行したアプリケーションのソフトウェア・システムへの統合 .....	4-11
4.8	特定のタイプのコードの変更 .....	4-12
4.8.1	条件付きコード .....	4-12
4.8.1.1	MACRO のソース .....	4-13
4.8.1.2	BLISS のソース .....	4-13
4.8.1.3	C のソース .....	4-14
4.8.1.4	既存の条件付きコード .....	4-14
4.8.2	Alpha アーキテクチャに依存しているシステム・サービス .....	4-15
4.8.2.1	SYSSGOTO_UNWIND .....	4-15
4.8.2.2	SYSSLKWSET と SYSSLKWSET_64 .....	4-15
4.8.3	Alpha アーキテクチャに依存するその他の機能を含むコード .....	4-16
4.8.3.1	初期化済みのオーバーレイ・プログラム・セクション .....	4-16
4.8.3.2	条件ハンドラでの SS\$_HPARITH の使用 .....	4-16
4.8.3.3	メカニズム・アレイ・データ構造 .....	4-16
4.8.3.4	Alpha のオブジェクト・ファイル形式への依存 .....	4-16
4.8.4	浮動小数点データ・タイプを使用するコード .....	4-17
4.8.4.1	LIB\$WAIT に関する問題と解決策 .....	4-18
4.8.5	コマンド・テーブル宣言に関する注意 .....	4-19
4.8.6	スレッドを使用するコード .....	4-20
4.8.6.1	スレッド・ルーチン cma_delay および cma_time_get_expiration .....	4-21
4.8.7	自然な境界に配置されていないデータを含むコード .....	4-22
4.8.8	OpenVMS Alpha の呼び出し規則に依存するコード .....	4-24
4.8.9	特権コード .....	4-24
4.8.9.1	SYSSLKWSET と SYSSLKWSET_64 の使用 .....	4-24
4.8.9.2	SYSSLCKPAG および SYSSLCKPAG_64 の使用 .....	4-25
4.8.9.3	ターミナル・ドライバ .....	4-26
4.8.9.4	保護されたイメージ・セクション .....	4-26
5	OpenVMS I64 開発環境	
5.1	I64 のネイティブ・コンパイラ .....	5-1
5.1.1	VAX MACRO-32 Compiler for OpenVMS I64 .....	5-2
5.2	その他の開発ツール .....	5-2
5.2.1	Alpha コードの変換 .....	5-3
5.3	モジュールのリンク .....	5-3

5.3.1	OpenVMS I64 システムでリンクする場合の相違点	5-4
5.3.1.1	CLUSTER オプションでのベース・アドレスの指定	5-5
5.3.1.2	OpenVMS I64 での初期化済みオーバーレイ・プログラム・セクションの取り扱い	5-5
5.3.1.3	ELF 共通シンボルをリンクする際の動作の違い	5-6
5.3.2	拡張されたマップ・ファイル情報	5-7
5.3.3	OpenVMS I64 用の新しいリンカ修飾子とオプション	5-7
5.3.3.1	新しい /BASE_ADDRESS 修飾子	5-7
5.3.3.2	新しい /SEGMENT_ATTRIBUTE 修飾子	5-8
5.3.3.3	新しい /FP_MODE 修飾子	5-8
5.3.3.4	新しい /EXPORT_SYMBOL_VECTOR および /PUBLISH_GLOBAL_SYMBOLS 修飾子	5-9
5.3.3.5	PSECT_ATTRIBUTE オプションのための新しいアラインメント	5-9
5.3.3.6	/FULL 修飾子の新しいキーワード GROUP_SECTIONS および SECTION_DETAILS	5-10
5.3.4	リンカ・オプションの引数における大文字小文字の区別	5-10
5.4	OpenVMS I64 システムでのデバッグ機能	5-11
5.4.1	OpenVMS デバッガ	5-11
5.4.1.1	アーキテクチャのサポート	5-12
5.4.1.2	言語のサポート	5-12
5.4.1.3	機能とコマンド	5-13
5.4.1.4	まだ移植されていない機能	5-13
5.4.2	XDelta デバッガ	5-13
5.4.2.1	OpenVMS I64 での XDelta の機能	5-14
5.4.2.2	OpenVMS I64 システムと OpenVMS Alpha システムの XDelta の相違点	5-14
5.5	OpenVMS I64 Librarian ユーティリティ	5-14
5.5.1	I64 Librarian を使用する場合の検討事項	5-15
5.5.2	LBR\$ルーチンの変更点	5-15
5.5.3	UNIX スタイルの弱いシンボルを取り扱う I64 ライブラリ形式	5-16
5.5.3.1	弱いシンボルに対する新しい ELF タイプ	5-16
5.5.3.2	バージョン 6.0 ライブラリのインデックスの形式	5-16
5.5.3.3	新しいグループ・セクション・シンボル	5-17
5.5.3.4	弱いシンボルとグループ・シンボルに関する現在のライブラリの制限事項	5-17
6	アプリケーションのポーティングの準備	
6.1	ADA	6-2
6.2	BASIC	6-2
6.3	BLISS コンパイラ	6-2
6.3.1	BLISS ファイル・タイプとファイル位置のデフォルト	6-3
6.3.2	提供されない Alpha BLISS の機能	6-4
6.3.3	OpenVMS I64 BLISS の機能	6-6
6.4	COBOL	6-19
6.4.1	浮動小数点演算	6-19
6.4.2	/ARCH および /OPTIMIZE=TUNE 修飾子	6-19
6.5	Fortran	6-20
6.5.1	浮動小数点演算	6-20
6.5.2	サポートされるのは F90 コンパイラのみ	6-22

6.5.3	/ARCH および/OPTIMIZE=TUNE 修飾子	6-23
6.6	HP C/ANSI C	6-23
6.6.1	I64 浮動小数点のデフォルト	6-23
6.6.2	/IEEE_MODE 修飾子のセマンティック	6-24
6.6.3	定義済みマクロ	6-25
6.7	HP C++	6-25
6.7.1	浮動小数点と定義済みマクロ	6-25
6.7.2	long double	6-25
6.7.3	オブジェクト・モデル	6-25
6.7.4	言語ダイアレクト	6-26
6.7.5	テンプレート	6-26
6.8	Java	6-26
6.9	MACRO-32	6-26
6.10	HP Pascal	6-27
6.10.1	/ARCH および/OPTIMIZE=TUNE 修飾子	6-27
7	その他の検討事項	
7.1	ハードウェアに関する検討事項	7-1
7.1.1	Intel Itanium プロセッサ・ファミリの概要	7-1
7.1.2	Alpha プロセッサ・ファミリの概要	7-2
7.2	エンディアンに関する検討事項	7-3
A	アプリケーション評価チェックリスト	
B	サポート対象外のレイヤード・プロダクト	
C	アプリケーション固有のスタック切り換えコードの I64 へのポーティング	
C.1	KP サービスの概要	C-2
C.1.1	用語	C-2
C.1.2	スタックとデータ構造	C-3
C.1.3	KPB	C-4
C.1.4	提供される KPB 割り当てルーチン	C-5
C.1.5	カーネル・モードの割り当て	C-6
C.1.6	モードに依存しない割り当て	C-7
C.1.7	スタック割り当て API	C-9
C.1.8	システムで提供される割り当てルーチンと割り当て解除ルーチン	C-10
C.1.9	終了ルーチン	C-11
C.2	KP 制御ルーチン	C-11
C.2.1	概要	C-12
C.2.2	ルーチン説明	C-14
C.2.2.1	EXE\$KP_START	C-14
C.2.2.2	EXE\$KP_STALL_GENERAL	C-14
C.2.2.3	EXE\$KP_RESTART	C-15
C.2.2.4	EXE\$KP_RESTART	C-15
C.3	設計上の考慮点	C-16

## 用語集

## 索引

### 図

3-1	アプリケーションのポーティング .....	3-4
4-1	Alpha から I64 へのアプリケーションの移行 .....	4-2
7-1	ビッグ・エンディアン・バイト順 .....	7-4
7-2	リトル・エンディアン・バイト順 .....	7-4
C-1	KP ルーチンの実行 .....	C-13

### 表

2-1	OpenVMS の呼び出し規則の変更点 .....	2-1
2-2	OpenVMS 呼び出し規則の拡張 .....	2-4
4-1	コンパイル時の参照を報告するためのコンパイラ・スイッチ .....	4-23
5-1	OpenVMS の開発ツール .....	5-2
6-1	Alpha コンパイラのバージョンと I64 コンパイラのバージョンの対応関係 ....	6-1
B-1	OpenVMS I64 にポーティングされないレイヤード・プロダクト .....	B-1
C-1	モードおよびスコープ別割り当てガイドライン .....	C-4
C-2	カーネル・モード割り当てフラグ .....	C-7
C-3	モードに依存しない割り当てのフラグ .....	C-8





## 対象読者

『HP OpenVMS Alpha から OpenVMS I64 へのアプリケーション・ポーティング・ガイド』は、OpenVMS Alpha から OpenVMS I64 へアプリケーションを移行しようとするアプリケーション開発者を対象としています。

## 本書の構成

本書の構成は以下のとおりです。

第 1 章では、OpenVMS I64 8.2 オペレーティング・システムについて概要を説明しています。

第 2 章では、OpenVMS Alpha と OpenVMS I64 との基本的な相違点について説明しています。

第 3 章では、OpenVMS I64 へのポーティングの準備として、アプリケーションを評価する作業の概要を説明しています。

第 4 章では、移行のためにソース・モジュールに対して行う作業について説明します。ポーティングを行うアプリケーションのコンパイル、リンク、テスト、および本番システムへの導入についての情報も提供します。

第 5 章では、OpenVMS I64 の開発環境について説明します。

第 6 章では、OpenVMS I64 で利用できる主なコンパイラに関する注意事項を説明します。

第 7 章では、ポーティングに関するその他の注意事項を説明します。

付録 A には、ポーティングを行う前にアプリケーションを評価するのに利用できるサンプル・チェックリストを示します。

付録 B には、OpenVMS Alpha でサポートされていて OpenVMS I64 ではサポートされていない HP レイヤード・プロダクトの一覧を示します。

付録 C では、アプリケーション固有のスタック切り替えコードを I64 へポーティングする際の KP ルーチンの使い方について説明します。

用語集では、本書で使用されている重要な用語と省略形について説明します。

## 関連資料

以下のドキュメントも参考に参考にしてください。

- 『OpenVMS Calling Standard』
- 『OpenVMS デバッガ説明書』
- 『OpenVMS Linker Utility Manual』
- 『OpenVMS System Analysis Tools Manual』

個々のコンパイラのドキュメントもポーティング作業の際に役に立ちます。

ホワイト・ペーパー『Intel® Itanium®における OpenVMS 浮動小数点演算について』では、OpenVMS Alpha と OpenVMS I64 の間での、浮動小数点のデータ型の違いについて説明しています。このホワイト・ペーパーは、次の場所から入手できます。

<http://h50146.www5.hp.com/products/software/oe/openvms/technical/>

HP OpenVMS の製品情報およびサービス情報については、以下の Web サイトを参照してください。

<http://www.hp.com/go/openvms>

<http://www.hp.com/jp/openvms>

## 本書の表記法

本書の表記法は以下のとおりです。

Ctrl/x	Ctrl/xという表記は、Ctrl キーを押しながら別のキーまたはポインティング・デバイス・ボタンを押すことを示します。
PF1 x	PF1 xという表記は、PF1 に定義されたキーを押してから、別のキーまたはポインティング・デバイス・ボタンを押すことを示します。
<div>Return</div>	<p>例の中で、キー名が四角で囲まれている場合には、キーボード上でそのキーを押すことを示します。テキストの中では、キー名は四角で囲まれていません。</p> <p>HTML 形式のドキュメントでは、キー名は四角ではなく、括弧で囲まれています。</p>

...	例の中の水平方向の反復記号は、次のいずれかを示します。
	<ul style="list-style-type: none"> <li>• 文中のオプションの引数が省略されている。</li> <li>• 前出の 1 つまたは複数の項目を繰り返すことができる。</li> <li>• パラメータや値などの情報をさらに入力できる。</li> </ul>
.	垂直方向の反復記号は、コードの例やコマンド形式の中の項目が省略されていることを示します。このように項目が省略されるのは、その項目が説明している内容にとって重要ではないからです。
()	コマンドの形式の説明において、括弧は、複数のオプションを選択した場合に、選択したオプションを括弧で囲まなければならないことを示しています。
[]	コマンドの形式の説明において、大括弧で囲まれた要素は任意のオプションです。オプションをすべて選択しても、いずれか 1 つを選択しても、あるいは 1 つも選択しなくても構いません。ただし、OpenVMS ファイル指定のディレクトリ名の構文や、割り当て文の部分文字列指定の構文の中では、大括弧に囲まれた要素は省略できません。
[ ]	コマンド形式の説明では、括弧内の要素を分けている垂直棒線はオプションを 1 つまたは複数選択するか、または何も選択しないことを意味します。
{ }	コマンドの形式の説明において、中括弧で囲まれた要素は必須オプションです。いずれか 1 のオプションを指定しなければなりません。
太字	太字のテキストは、新しい用語、引数、属性、条件を示しています。
<i>italic text</i>	イタリック体のテキストは、重要な情報を示します。また、システム・メッセージ (たとえば内部エラー <i>number</i> )、コマンド・ライン (たとえば <i>/PRODUCER=name</i> )、コマンド・パラメータ (たとえば <i>device-name</i> ) などの変数を示す場合にも使用されます。
UPPERCASE TEXT	英大文字のテキストは、コマンド、ルーチン名、ファイル名、ファイル保護コード名、システム特権の短縮形を示します。
Monospace type	モノスペース・タイプの文字は、コード例および会話型の画面表示を示します。  C プログラミング言語では、テキスト中のモノスペース・タイプの文字は、キーワード、別々にコンパイルされた外部関数およびファイルの名前、構文の要約、または例に示される変数または識別子への参照などを示します。
—	コマンド形式の記述の最後、コマンド・ライン、コード・ラインにおいて、ハイフンは、要求に対する引数がその後の行に続くことを示します。
数字	特に明記しない限り、本文中の数字はすべて 10 進数です。10 進数以外 (2 進数、8 進数、16 進数) は、その旨を明記してあります。



---

## はじめに

この章では、Integrity Server (I64) 向けの OpenVMS Industry Standard 64 システムについて紹介し、移行プロセスの概要を示します。

詳細については、『HP OpenVMS Version 8.2 リリース・ノート[翻訳版]』を参照してください。

---

### 1.1 OpenVMS Industry Standard 64 for Integrity Servers

OpenVMS I64 アーキテクチャの 64 ビット・モデルおよび基本的なシステム機能は、Alpha アーキテクチャとほぼ同じです。そのため、OpenVMS Alpha で現在稼動しているアプリケーションの大部分は、ほとんど変更せず、簡単に I64 アーキテクチャへのポータリングが可能です。

OpenVMS Alpha と OpenVMS I64 のそれぞれの OpenVMS オペレーティング・システムは単一ソースのコード・ベースから開発されるため、各バージョンでソース・コードを個別に変更しなくても、ハードウェアに依存しない機能を両方のバージョンに組み込むことができます。その結果、評価テストに必要な時間を大幅に短縮でき、両方の OpenVMS プラットフォームで重要なアプリケーションを確実に提供することができます。

HP は、OpenVMS Alpha の推奨バージョンで現在「正常に動作」しているアプリケーションが OpenVMS I64 システムでも正常に動作するように、上位互換性を保証していくための厳密なポリシーを設定しています。一般に公開されているシステム・サービスやライブラリ・インタフェースを使用しているアプリケーションであれば、ソース・コードをまったく変更せずに最新バージョンの OpenVMS I64 に移行できるはずです。

OpenVMS I64 は、OpenVMS の利用者が慣れ親しんだ見た目と使い勝手 (ルック・アンド・フィール) を備えています。新しいアーキテクチャに対応するためにわずかな変更は行っていますが、OpenVMS の基本構造および機能については違いはありません。

---

### 1.2 ポータリング・プロセスの概要

ここでは、ほとんどのアプリケーションに適用できるポータリング・プロセスの概要を示します。

### 1.2.1 アプリケーションの評価

アプリケーションの評価では、HP OpenVMS I64 への移行に必要な手順を確認します。評価の結果、以下の質問に対する回答から移行プランを作成できます。

- アプリケーションの移行方法
- 移行に必要な時間、作業量、コスト
- HP のサービス部門から提供されるサポートの必要性

評価は次の 3 段階に分かれます。

1. 他のソフトウェアへの依存性の特定も含めた、アプリケーションの全般的な洗い出し
2. 他のソフトウェアへの依存性を特定するためのソースの分析
3. 移行方法の選択 (ソースから再コンパイル・再リンクするのか、バイナリ・トランスレータを使用するのか)

これらの評価を完了すると、効果的な移行プランを作成するのに必要な情報が得られます。

移行のためのアプリケーション評価の第 1 ステップでは、何を移行するのかを正確に判断します。このステップでは、アプリケーションだけでなく、アプリケーションを正常に実行するのに必要なあらゆる要素の洗い出しを行います。アプリケーションの評価を開始するには、以下の各項目を特定し、その場所を確認します。

- アプリケーションの構成要素
  - メイン・プログラムのソース・モジュール
  - 共有イメージ
  - オブジェクト・モジュール
  - ライブラリ (オブジェクト・モジュール、共有イメージ、テキスト、またはマクロ)
  - データ・ファイルとデータベース
  - メッセージ・ファイル
  - ドキュメント
- アプリケーションが依存している他のソフトウェア。以下のものが含まれる。
  - ランタイム・ライブラリ
  - HP 製レイヤード・ソフトウェア
  - 他社製ソフトウェア
- 必要とされるオペレーティング環境
  - システム属性

アプリケーションの実行と保守に必要なシステム要件。たとえば、必要なメモリ容量やディスク容量など。

- ビルド・プロシージャ

たとえば、CMS (Code Management System) や MMS (Module Management System) などのツールがこれに含まれます。

- テスト・ツール

テストでは、移行したアプリケーションが正常に動作することを確認し、パフォーマンスを評価する必要があります。

これらのコンポーネントの多くは、すでに OpenVMS I64 への移行が完了しています。たとえば、以下のものが移行されています。

- OpenVMS オペレーティング・システムに付属している HP 製ソフトウェア
  - ランタイム・ライブラリ
  - その他の共有ライブラリ、たとえば、呼び出し可能ユーティリティ・ルーチンやその他のアプリケーション・ライブラリ・ルーチン
- HP 製レイヤード・ソフトウェア
  - コンパイラとコンパイラ・ランタイム・ライブラリ
  - データベース・マネージャ
  - ネットワーク環境
- 他社製品

すでに多くの他社製品が OpenVMS I64 で稼動しています。特定のアプリケーションが I64 に対応済みかどうかについては、アプリケーションを提供しているベンダにお問い合わせください。

ビルド・プロシージャやテスト・ツールも含めて、アプリケーションおよび開発環境の移行は、お客様の責任で行っていただく必要があります。

アプリケーションの評価が完了したら、第 4 章の説明に従って、各モジュールおよびイメージの詳細な評価を行います。





## 基本的な相違点

この章では、Alpha アーキテクチャと I64 アーキテクチャの基本的な相違点について説明します。

### 2.1 呼び出し規則

Intel® Itanium® プロセッサ・ファミリ上で OpenVMS 呼び出し規則を実装する際には、他のコンポーネントと同様に、OpenVMS VAX および OpenVMS Alpha の規則との違いができるだけ少なくなるように考慮しました。設計にあたっては、基本的に Itanium の呼び出し規則に従い、必要な場合だけ変更するようにしました。その結果、アプリケーションおよびオペレーティング・システムを Itanium アーキテクチャに移植する際に必要となるコストと問題点をできるだけ少なくする一方で、従来の OpenVMS の設計との互換性を維持することが可能になっています。

以降の各節では、『Itanium® Software Conventions and Runtime Architecture Guide』に定義されている規則と OpenVMS の呼び出し規則の相違点について説明します。詳細については、『OpenVMS Calling Standard』を参照してください。

#### 2.1.1 OpenVMS の呼び出し規則の変更点

OpenVMS I64 での、OpenVMS の呼び出し規則の主な変更点を、表 2-1 に示します。

表 2-1 OpenVMS の呼び出し規則の変更点

項目	説明
データ・モデル	OpenVMS Alpha は、使用されているデータ・モデルに関して意図的にあいまいになっています。多くのプログラムは ILP32 モデルであるかのようにコンパイルされますが、システムの大部分は P64 または LP64 モデルを使用しているかのように動作します。整数の符号拡張規則は、このデータ・モデルの透過性を向上または低下させるのに重要な役割を果たします。Itanium アーキテクチャの規則では純粋な LP64 データ・モデルが定義されていますが、OpenVMS I64 では上記の属性を維持しています。

(次ページに続く)

## 基本的な相違点

### 2.1 呼び出し規則

表 2-1 (続き) OpenVMS の呼び出し規則の変更点

項目	説明
データを表す用語	OpenVMS I64 のこの規則では、ワードは 2 バイト、コードワードは 8 バイトを表します。一方、Itanium の用語では、ワードは 4 バイト、コードワードは 16 バイトを表します。
汎用レジスタの使用	<p>汎用レジスタは、整数演算、VAX 浮動小数点エミュレーションの一部、およびその他の汎用演算に使用されます。OpenVMS は、これらのレジスタをこれまでと同じように使用しますが (デフォルト)、以下の例外があります。</p> <ul style="list-style-type: none"><li>• R8 と R9 に戻り値が格納されます。</li><li>• R10 と R11 はスクラッチ・レジスタとして使用され、値を返すためには使用されません。</li><li>• R25 は AI (引数情報) レジスタとして使用されます。</li></ul>
浮動小数点レジスタの使用	<p>浮動小数点レジスタは、浮動小数点演算、VAX 浮動小数点エミュレーションの一部、特定の整数演算で使用されます。OpenVMS は、これらのレジスタに対して同じ規則を使用しますが (デフォルトの動作)、以下の例外があります。</p> <ul style="list-style-type: none"><li>• F8 と F9 に戻り値が格納されます。</li><li>• F10 ~ F15 はスクラッチ・レジスタとして使用され、値を返すためには使用されません。</li></ul>

(次ページに続く)

表 2-1 (続き) OpenVMS の呼び出し規則の変更点

項目	説明
引数の受け渡し	<p>OpenVMS の引数の受け渡しは、Itanium の規則とほぼ同じです。ただし、OpenVMS 規則での以下の相違点に注意してください。</p> <ul style="list-style-type: none"> <li>• OpenVMS 規則には、AI (引数情報) レジスタが追加されています (引数の数およびデータ・タイプに関する情報用)。</li> <li>• 引数は汎用レジスタにも浮動小数点レジスタにもコピーされません。</li> <li>• レジスタを介して受け渡される引数の場合、最初の引数は最初の汎用レジスタ・スロット (R32) または最初の浮動小数点レジスタ・スロット (F16) に渡され、2 番目の引数は 2 番目の汎用レジスタ・スロット (R33) または 2 番目の浮動小数点レジスタ・スロット (F17) に渡されます。以下も同様です。浮動小数点引数が使用可能な浮動小数点レジスタにバックされることはなく、引数は最大 8 つまで、レジスタを介して受け渡されます。</li> <li>• 汎用レジスタを介して受け渡される引数の場合、32 ビット値は、ビット 31 をコピーすることで、完全な 64 ビットのパラメータ・スロットに符号拡張されます (符号なしデータ・タイプの場合も同様)。</li> <li>• 64 ビットより大きな引数の場合、偶数スロット・アライメントは行われません。</li> <li>• 一般に、HFA (homogeneous floating-point aggregate) に対する特別な処理は行われません。</li> <li>• OpenVMS では、<code>_float128</code> の値渡しであっても、参照渡しとして実装されています。</li> <li>• OpenVMS では、リトル・エンディアン表現だけがサポートされます。</li> <li>• OpenVMS では、下位互換性を維持するために、<code>F_floating</code> (32 ビット)、<code>D_floating</code> (64 ビット)、および <code>G_floating</code> (64 ビット) の 3 つの VAX 浮動小数点データ・タイプが追加サポートされます。これらのデータ・タイプの値は、汎用レジスタを使用して渡されます。</li> </ul>
戻り値	<p>最大 16 バイトまでの戻り値は、レジスタを介して呼び出し元に返すことができます。それより大きな戻り値は、最初または 2 番目のパラメータ・スロットを使用して、隠し引数方式で返されます。</p>

### 2.1.2 OpenVMS 呼び出し規則の拡張

表 2-2 では、OpenVMS 呼び出し規則の追加規則や拡張規則について説明します。

表 2-2 OpenVMS 呼び出し規則の拡張

項目	説明
浮動小数点データ・タイプ	OpenVMS の呼び出し規則では、VAX および Alpha システムでサポートされていた VAX F_floating (32 ビット)、D_floating (64 ビット)、G_floating (64 ビット) データ・タイプがサポートされます。Itanium アーキテクチャの 80 ビット倍精度拡張浮動小数点データ・タイプはサポートされません。
VAX 互換のレコード・レイアウト	OpenVMS の呼び出し規則では、ユーザ・オプションとして VAX 互換のレコード・レイアウトが追加されています。
リンケージ・オプション	OpenVMS では、汎用レジスタを入力や出力、グローバル・レジスタ、その他の目的で使用する際に、柔軟性とユーザの制御機能を向上することができます。
メモリ・スタック・オーバーフローのチェック	OpenVMS では、メモリ・スタック・オーバーフローのチェック方法が定義されています。
関数記述子	OpenVMS では、バインドされたプロシージャ値や変換されたイメージのサポートのために、関数記述子の拡張形式を定義して追加機能をサポートしています。
アンwind情報	OpenVMS では、オペレーティング・システム固有のデータ領域が Itanium アーキテクチャのアンwind情報ブロックに追加されています。オペレーティング・システム固有のデータ領域の存在は、アンwind情報ヘッダのフラグによって示されます。
ハンドラの起動	OpenVMS では、制御がルーチンのプロローグ領域またはエピローグ領域にある間、ハンドラを起動しません。この動作の違いは、アンwind情報ヘッダのフラグによって示されます。
変換されたイメージ	OpenVMS では、VAX または Alpha のネイティブ・イメージと変換されたイメージの間の呼び出しのサポート (シグネチャ情報と特殊な ABI) が追加されています。

## 2.2 浮動小数点演算

この節では、OpenVMS VAX、OpenVMS Alpha、および OpenVMS I64 システムでの浮動小数点演算の相違点について説明します。

### 2.2.1 概要

Alpha アーキテクチャでは、IEEE 浮動小数点形式と VAX 浮動小数点形式の両方がハードウェアでサポートされます。OpenVMS コンパイラは、デフォルトでは VAX 形式を使用してコードを生成しますが、Alpha 上では IEEE 形式を使用するためのオプションも提供しています。Itanium アーキテクチャでは、IEEE 単精度と IEEE 倍精度を含む IEEE 浮動小数点形式を使用して、ハードウェアで浮動小数点演算を実装しています。

OpenVMS VAX または OpenVMS Alpha 用にデフォルトの浮動小数点形式を使用して作成されているアプリケーションを OpenVMS I64 へ移植する方法には、2 種類の方法があります。1 つは HP 製コンパイラの変換機能を利用して今後も VAX 浮動小

数点形式を使用する方法で、もう一つは IEEE 浮動小数点形式を使用するようにアプリケーションを変換する方法です。VAX 浮動小数点形式は、以前に生成されたバイナリの浮動小数点データにアクセスすることが必要な状況で使用できます。HP 製コンパイラは、VAX 形式と IEEE 形式の間でデータを変換するのに必要なコードを生成します。変換プロセスの詳細については、下記の Web サイトで提供されている『Intel Itanium アーキテクチャにおける OpenVMS 浮動小数点演算について』を参照してください。

<http://h50146.www5.hp.com/products/software/oe/openvms/technical/>

IEEE 浮動小数点形式は、VAX 浮動小数点形式が不要な場合に使用してください。IEEE 浮動小数点形式を使用すると、より効率のよいコードを生成できます。

### 2.2.2 OpenVMS アプリケーションへの影響

Itanium アーキテクチャでは、IEEE 単精度および IEEE 倍精度浮動小数点データ・タイプに変換することにより VAX 浮動小数点形式がサポートされます。デフォルトでは、この処理は透過的に実行されるため、ほとんどのアプリケーションには影響を与えません。HP は C、C++、Fortran、BASIC、Pascal、および COBOL 用のコンパイラを提供しており、すべてのコンパイラで同じ浮動小数点オプションが提供されます。アプリケーション開発者にとって必要な作業は、適切なコンパイラを使用してアプリケーションを再コンパイルすることだけです。浮動小数点オプションの詳細については、各コンパイラのドキュメントを参照してください。

IEEE 浮動小数点形式がデフォルトであるため、アプリケーションで VAX 浮動小数点形式オプションを明示的に指定しない限り、OpenVMS I64 向けにアプリケーションを単純に再ビルドするとネイティブな IEEE 形式が直接使用されます。VAX 形式に直接依存しなくても正常に動作するプログラムの場合は、OpenVMS I64 向けのビルド方式としては、これが最も望ましい方法です。

---

## 2.3 オブジェクト・ファイルの形式

Intel が開発したコンパイラ・テクノロジーを利用するための最も効率のよい方法は、以下に示す 2 つの業界標準 (いずれも Itanium のコンパイラで使用) を採用することで、と弊社は考えています。

- ELF (executable and linkable format) は、オブジェクト・ファイルと実行可能ファイルの形式を記述しています。
- DWARF (debugging and traceback information format) は、デバッグおよびトレースバック情報がオブジェクト・ファイルおよび実行可能ファイルに保存される方法を記述しています。

弊社が下したこの判断には、将来、開発ツールを OpenVMS へ移植する作業がより容易になるという明確な利点があります。OpenVMS I64 オペレーティング・システムで提供されるコンパイラ、開発ツール、およびユーティリティはすべて、これらの新しい形式を認識し、利用することができます。アプリケーションで ELF および DWARF を明示的に取り扱う場合を除き、アプリケーション開発者がこれらの形式について意識する必要はありません。

アプリケーションのコンパイル、リンク、デバッグ、そして導入が行われるという典型的なアプリケーション開発のシナリオでは、オブジェクト・ファイルの形式、実行イメージ・ファイルの形式、デバッグ情報やトレースバック情報などの詳細について、開発者が特に考慮することはありません。しかし、コンパイラやデバッガ、解析ツールをはじめ、これらのファイルの形式に依存するその他のユーティリティなどを開発するソフトウェア・エンジニアは、これらのファイルが OpenVMS でどのように実装されているかを理解しておく必要があります。

---

## ポーティングが必要なアプリケーションの調査

通常、新しいプラットフォームへのアプリケーションのポーティングでは、次の手順を実施することになります。

1. ポーティングの必要性の調査
2. アーキテクチャに依存するコード、および標準に準拠していないコードを必要に応じて書き直す作業
3. アプリケーションのコンパイル、リンク、および実行
4. ポーティング前後のテストの実施
5. プログラムの再コンパイルと、必要に応じて手順 1 ~ 4 の繰り返し
6. 変更したアプリケーションの出荷
7. プロセスまたはプロシージャの運用とインストール
8. アプリケーションのサポートと保守

上記作業は順番に行いますが、問題が発生した場合は、一部のステップを繰り返さなければならないこともあります。

この章では、最初の作業、つまりポーティングの必要性の調査について説明します。残りの作業については、次の章以降でそれぞれ説明します。この章では、アプリケーションのポーティング計画を作成する際に開発者が考慮しなければならない項目および開発者の責任について説明します。特に、OpenVMS Alpha 環境から OpenVMS I64 環境へのポーティングを想定して説明します。また、開発者がポーティングの必要性を評価し、プランを作成し、ソフトウェアのポーティングを開始するのに必要な情報を提供します。

付録 A には、計画プロセスとポーティング・プロセスで使用するチェックリストがあります。

---

### 3.1 概要

OpenVMS Alpha で現在動作しているほとんどのアプリケーションは、ほとんど変更せずに OpenVMS I64 に簡単に移植できます。OpenVMS の Alpha バージョンと I64 バージョンは、1 つのソース・コードから開発されているため、開発者はそれぞれのバージョンでソース・コードを個別に変更しなくても、ハードウェアに依存しない機能を両方のバージョンに盛り込むことができます。その結果、確認テストに必要な時

間を大幅に短縮でき、両方の OpenVMS プラットフォームで重要なアプリケーションを確実に提供することができます。

HP は、OpenVMS Alpha の推奨バージョンで現在「正常に動作」しているアプリケーションが OpenVMS I64 システムでも正常に動作するように、ソース・コードの上位互換性を保証し、維持していくための厳密なポリシーを設定して管理しています。公開されているシステム・サービスやライブラリ・インタフェースを使用しているアプリケーションであれば、ほとんどのケースで、ソース・コードをまったく変更せずに最新バージョンの OpenVMS I64 に移行できます。しかし、OpenVMS Alpha で提供されている公開済みのシステム・サービスやライブラリ・インタフェースの中には、OpenVMS I64 で使用できなかったり、異なる動作をするものがあります。さらに、リンカ・ユーティリティは、OpenVMS Alpha でサポートされているすべての修飾子をサポートするわけではありません。これらの例外の詳細については、第 4 章と第 5 章を参照してください。

OpenVMS I64 は、従来の OpenVMS システムとほとんど同じ見た目と使い勝手で使用できます。新しいアーキテクチャに対応するために、わずかな変更は必要でしたが、OpenVMS の基本構造および機能は変更されていません。

---

## 3.2 アプリケーションの評価

ポーティング作業の第 1 ステップでは、アプリケーションを OpenVMS I64 へ移植するために必要となる手順について、評価および確認を行います。評価作業の段階で以下の質問に回答することにより、ポーティング計画を作成することができます。

- どのような方法でアプリケーションのポーティングを行うか?
- ポーティングに必要な時間、作業量、コストはどの程度か?
- HP のサービス部門から提供されるサポートが必要か?

評価プロセスは次の 4 つの手順に分かれます。

1. ポーティングが必要なアプリケーションの洗い出し
2. 他のソフトウェアへの依存関係の洗い出し
3. ポーティング方法の選択
4. 運用タスクの分析

これらのステップを完了すると、効果的なポーティング計画を作成するのに必要な情報が得られます。最初の基本的な評価を行うための各ステップについては、3.2.1 ~ 3.2.4 項で説明します。その後、各モジュール、イメージ、およびすべての依存関係の詳細な評価を行うことができます。



この後の各項で説明する評価ステップを完了した後、その結果を、ターゲット・プラットフォームでサポートされている製品、プロシージャ、および機能と比較してください。スケジュール上の問題点、サポートされない機能、矛盾する手順、サポートの問題などがないか確認します。ポーティング計画を完成させるには、すべてのコストと時間の影響も盛り込む必要があります。

### 3.2.1 ポーティング方法の選択

以下の各項で説明する評価プロセスを完了したら、実際にアプリケーションと開発環境のポーティングを行います。ソース・モジュールの他に、ビルド・プロシージャやテスト・ツール、場合によってはアプリケーションのデータ構造など、その他のコンポーネントについてもポーティングが必要になる場合があります。アプリケーションの各要素に最適なポーティング方法を選択する必要があります。一般に問題となるのは、アプリケーションをソースから再ビルドするのか、バイナリ・トランスレータを使用するのかという点です。この判断を下すには、アプリケーションの各要素に対して使用可能な方法と、それらに必要な作業量を把握しておく必要があります。これらの疑問に答えるには、図 3-1 に示す一連の質問に答え、各作業を行う必要があります。

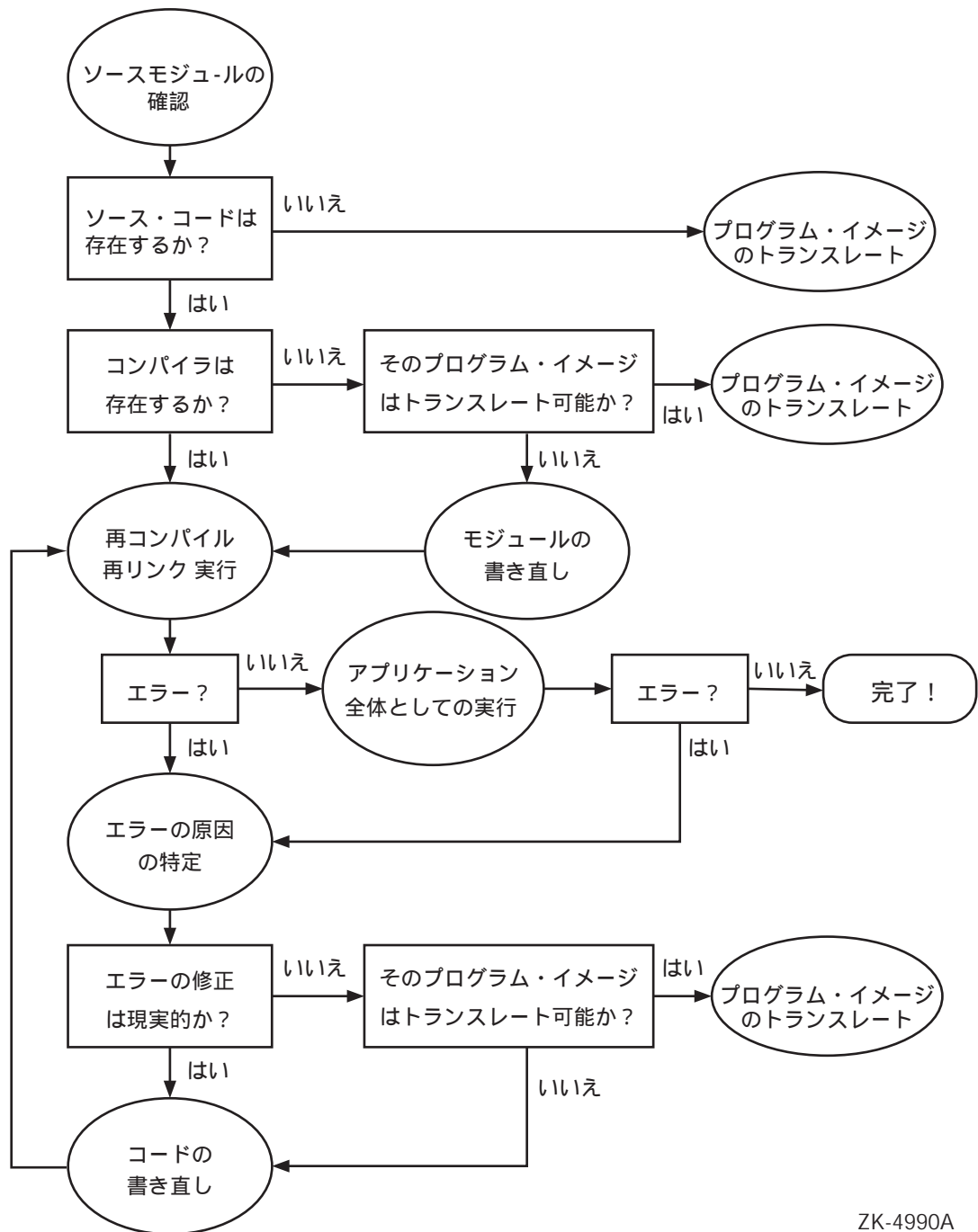
ほとんどのアプリケーションは、再コンパイルと再リンクを実行することによりポーティングが可能です。ユーザ・モードのみで動作し、標準的な高級言語で書かれているアプリケーションは、ほとんどの場合、このカテゴリに分類されます。

### 3.2.2 ポーティングが必要なアプリケーションの洗い出し

ポーティングのためのアプリケーション評価の最初のステップでは、ポーティングが必要となる対象を正確に洗い出します。アプリケーションだけでなく、アプリケーションを正常に実行するのに必要なあらゆる要素を洗い出す必要があります。アプリケーションの評価を開始するには、まず、以下の要素の場所を確認します。

- メイン・プログラムのソース・モジュール
- 共有イメージ
- オブジェクト・モジュール
- ライブラリ (オブジェクト・モジュール、共有イメージ、テキスト、またはマクロ)
- データ・ファイルとデータベース
- メッセージ・ファイル
- スクリプトとプロシージャ (ビルド・ファイル)
- アプリケーション・ドキュメント (用意されている場合)

図 3-1 アプリケーションのポータリング



ZK-4990A

第2章で説明した OpenVMS Alpha と OpenVMS I64 の相違点を考慮し、実際にポータリングを開始する前に、アプリケーション・コンポーネントに対して必要となる変更について把握します。さらに、第7章で説明する CPU の違いによる開発上の問題点も考慮してください。

これらの変更に必要な時間とコストを評価します。OpenVMS Alpha アーキテクチャに特に依存するコードは、変更する必要があります。たとえば、アプリケーションに以下のようなコードが含まれている場合は、変更が必要です。

- 特定のマシン・インストラクションを操作したり、レジスタの数や特定のレジスタの機能を前提にしているコード。
- OpenVMS Alpha の呼び出し規則の特定の要素に依存しているコード。

OpenVMS Alpha と OpenVMS I64 の呼び出し規則の相違点については、第 2.1.1 項を参照。

- 未公開のシステム・サービスやライブラリ・インタフェースに依存しているコード、または OpenVMS I64 で提供されていないが、提供されていても動作が異なる特定のシステム・インタフェースやライブラリ・インタフェースに依存しているコード。

特定のシステム・インタフェースおよびライブラリ・インタフェースが提供されているかどうか、および動作が異なるかどうかについては、第 4 章と第 5 章を参照。

- OpenVMS Alpha のオブジェクト・ファイルの形式、実行イメージ・ファイルの形式、またはデバッグ・シンボル・テーブルの形式に依存しているコード。
- 条件付きコード、あるいは OpenVMS VAX または OpenVMS Alpha システムで実行されることを前提にしており、それ以外のプラットフォームを取り扱うことができないようなロジックを含むコード。
- 以下に示すような、OpenVMS Alpha または OpenVMS VAX の内部データ構造に依存しているコード。
  - IEEE 標準に準拠しない浮動小数点形式 (IEEE 浮動小数点形式は OpenVMS I64 のデフォルトであり、VAX 浮動小数点形式が不要な場合はこの形式を使用すべきです。詳細については、第 3.2.4 項と第 4.8.4 項を参照)。
  - OpenVMS I64 のメカニズム・アレイ・データ構造は、OpenVMS Alpha と大きく異なります (第 4.8.3.3 項を参照)。

---

#### 注意

---

データ参照のパフォーマンスを向上するために、データは自然なアラインメントにすることをお勧めします。OpenVMS Alpha システムでも OpenVMS I64 システムでも、アラインメントされていないデータを参照すると、パフォーマンスが大幅に低下します。さらに、アラインメントされていない共有データがあると、プログラムを正常に実行できないこともあります。共有データは自然なアラインメントにする必要があります。共有データは、シングル・プロセスのスレッドの間、プロセスと AST の間、またはグローバル・セクション内の複数のプロセスの間で使用されることがあります。データ・アラインメントの詳細については、第 4.8.7 項および『OpenVMS Programming Concepts Manual』を参照してください。

---

## ポーティングが必要なアプリケーションの調査

### 3.2 アプリケーションの評価

- ターミナル・ドライバを参照する，(通常の) 呼び出しインタフェースを使用しないコード。

たとえば，JSB (jump to subroutine) マクロを使用しているコードは，呼び出しインタフェースを使用するように変更する必要があります。

- スレッドを実現するためにユーザが書いたコード。

たとえば，ダイレクト・スタック・スイッチングを実行するコード，コルーチンやタスキング・モデルをインプリメントするコード，プロシージャ呼び出しスタック・フレームの性質に依存するコードなど。詳細については，第 4.8.6 項を参照。

- 標準に準拠しなかったり，ドキュメントに記載されていないコーディング手法やインタフェースを使用しているコード。

さらに，ソース・コードのうち OpenVMS I64 ではコンパイラが提供されない言語で書かれたものは，書き直すか，変換する必要があります (サポートされるコンパイラの一覧については，第 6 章を参照)。OpenVMS Alpha の特定の言語機能は，OpenVMS I64 ではサポートされませんので注意してください。詳細については，『HP OpenVMS Version 8.2 リリース・ノート[翻訳版]』を参照してください。また，特権モードで実行されるアプリケーションも変更が必要になることがあります。

ポーティングの前にソース・モジュールで必要な変更の詳細については，第 4 章を参照してください。

#### 3.2.3 依存性の評価

次のステップでは，アプリケーションが依存しているソフトウェアおよび環境について評価します。

##### 3.2.3.1 ソフトウェアへの依存性

ソフトウェアへの依存性を考慮する場合，次のコンポーネントが関係します。

- ランタイム・ライブラリ
- HP 製レイヤード・ソフトウェア
- 他社製ソフトウェア
- ツール

これらの要素の多くは，すでに HP OpenVMS I64 に移植されています。たとえば，次のものはすでに移植済みです。

- 以下のコンポーネントも含めて，OpenVMS オペレーティング・システムにバンドルされている HP 製ソフトウェア
  - ランタイム・ライブラリ
  - その他の共有ライブラリ，たとえば，ユーティリティ・ルーチンや，その他のアプリケーション・ライブラリ・ルーチンを提供するライブラリ

- 以下のコンポーネントを含む HP 製レイヤード・ソフトウェア
  - コンパイラとコンパイラ・ランタイム・ライブラリ
  - データベース・マネージャ
  - ネットワーク環境

- 他社製品

現在、多くの他社製品が OpenVMS I64 で稼動しています。特定のアプリケーションが移行されているかどうかについては、アプリケーションを提供しているベンダにお問い合わせください。

---

重要

---

他社製のソフトウェアが提供されているかどうか、ポータリング・プロジェクトの最大の障害になることがあります。ターゲット・オペレーティング・システム用の他社製ソフトウェアの有無とその価格を確認する必要があります。これには、ビルド環境ツール、サードパーティ・ライブラリ、自動テスト・ツールなどが含まれます。OpenVMS で無料で提供されるツールについては、第 5 章を参照してください。

---

### 3.2.3.2 開発環境

アプリケーションが依存している開発環境について検討します。たとえば、オペレーティング・システムの構成、ハードウェア、開発ユーティリティ、ビルド・プロセス (CMS: Code Management System や MMS: Module Management System などの HP 製ツールを含む) などについて検討してください。

### 3.2.3.3 オペレーティング環境

オペレーティング環境に関して、次の問題点を検討します。

- システム属性

アプリケーションの実行と保守に必要とされるシステム。たとえば、必要なメモリ容量やディスク容量など。

- テスト・ツール

テストを行って、移植したアプリケーションが正常に動作することを確認し、パフォーマンスを評価する必要があります。新製品の開発、特に新しいシステムへのアプリケーションのポータリングの際には、リグレッション・テストがきわめて重要です。リグレッション・テストは、新しいバージョンのオペレーティング・システムおよびプラットフォームでソフトウェアをテストするのに役立ちます。リグレッション・テストは以下のいずれか 1 つ、またはその組み合わせによって構成できます。

- 考えられるすべてのコード・パスおよび機能を実行するために、開発チームが作成したソフトウェア・プログラム
- 一連の DCL コマンド・プロセス
- (ソフトウェアによる自動実行ではなく) 手動で実行する対話型テスト

- HP Digital Test Manager またはそれに相当するオープンソース・ツール

### 3.2.4 運用タスク

アプリケーションのポータリングと保守に必要な運用タスクの要件と責任を評価します。たとえば、次のタスクに関する検討が必要です。

- インストールの要件
- コンパイルの要件
  - アプリケーションで使用したコンパイラは、本バージョンの OpenVMS I64 でもサポートされますか?
  - アプリケーションのポータリングに先立って、OpenVMS Alpha システムで最新バージョンの OpenVMS Alpha コンパイラを使用してアプリケーションをコンパイルすることをお勧めします。この作業を行うことで、アプリケーションを OpenVMS I64 でコンパイルしたときに発生する可能性がある問題点を、あらかじめ検出できることがあります。新しいバージョンのコンパイラでは、既存のコンパイラ標準より厳密な解釈が適用されたり、新たに厳密な標準が適用されていることがあります。本バージョンの OpenVMS I64 でサポートされるコンパイラの詳細については、第 6 章を参照してください。
  - OpenVMS Alpha システムでコンパイルのテストを行い、問題を解決した後、アプリケーションを OpenVMS I64 システムに移植し、再度コンパイル、リンク、および確認作業を行う必要があります。これらの作業に関するプランを作成します。
  - VAX 浮動小数点データ・タイプを IEEE 浮動小数点データ・タイプに変更することが出来ますか?

一般に、OpenVMS I64 でも、アプリケーションで VAX 浮動小数点データ・タイプを使用できます。以前に生成されたバイナリ浮動小数点データにアクセスすることが必要な場合は、VAX 浮動小数点形式を使用します。OpenVMS I64 システムでは、VAX 浮動小数点形式はソフトウェアでインプリメントされます。HP 製コンパイラは、VAX 浮動小数点形式の値を IEEE 浮動小数点形式に変換して算術演算を実行するのに必要なコードを自動的に生成します。しかし、その後、値を VAX 浮動小数点形式に戻します。この追加変換のために、アプリケーションの実行時にオーバーヘッドが発生します。この理由から、できるだけアプリケーションを IEEE 浮動小数点形式に変更することを検討してください。

Alpha ハードウェアでは IEEE 浮動小数点形式がサポートされるため、OpenVMS Alpha で動作するアプリケーションは、パフォーマンスを犠牲にせずに、IEEE 浮動小数点形式を使用するように変更できます。このような変更をあらかじめ行っておくと、OpenVMS I64 へのアプリケーションの移行が容易になります。IEEE 浮動小数点形式は、Itanium ハードウェアでサポートされる唯

一の浮動小数点形式です。さらに、この形式を使用することで、アプリケーションのパフォーマンスも向上します。

OpenVMS Alpha システムで IEEE 修飾子を指定してアプリケーションをコンパイルすると、IEEE 浮動小数点形式の値を使用してアプリケーションの動作をテストできます (多くのコンパイラでは、コンパイラ・オプション/FLOAT=IEEE を指定します。I64 BASIC を使用している場合は、/REAL\_SIZE 修飾子を指定します)。その結果、望ましい結果が得られた場合は、同じ修飾子を使用して OpenVMS I64 システムで (さらに必要であれば Alpha でも) アプリケーションを単純にビルドすることができます。

詳細については、第 4.8.4 項およびホワイトペーパー『Intel® Itanium®における OpenVMS 浮動小数点演算について』を参照してください。このホワイトペーパーが入手できる Web サイトについては、「まえがき」の「関連資料」の項を参照してください。

- バックアップおよび復元機能
- オペレータ・インタフェースの見た目と使い勝手
- システム管理

---

### 3.3 HP から提供されるポーティング・リソース

どのような組織でも、ポーティング・プロセスのあらゆる段階に習熟しているわけではありません。そのため、HP はポーティング・プロセス全体にわたってお客様を支援するためのリソースを用意しています。これらのリソースには、ポーティング・ガイドから、ポーティングの全作業に対処するためのツールを用意したスタッフ・チームに至るまで、あらゆるものを提供できます。詳細については、HP の担当者にお問い合わせいただくか、以下のアドレスの Alpha Retain Trust サービスの Web サイトをご覧ください。

[http://www.hp.com/products1/evolution/alpha\\_retaintrust/services.html](http://www.hp.com/products1/evolution/alpha_retaintrust/services.html)

提供されるその他のリソースの例については、以下のサイトをご覧ください。

- **Developer and Solution Partner Program (DSPP)**

<http://www.hp.com/go/dspp>

- **Test Drive Program**

<http://www.testdrive.hp.com/>

- **HP Services**

<http://www.hp.com/go/services/>

Itanium アーキテクチャに関する HP の戦略が引き続き展開されていく間、セールス部門およびグローバル・サービス部門は、HP のお客様および ISV 各位の長期プランに与える影響を引き続き調査していきます。各ユーザに共通する一般的な状況および個々のお客様固有の状況を解決するために、数多くの定型サービスおよびカスタマイズ・サービスが用意され、ドキュメント化される予定です。HP は、アーキテクチャの過渡期に発生する重大な変更に対応するのに必要なツールとリソースを提供することで、お客様のニーズに最適なサービス製品を提供しています。

---

## 3.4 その他の検討事項

すべてのレイヤード・ソフトウェアおよびミドルウェア製品が、早期に OpenVMS I64 へ移植されるわけではありません。HP はソフトウェア・ベンダを支援するために、ポータリング・プロセスに役立つ情報、ハードウェア、サポート、およびツールを提供しています。各ソフトウェア・ベンダは、ソフトウェアの開発に必要なコンポーネントが開発スケジュールに間に合うように提供されるかどうかについて、各コンポーネントを提供する企業のポータリング・プロセスの進捗状況を確認してください。

HP は現在、コード・ベース、コマンド、およびプロセスに関して、大部分の開発者が新しい OpenVMS I64 プラットフォームへのポータリングを特に意識せずに透過的に行うことができるツール群を、既存の OpenVMS Alpha プラットフォームに実装することを計画しています。このツール群は、高品質な製品のリリースに伴うテストに代わるものではなく、製品のポータリングに必要となる時間を削減することを目的にしています。



---

## ソース・モジュールの移行

この章では、ポータリング作業においてすべてのアプリケーションに共通のステップについて説明します。その後、アプリケーションの再コンパイル前に変更の必要がある特定のタイプのコードやコーディング手法について説明します。

あらかじめコードを徹底的に分析し、移行プロセスのプランを作成しておけば、ソース・モジュールの移行の最終段階はかなり簡単です。多くのプログラムは、まったく変更せずに再コンパイルまたは変換することができます。直接再コンパイルまたは変換できないプログラムは、多くの場合、簡単な変更を行うだけで、I64 システムで動作させることができます。しかし、特定のタイプのコードは変更が必要です。

アプリケーションを移行するには、以下のステップが必要です。

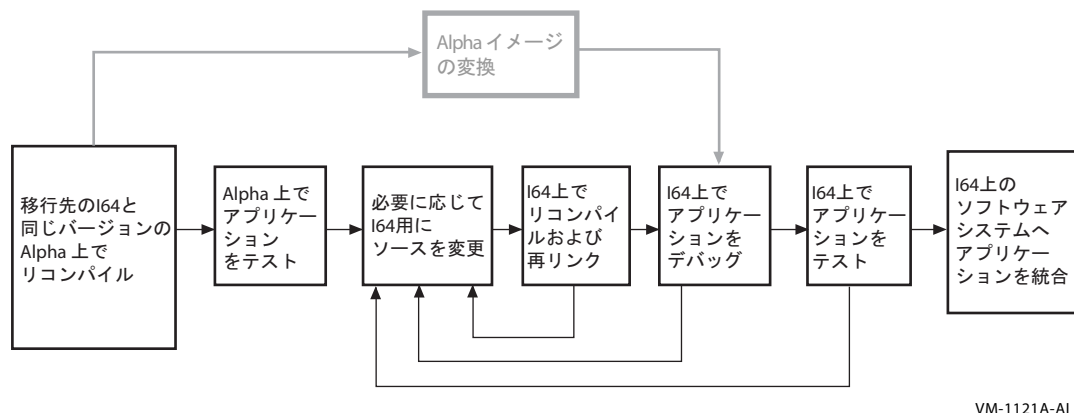
1. 移行環境を用意します。
2. Alpha システム上で最新バージョンの Alpha コンパイラを使用してアプリケーションをコンパイルします。
3. Alpha システム上でアプリケーションをテストし、移行評価のために基準となるデータを作成します。
4. I64 システム上でアプリケーションを再コンパイルおよび再リンクします。
5. 移行したアプリケーションをデバッグします。
6. 移行したアプリケーションをテストします。
7. 移行したアプリケーションをソフトウェア・システムに統合します。

移行環境の用意を除いた上記のステップを図 4-1 に示します。

ソース・ファイルが次のいずれかに該当する場合は、ソース・ファイルを変更しなければならない可能性があります。

- VAX Macro-32 または Alpha Macro-64 アセンブリ言語を使用している。
- 異なるプラットフォーム用のアプリケーションを共通のコードから作成するために条件付きステートメントを使用している。
- Alpha のオブジェクト・コードの形式など、Alpha アーキテクチャに依存している。
- 浮動小数点データ・タイプを使用している。

図 4-1 Alpha から I64 へのアプリケーションの移行



- スレッドを使用している。
- 自然な境界に配置されていないデータを使用している。
- OpenVMS Alpha の呼び出し規則に明示的に依存している。
- 特権インタフェースを使用しているか、または内部アクセス・モードで動作する。

詳細については、第 4.8 節を参照してください。

## 4.1 移行環境の用意

I64 でのネイティブな開発環境は、Alpha システムの開発環境と同等のものになります。OpenVMS Alpha で使い慣れているプログラム開発ツールの大部分は、I64 環境でも使用できます。2 つのアーキテクチャおよび呼び出し規則の相違点を考慮して、これらのツールには多少の変更が加えられています。しかし、このような変更の大部分は、ユーザが意識する必要はありません。

I64 への移行プロセスの重要な要素として、HP Services から提供されるサポートがあります。アプリケーションの変更、デバッグ、テストに役立つサポートが提供されます。詳細については、第 3.3 節を参照してください。

### 4.1.1 ハードウェア

移行に必要なハードウェアについて準備する場合、いくつかの点について考慮する必要があります。まず最初に、通常の Alpha 開発環境で必要なリソースについて検討します。

- CPU
- ディスク

- メモリ

I64 においては、本リリースでは次の構成が推奨されています。

- HP rx2600 Integrity サーバ (1 CPU または 2 CPU)
- 1 ~ 24 GB のメモリ
- 1 ~ 4 台の 36 GB のディスク
- DVD/CD-ROM ドライブ

詳細については、『HP OpenVMS Version 8.2 リリース・ノート[翻訳版]』を参照してください。

#### 4.1.2 ソフトウェア

効率のよい移行環境を構築するには、次の点を確認してください。

- 移行ツール

以下に示す、互換性のある移行ツール・セットが必要です。

- コンパイラ
- 変換ツール
  - HP OpenVMS Migration Software for Alpha to Integrity Servers
  - TIE
- ランタイム・ライブラリ
- システム・ライブラリ
- C プログラム用のインクルード・ファイル

- 論理名

論理名が Alpha バージョンあるいは I64 バージョンのツールおよびファイルをそれぞれ参照するように、整合性のある論理名定義を行う必要があります。

- コンパイル・プロシージャとリンク・プロシージャ

これらのプロシージャは、新しいツールおよび新しい環境に適合するように調整する必要があります。

- テスト・ツール

一連のテスト・ツールがまだ移植されていない場合は、Alpha のテスト・ツールを OpenVMS I64 に移植する必要があります。また、OpenVMS I64 固有の OpenVMS 呼び出し規則に準拠するための変更など、OpenVMS I64 版の特徴をテストするために設計されたテスト・ツールも必要です。

- 以下に示すような、ソースの保守とイメージのビルドのためのツール
  - CMS (Code Management System)

## ソース・モジュールの移行

### 4.1 移行環境の用意

- MMS (Module Management System)

Alpha ネイティブな開発ツール

Alpha 上で利用可能な標準の開発ツールは、I64 システムでもネイティブなツールとして利用できます。

変換ツール

ソフトウェア・トランスレータ HP OpenVMS Migration Software for Alpha to Integrity Servers は、Alpha システムと I64 システムの両方で動作します。変換イメージを動作させるために必要となる Translated Image Environment (TIE) は、OpenVMS I64 に含まれています (V8.2 では追加インストールが必要です)。このため、変換イメージの最終的なテストは、I64 システム上、あるいは I64 移行センターのどちらかで実行する必要があります。一般に、Alpha 用のイメージを I64 システムで動作するように変換する作業は簡単ですが、エラーなしで変換するためには、コードを多少変更しなくてはならない場合があります。

#### 4.1.2.1 HP OpenVMS Migration Software for Alpha to Integrity Servers と Translated Image Environment (TIE)

Alpha 用のユーザ・モード・イメージを OpenVMS I64 に移行するためのツールとして、静的なトランスレータと実行時サポート環境があります。

- HP OpenVMS Migration Software for Alpha to Integrity Servers は、Alpha 用のイメージを分析して、機能的に同等な I64 用のイメージに変換するユーティリティです。HP OpenVMS Migration Software for Alpha to Integrity Servers を使用すると、以下のことが可能になります。
  - Alpha 用のイメージが変換可能かどうかを判断する。
  - Alpha 用のイメージを I64 用のイメージに変換する。
  - イメージ内で OpenVMS I64 と互換性がない部分を見つけ、適切な場合には、これらの非互換部分をソース・ファイル中でどのように修正すればよいかについての情報を入手する。
  - 変換されたイメージの実行時性能を高める方法を見つける。
- Translated Image Environment (TIE) は I64 の共有イメージで、実行時に変換イメージをサポートします。TIE は、変換されたイメージに対して OpenVMS Alpha と同様の環境を提供し、ネイティブな I64 システムとのやり取りをすべて処理します。TIE には次のものが含まれています。
  - 以下の機能をサポートする Alpha 命令インタプリタ
    - (命令の不可分性を含め) Alpha システムで実行するのと同じように Alpha 命令を実行
    - 複雑な Alpha 命令をサブルーチンとしてサポート
  - Alpha 互換の例外ハンドラ
  - ネイティブ・コードと変換コード間の通信を可能とするジャケット・ルーチン

– エミュレートされた Alpha スタック

TIE は変換したイメージに対して自動的に起動されるため、明示的に呼び出す必要はありません。

HP OpenVMS Migration Software for Alpha to Integrity Servers は、可能な限り多くの Alpha コードを見つけて I64 コードに変換します。TIE は、I64 の命令に変換できない Alpha コードについては命令の解釈実行を行います。たとえば、以下のような命令がこれに該当します。

- HP OpenVMS Migration Software for Alpha to Integrity Servers が静的に見つけることができなかった命令
- VAX (F\_, G\_および D\_floating) の浮動小数点操作

命令の解釈実行は処理に時間がかかるため、HP OpenVMS Migration Software for Alpha to Integrity Servers は、実行時に解釈実行しなくてすむようにできるだけ多くの Alpha 用コードを変換しようと試みます。変換イメージは、TIE がどの程度 Alpha コードの解釈実行を行うかに依存して、対応するネイティブな Alpha 用イメージよりも動作が遅くなります。

I64 システムで Alpha イメージをそのまま、動的に解釈実行させることはできない点に注意してください。OpenVMS I64 で動作させる前に、HP OpenVMS Migration Software for Alpha to Integrity Servers を使ってイメージを変換しておく必要があります。

Alpha イメージを変換することにより、I64 ハードウェアでネイティブに動作するイメージが生成されます。変換された I64 イメージは単に Alpha イメージの解釈実行やエミュレートを行うものではなく、元の Alpha イメージ中の命令で実行していたのと同じ操作を行う I64 命令が含まれています。HP OpenVMS Migration Software for Alpha to Integrity Servers が変換できなかった命令を TIE が解釈実行できるように、I64 の EXE ファイルには元の Alpha 用のイメージもそのまま含まれています。

HP OpenVMS Migration Software for Alpha to Integrity Servers の分析機能は、変換ではなく再コンパイルしようと考えているプログラムの評価にも役立ちます。

#### 4.1.3 変換イメージとの共存

アプリケーション・コンポーネントは、バイナリ変換により、OpenVMS VAX および OpenVMS Alpha から OpenVMS I64 に移行させることができます。実行イメージと共有イメージは個別に変換できるため、単一のイメージ実行中にネイティブなイメージと変換されたイメージが混在するような環境を構築することも可能です(多くの場合そうなります)。

変換イメージでは、元の実装で使用していた呼び出し規則が使用されます。つまり、VAX から変換されたイメージの引数リストは、VAX プラットフォームで使用していたのと同じバイナリ形式になっています。OpenVMS I64 の呼び出し規則は、OpenVMS VAX および OpenVMS Alpha の呼び出し規則とは異なるため、ネイティブなイメージと変換されたイメージとの間で直接呼び出しはできません。このような呼び出しは、ある呼び出し規則から別の呼び出し規則に引数を変換するインタフェース・ルーチン (ジャケット・ルーチンと呼ぶ) によって処理されます。

ジャケット・ルーチンは TIE に含まれています。引数の変換の他、TIE は変換コードへの例外の伝達も行います。変換イメージ (メイン・プログラムまたは共有イメージ) が起動されると、TIE は追加の共有イメージとしてロードされます。イメージ・アクティベータは、ネイティブなイメージと変換されたイメージ間の参照を解決するために、必要に応じてジャケット・ルーチンへの呼び出しを介入させます。

ネイティブなコードと変換されたコードを一緒に使用するためには、以下の要件を満たしている必要があります。

- すべての関数ディスクリプタがシグネチャ・データを持っていること。

シグネチャデータは、ネイティブ形式と VAX または Alpha 形式の間で引数と戻り値を変換するために、ジャケット・ルーチンによって使用されます。ネイティブな I64 イメージは、アウトバウンド呼び出しとすべてのエントリ・ポイントの両方に対し、関数ディスクリプタを持っています。

- 関数変数の呼び出し (呼び出し先関数の識別情報が変数に格納されている) は、すべてライブラリ・ルーチン OTSSCALL\_PROC を経由して行うこと。

OTSSCALL\_PROC は、呼び出そうとしている関数がネイティブなものか変換されたものかを実行時に判断し、ネイティブなら直接呼び出し、変換されたものなら TIE 経由で呼び出します。

これらの要件は、/TIE 修飾子を使ってコードをコンパイルし、/NONATIVE\_ONLY 修飾子を使ってリンクすることで満たされます。/TIE は、Macro-32 コンパイラを含むほとんどの OpenVMS I64 コンパイラでサポートされています (C++ ではサポートされません)。どちらの修飾子も、コンパイルおよび LINK コマンドで明示的に指定する必要があります。これらの修飾子を指定しなかった場合のデフォルト値は、それぞれ /NOTIE および /NATIVE\_ONLY です。I64 アセンブラではシグネチャがサポートされていないため、I64 アセンブリ言語で記述されたコードは変換されたコードから直接呼び出すことはできません。I64 アセンブリ言語で記述されたコードから変換コードを呼び出すには、必ず明示的に OTSSCALL\_PROC を呼び出します。

変換イメージのサポートのため、性能は多少犠牲になります。関数変数の呼び出しは、すべて OTSSCALL\_PROC 経由で行われるため、ほとんどの場合約 10 の命令が必要になります。性能が要求されるコードを変換イメージと一緒に使用する場合は、必ず /TIE および /NONATIVE\_ONLY を使ってイメージを作成します。

/NOSYSSHR でリンクされた実行イメージと共有イメージには、他にも留意事項があります。通常、OTSSCALL\_PROC への参照は、共有イメージ LIBOTS.EXE によって解決され、特別な注意は必要ありません。しかし、イメージを/NOSYSSHR を指定して作成すると、共有イメージ・ライブラリの検索が省略され、代わりに STARLET.OLB 内のオブジェクト・モジュールから外部参照が解決されます。OTSSCALL\_PROC はシステム・シンボル・テーブル内に定義されたデータ・セルを参照するため、単に STARLET.OLB から取り込んだだけでは CTL\$GQ\_TIE\_SYMVECT の参照が未解決となります。

/NOSYSSHR でリンクされるほとんどのイメージは、外部イメージとのやり取りを避けるためにそのように作成されているため、変換されたコードの呼び出しで問題は発生しません。そのため、STARLET.OLB には、変換されたイメージの呼び出しをサポートしない OTSSCALL\_PROC のサブセット版が含まれています。この OTSSCALL\_PROC\_NATIVE という名前のモジュールはデフォルトでロードされ、/NOSYSSHR でリンクされたイメージは、デフォルトでは変換されたコードを呼び出せなくなります。

さらに、STARLET.OLB には完全版の OTSSCALL\_PROC モジュールも含まれています。ライブラリのシンボル・テーブルにはこのモジュールのエントリがないため、明示的な参照によってのみロードされます。万一/NOSYSSHR を指定してリンクしたイメージから変換された共有イメージを呼び出す必要がある場合には、完全版の OTSSCALL\_PROC を STARLET.OLB から明示的にインクルードして、イメージをシステム・シンボル・テーブルとリンクする必要があります。リンク・コマンド・ファイルには、以下の 2 つのオプションが必要です。

- 入力ファイル・リストに SY\$LIBRARY:STARLET.OLB  
/INCLUDE=OTSSCALL\_PROC を含める
- LINK コマンドに/SYSEXE 修飾子を含める

---

## 4.2 最新バージョンのコンパイラによる Alpha でのアプリケーションのコンパイル

ネイティブの I64 コンパイラでコードを再コンパイルする前に、まず、最新バージョンのコンパイラを使用して Alpha でコードをコンパイルすることをお勧めします。たとえば、アプリケーションが Fortran で書かれている場合は、Alpha で Fortran Version 7.5 (Fortran 90) を使用して再コンパイルしたアプリケーションが問題なく動作するか確認します。Fortran Version 7.5 は I64 に移植されているバージョンです。

最新バージョンのコンパイラを使用して OpenVMS Alpha でアプリケーションを再コンパイルすると、コンパイラのバージョンに起因する問題を検出できます。たとえば、新しいバージョンのコンパイラでは、以前は無視されていたプログラミング言語標準が新たに適用されていることがあり、その結果、アプリケーション・コードの潜在的な問題点が顕在化することがあります。また、新しいバージョンでは、それ以前

のバージョンがリリースされた後で標準に追加された内容が適用されていることもあります。このような問題を OpenVMS Alpha であらかじめ修正しておけば、I64 へのアプリケーションのポーティングが容易になります。

OpenVMS I64 のネイティブ・コンパイラの一覧については、第 5.1 節を参照してください。

---

### 4.3 移行するアプリケーションの Alpha 上での動作を確認するためのテスト

テストの第 1 ステップでは、Alpha アプリケーションに対してユーザが用意した一連のテスト・ツールを実行して、ポーティング後のアプリケーションをテストする際の比較基準となるデータを収集します。実行するのは、アプリケーションを I64 に移植する前でも後でもかまいません。その後、これらのテスト結果を、I64 システムで行った同様のテストの結果と比較します。第 4.6 節を参照してください。

---

### 4.4 I64 システムでの再コンパイルと再リンク

一般に、アプリケーションの移行では、コードの変更、コンパイル、リンク、デバッグを繰り返していきます。この過程で、開発ツールで検出されたすべての構文エラーとロジック・エラーを解決します。通常、構文エラーは簡単に修正できます。ロジック・エラーを解決するには、一般にコードの大幅な変更が必要になります。

新しいコンパイラ・スイッチやリンカ・スイッチの追加など、コンパイル・コマンドとリンク・コマンドはある程度変更しなければならない可能性があります。新しいリンカ修飾子の詳細については、第 5 章を参照してください。コンパイラ・スイッチの詳細については、第 6 章を参照してください。VAX MACROコードを移植する場合は、『OpenVMS MACRO-32 Porting and User's Guide』を参照してください。

---

### 4.5 移行したアプリケーションのデバッグ

アプリケーションを OpenVMS I64 に移行した後、デバッグが必要になるかもしれません。ここでは、アプリケーションのデバッグのために提供されている OpenVMS ツールについて説明します。

これらのツールの詳細については、第 5 章、第 6 章、および『OpenVMS デバッガ説明書』を参照してください。



#### 4.5.1 デバッグ

OpenVMS I64 オペレーティング・システムでは、以下のデバッグが提供されます。

- OpenVMS デバッグ

OpenVMS デバッグはシンボリック・デバッグです。プログラムで使用されているシンボル(変数名, ルーチン名, ラベル名など)によってプログラム・ロケーションを参照できます。プログラム・ロケーションを参照するときに、メモリ・アドレスやマシン・レジスタを指定する必要はありません。

OpenVMS Debugger は変換イメージのデバッグはサポートしていません。ただし、変換イメージを使用するネイティブ・アプリケーションのデバッグは可能です。

- XDelta デバッグ

XDelt デバッグはアドレス・ロケーション・デバッグです。このデバッグではアドレス・ロケーションでプログラム・ロケーションを参照しなければなりません。このデバッグは主に、特権プロセッサ・モードや割り込みレベルが高められた状態で動作するプログラムのデバッグに使用されます。Delta デバッグはまだ提供されていません。

シンボリック・デバッグであるシステムコード・デバッグは、常駐コードやデバイス・ドライバをどの IPL でもデバッグできるものです。

#### 4.5.2 システム・クラッシュの分析

OpenVMS では、システム・クラッシュを分析するために、System Dump Analyzer と Crash Log Utility Extractor の 2 つのツールが提供されます。

##### 4.5.2.1 System Dump Analyzer

OpenVMS I64 システムの System Dump Analyzer (SDA) ユーティリティは、OpenVMS Alpha システムで提供されているユーティリティとほとんど同じです。Crash Log Utility Extractor (CLUE) ユーティリティの機能にアクセスするための機能も含めて、多くのコマンド、修飾子、表示は OpenVMS Alpha のものと同じです。一部の表示は、OpenVMS I64 システム固有の情報も表示するように変更されています。

Alpha システムで SDA を使用するには、Alpha システムの OpenVMS 呼び出し規則について十分理解している必要があります。同様に、I64 システムで SDA を使用するには、I64 システムの OpenVMS 呼び出し規則について十分理解している必要があります。理解が十分でないと、スタックで発生したクラッシュのパターンを解読することはできません。詳細については、『OpenVMS Calling Standard』を参照してください。

#### 4.5.2.2 Crash Log Utility Extractor

Crash Log Utility Extractor (CLUE) は、クラッシュ・ダンプの履歴と各クラッシュ・ダンプのキー・パラメータを記録し、キー情報を抽出して要約するためのツールです。クラッシュ・ダンプはシステム障害が発生するたびに上書きされ、最新の障害に関する情報しか残りませんが、サマリ・クラッシュ・ヒストリ・ファイルには障害ごとの個別のリスト・ファイルが記録されるため、システム障害の永久的な記録になります。

---

## 4.6 移行したアプリケーションのテスト

移行したアプリケーションのテストを行い、移行したバージョンと Alpha バージョンの機能を比較する必要があります。

テストの最初のステップで、Alpha アプリケーションに対してテスト・ツールを実行し、アプリケーションの基準値を設定します。第 4.3 節を参照してください。

アプリケーションが I64 システムで動作するようになったら、以下の 2 種類のテストを行います。

- Alpha バージョンのアプリケーションに対して使用した標準的なテスト
- アーキテクチャが変更されたために問題が発生していないか確認するための新しいテスト

### 4.6.1 I64 への Alpha テスト・ツールのポーティング

移植したアプリケーションでは、新しいアーキテクチャの使用に伴う変更が含まれているため、OpenVMS I64 に移行した後でアプリケーションをテストすることは特に重要です。アプリケーションの変更によってエラーが発生するだけでなく、新しい環境に移行したことで、Alpha バージョンに潜在していた問題点が顕在化することもあります。

次の手順で、移行したアプリケーションをテストします。

1. 移行の前に、そのアプリケーションの標準的データをすべて用意します。
2. アプリケーションとともに Alpha 用のテスト・ツールも移行します (I64 にまだテスト・ツールがない場合)。
3. I64 システムでテスト・ツールを検証します。
4. 移行したアプリケーションに対して、移行したテスト・ツールを実行します。

ここでは、リグレーション・テストとストレス・テストの両方が役立ちます。同期に関するプラットフォーム間の違いをテストするには、ストレス・テストが重要です。特に、複数の実行スレッドを使用するアプリケーションの場合は、ストレス・テストが重要になります。

#### 4.6.2 新しい I64 テスト

移行した一連のテスト・ツールによるテストは、移行したアプリケーションの機能の検証に大いに役立ちますが、移行に伴って発生する問題点を検証するためのテストも追加しなければなりません。以下のポイントに注意する必要があります。

- コンパイラの違い
- アーキテクチャの違い
- 異なる言語で作成されたモジュールなどの統合

#### 4.6.3 潜在的なバグの検出

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

たとえば、プログラムで一部の変数の初期化をしていない場合、Alpha システムでは問題にならなくても、I64 システムでは算術演算例外が発生することがあります。同様に、使用できるインストラクションやコンパイラの最適化方法が変更されたために、2つのアーキテクチャ間で移行したときに、問題が発生することもあります。隠れているバグを取り除くのは容易ではありません。アプリケーションを移植した後、実際の運用に入る前にプログラムをテストすることが必要です。

---

### 4.7 移行したアプリケーションのソフトウェア・システムへの統合

アプリケーションを移行した後、他のソフトウェアとの相互関係によって発生する問題や、移行によって発生した問題がないか確認します。

相互運用性に関する問題の原因としては、以下の原因が考えられます。

- OpenVMS Cluster 環境内の Alpha システムと I64 システムは、それぞれ別のシステム・ディスクを使用する必要があります。アプリケーションが適切なシステム・ディスクを参照しているかどうか、確認する必要があります。

- イメージ名

アーキテクチャが混在した環境では、アプリケーションが CPU に対応した適切なイメージを参照しているかどうか確認する必要があります。

- Alpha 環境でも I64 環境でもイメージの名前は同一です。

I64 システムで Alpha イメージを実行しようすると、以下のようなエラー・メッセージが表示されます。

```
$ run alpha_image.exe
%DCL-W-ACTIMAGE, error activating image alpha_image.exe
-CLI-E-IMGNAME, image file alpha_image.exe
-IMGACT-F-NOT_I64, image is not an HP OpenVMS Industry Standard 64 image
$
```

---

## 4.8 特定のタイプのコードの変更

特定のコーディング手法および特定のタイプのコードは変更する必要があります。変更の必要なコーディング手法およびコード・タイプとしては以下のようなものがあります。

- Alpha Macro-64 アセンブラ・コード  
このコードは他の言語で書き直す必要があります。
- Alpha または VAX システムで実行するための条件付きステートメントを含むコード  
このようなコードは、I64 で実行するための条件に変更する必要があります。
- Alpha アーキテクチャに依存する OpenVMS システム・サービスを使用しているコード
- Alpha アーキテクチャに依存するその他の要素を含むコード
- 浮動小数点データ・タイプを使用するコード
- コマンド定義ファイルを使用するコード
- スレッド、特にカスタム作成タスキングやスタック・スイッチングを使用しているコード
- 特権コード

### 4.8.1 条件付きコード

ここでは、I64 に移行するときに、OpenVMS コードを条件付けする方法について説明します。このコードは、Alpha と I64 の両方を対象にコンパイルされるか、または VAX, Alpha, および I64 を対象にコンパイルされます。この後の各項で参照する ALPHA というシンボルは新しいシンボルです。しかし、EVAX というシンボルも削除されていません、必ずしも EVAX を ALPHA に置き換えなければならないわけではなく、都合の良いときに置き換えるということでもかまいません。MACRO と BLISS で使用できるアーキテクチャ・シンボルは、VAX, EVAX, ALPHA, および IA64 です。

#### 4.8.1.1 MACRO のソース

Macro-32 ソース・ファイルの場合、アーキテクチャ・シンボルは ARCH\_DEFS.MAR に定義されており、これはコマンド・ラインに指定されるプレフィックス・ファイルです。Alpha では、ALPHA と EVAX は 1 に定義されており、VAX と IA64 は未定義です。I64 では、IA64 は 1 に定義されており、VAX、EVAX、および ALPHA は未定義です。

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

Alpha 固有のコードの場合:

```
.IF DF ALPHA
.
.
.
.ENDC
```

I64 固有のコードの場合:

```
.IF DF IA64
.
.
.
.ENDC
```

#### 4.8.1.2 BLISS のソース

BLISS ソース・ファイル (BLISS-32 または BLISS-64) の場合、VAX、EVAX、ALPHA、および IA64 マクロは ARCH\_DEFS.REQ に定義されています。Alpha では、EVAX と ALPHA は 1 に、VAX と IA64 は 0 に定義されています。I64 では、IA64 は 1 に、VAX、EVAX、および ALPHA は 0 に定義されています。BLISS の条件付きステートメントでこれらのシンボルを使用するには、ARCH\_DEFS.REQ が必要です。

---

#### 注意

---

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

---

ソース・コードに以下のステートメントを追加します。

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

Alpha システムと I64 システムの両方で実行するには、以下のステートメントをソース・ファイルに追加してコードを条件付けします。

## ソース・モジュールの移行

### 4.8 特定のタイプのコードの変更

Alpha 固有のコードの場合:

```
%if ALPHA %then
    .
    .
    .
%fi
```

I64 固有のコードの場合:

```
%if IA64 %then
    .
    .
    .
%fi
```

#### 4.8.1.3 C のソース

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

Alpha 固有のコードの場合:

```
#ifdef __alpha
    .
    .
    .
#endif
```

I64 固有のコードの場合:

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

#### 4.8.1.4 既存の条件付きコード

既存の条件付きコードは、変更が必要かどうか確認する必要があります。以下の BLISS のコードについて検討してみましょう。

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

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

IA64 アーキテクチャ固有のコードを追加する必要がある場合は、以下のケースを追加します。

```
%IF IA64 %THEN
    iii
    iii
%FI
```

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

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

## 4.8.2 Alpha アーキテクチャに依存しているシステム・サービス

特定のシステム・サービスは、OpenVMS Alpha ではアプリケーション内で問題なく動作しても、I64 へのポータリングがうまくいかないことがあります。以降の項では、このようなシステム・サービスと、それに代わる I64 でのシステム・サービスを説明します。

### 4.8.2.1 SYSS\$GOTO\_UNWIND

OpenVMS Alpha では、SYSS\$GOTO\_UNWIND システム・サービスは、プロシージャ起動ハンドルを 32 ビット・アドレスで受け取ります。このシステム・サービスを呼び出している箇所を 64 ビット・アドレス用の SYSS\$GOTO\_UNWIND\_64 に変更する必要があります。ソース・コードを変更して、プロシージャ起動ハンドルに 64 ビットの領域を割り当ててください。また、OpenVMS I64 の起動コンテキスト・ハンドルを返すライブラリ・ルーチンも、OpenVMS Alpha とは異なります。詳細については、『OpenVMS Calling Standard』を参照してください。

SYSS\$GOTO\_UNWIND は、プログラミング言語機能をサポートする際によく使用されます。多くの場合は、コンパイラや実行時ライブラリでの変更になりますが、SYSS\$GOTO\_UNWIND を直接使用している場合は、変更が必要です。

### 4.8.2.2 SYS\$LKWSET と SYS\$LKWSET\_64

SYS\$LKWSET および SYS\$LKWSET\_64 システム・サービスは変更されています。詳細については、第 4.8.9 項を参照してください。

### 4.8.3 Alpha アーキテクチャに依存するその他の機能を含むコード

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

#### 4.8.3.1 初期化済みのオーバーレイ・プログラム・セクション

初期化済みのオーバーレイ・プログラム・セクションの取り扱いは、Alpha と I64 とで異なります。OpenVMS Alpha システムでは、オーバーレイ・プログラム・セクションの異なる部分を複数のモジュールで初期化することができます。このような初期化は、OpenVMS I64 システムでは認められていません。この動作の変更の詳細については、第 5.3.1.2 項を参照してください。

#### 4.8.3.2 条件ハンドラでの SS\$\_HPARITH の使用

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

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

#### 4.8.3.3 メカニズム・アレイ・データ構造

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

#### 4.8.3.4 Alpha のオブジェクト・ファイル形式への依存

OpenVMS I64 システムで作成されるオブジェクト・ファイルの形式は Alpha の形式と異なるため、コードが Alpha のオブジェクト・ファイルのレイアウトに依存している場合は変更の必要があります。

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

<http://www.caldera.com/developers/gabi>

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



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

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

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

#### 4.8.4 浮動小数点データ・タイプを使用するコード

OpenVMS Alpha では、VAX 浮動小数点データ・タイプと IEEE 浮動小数点データ・タイプをハードウェアでサポートしています。OpenVMS I64 では、IEEE 浮動小数点データ・タイプをハードウェアで、VAX 浮動小数点データ・タイプをソフトウェアでサポートしています。

ほとんどの OpenVMS I64 のコンパイラでは、VAX 浮動小数点データ・タイプを生成できるように、/FLOAT=D\_FLOAT および/FLOAT=G\_FLOAT 修飾子が用意されています。これらの修飾子を指定しないと、IEEE 浮動小数点データ・タイプが使用されます。

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

OpenVMS Alpha で IEEE 修飾子を指定してアプリケーションをコンパイルすると、Alpha で IEEE 浮動小数点の値を使用してアプリケーションの動作をテストできます。その結果、適切な結果が得られた場合は、同じ修飾子を使用して、I64 システムでアプリケーションをビルドすることができます。

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

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

算術演算が実行されない場合 (VAX 浮動小数点データのフェッチの後にストア・インストラクションが続く場合) は、変換は行われません。このような状況は move 命令で処理されます。

VAX 形式と IEEE 形式には以下の相違点があるため、算術演算の結果が異なる場合が稀に発生します。

- 数値の表現
- 丸めの規則
- 例外の動作

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

OpenVMS Alpha と OpenVMS I64 の浮動小数点データ・タイプの相違点、およびこれらの相違点がポーティング後のアプリケーションに与える影響の詳細については、第 5 章およびホワイトペーパー『Intel® Itanium®における OpenVMS 浮動小数点演算について』を参照してください。このホワイトペーパーが入手できる Web サイトについては、「まえがき」の「関連資料」の項を参照してください。

---

#### 注意

---

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

---

#### 4.8.4.1 LIB\$WAIT に関する問題と解決策

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

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

OpenVMS I64 システムでは、このコードは S\_FLOATING を LIB\$WAIT に送ります。しかし、LIB\$WAIT は F\_FLOATING を想定しているため、FLTINV 例外を返します。

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

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

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

LIBWAITDEF は以下のシンボルをインクルードします。

- LIB\$K\_VAX\_F
- LIB\$K\_VAX\_D
- LIB\$K\_VAX\_G
- LIB\$K\_VAX\_H
- LIB\$K\_IEEE\_S
- LIB\$K\_IEEE\_T

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

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

#### 4.8.5 コマンド・テーブル宣言に関する注意

OpenVMS I64 へ移植するコードに正しくないコマンド・テーブル宣言があると、予想外の結果となる場合があります。一例として、以下のようなエラーがあります。アプリケーションでは、CLD ファイルからオブジェクト・モジュールを作成するのにコマンド定義ユーティリティが使用されます。このアプリケーションは CLISDCL\_PARSE を呼び出し、コマンド行を解析します。CLISDCL\_PARSE は次のようなエラーで失敗する場合があります。

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

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

たとえば、VAX あるいは Alpha アプリケーションの BLISS モジュールで、コマンド・テーブル (DFSCP\_CLD) が次のように間違っただけと宣言されていたとします。

```
EXTERNAL ROUTINE DFSCP_CLD
```

これは次のように変更すべきです。

```
EXTERNAL DFSCP_CLD
```

FORTTRAN のモジュールで次のような宣言があったとします。

```
EXTERNAL DFSCP_CLD
```

これは次のように変更すべきです。

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

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

```
int jams_master_cmd();
```

このコードは、次のように外部参照に変更すべきです。

```
extern void* jams_master_cmd;
```

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

#### 4.8.6 スレッドを使用するコード

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

オーバーフローの問題を軽減するために、OpenVMS I64 ではデフォルトのスタック・サイズが拡大されています。しかし、アプリケーションで特定のスタック・サイズを要求する場合、そのデフォルトの変更は機能しないため、より大きなスタックを要求するようにアプリケーションのソース・コードを変更しなければなりません。そ

のような場合は、まず、8 KB ページを 3 つ (24576 バイト) 追加してみることをお勧めします。

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

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

OpenVMS I64 8.2 では、DCL の THREADCP コマンドはサポートされません。OpenVMS I64 では、スレッド関係のイメージ・ヘッダ・フラグの状態のチェックおよび修正には、OpenVMS Alpha の THREADCP コマンドの代わりに、DCL コマンド SET IMAGE および SHOW IMAGE が使用できます。詳細は『OpenVMS DCL ディクショナリ』を参照してください。

THREADCP コマンドについては『Guide to the POSIX Threads Library』で説明しています。

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

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

#### 4.8.6.1 スレッド・ルーチン cma\_delay および cma\_time\_get\_expiration

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

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

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

このような警告メッセージの原因となるようなオブジェクト・ファイルがリンクされている場合、リンカはこのシンボルから未定義シンボル・メッセージを表示します (リンカが生成したイメージをこのあと実行すると、ルーチンの呼び出しで ACCVIO とともにフェールします)。

---

#### 注意

---

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

---

#### 4.8.7 自然な境界に配置されていないデータを含むコード

データ参照のパフォーマンスを最適なレベルに向上させるには、データを自然な境界に配置することをお勧めします。データが自然な境界に配置されていない場合、OpenVMS Alpha でも OpenVMS I64 でもパフォーマンスが大幅に低下します。

アドレスが、バイト数で表したデータ・サイズの整数倍になっている場合、そのデータは自然な境界に配置されています。たとえば、ロングワードは、4 の倍数のアドレスで自然な境界に配置され、クォードワードは 8 の倍数のアドレスで自然な境界に配置されます。構造体のすべてのメンバが自然な境界に配置されると、その構造体も自然なアラインメントになります。

自然な境界に配置できない場合もあるので、OpenVMS I64 システムでは、自然な境界に配置されていないデータを参照することによる影響を管理するための支援機能を提供しています。Alpha と I64 のコンパイラは、発生する可能性のある大部分のアラインメントの問題を自動的に修正し、修正できない問題にはフラグを付けます。

自然な境界に配置されていない共有データがあると、パフォーマンスが低下するだけでなく、プログラムが正常に実行されない原因にもなります。したがって、共有データは自然な境界に配置する必要があります。1 つのプロセスの中にあるスレッドの間、プロセスと AST の間、複数のプロセスで使われるグローバル・セクションにおいてデータは共有されます。

#### 問題の検出

自然な境界に配置されていないデータのインスタンスを検出するには、自然な境界に配置されていないデータへの参照をコンパイラがコンパイル時に報告する修飾子を使用します。この修飾子は I64 のほとんどのコンパイラで提供されています。

表 4-1 コンパイル時の参照を報告するためのコンパイラ・スイッチ

コンパイラ	スイッチ
BLISS	/CHECK=ALIGNMENT
C	/WARN=ENABLE=ALIGNMENT
Fortran	/WARNING=ALIGNMENT
HP Pascal	/USAGE=PERFORMANCE

自然な境界に配置されていないデータを実行時に検出する OpenVMS デバッガの修飾子など、他の支援機能は、将来のリリースで計画されています。

#### 問題の排除

以下の 1 つ以上の方法を使用すると、自然な境界に配置されていないデータに関する問題を回避することができます。

- 自然な境界に配置するオプションを指定してコンパイルします。コンパイラや言語仕様でこの機能が提供されていない場合は、自然な境界に配置されるようにデータを配置し直してください。データのアラインメントが維持されるように埋め込みデータを挿入する場合は、メモリ・サイズが増大するという不利な状況が発生します。大きな領域を必要とする変数を先に宣言するようにすると、メモリを節約すると同時に、データの自然なアラインメントを維持するのに役立ちます。
- データ構造内でデータを強制的に自然な境界に配置するための高級言語の命令を使用します。
- データ・アイテムをクォードワード境界 (8 バイト境界) に配置します。

#### 注意

自然な境界に配置するように変換されたソフトウェアは、同じ OpenVMS Cluster 環境内や、ネットワークを介して動作している他の変換されたソフトウェアとの間で、互換性の問題を起こすことがあります。次の例を参照してください。

- 既存のファイル形式には、自然な境界に配置されていないデータを含むレコードが指定されていることがあります。
- 変換されたイメージが自然な境界に配置されていないデータをネイティブ・イメージに渡したり、逆に自然な境界に配置されていないデータをネイティブ・イメージから渡されるのを想定したりする可能性があります。

このような場合は、アプリケーションのすべての部分が、同じタイプのデータ、つまり、自然な境界に配置されたデータか自然な境界に配置されていないデータのどちらか一方を想定するように変更する必要があります。

#### 4.8.8 OpenVMS Alpha の呼び出し規則に依存するコード

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

- フレームポインタ (FP) はありません。
- 複数のスタックが使用されます。
- 4 つのレジスタだけが呼び出し時に保存されます。
- 従来のレジスタ番号が変更されています。

詳細については、第 2 章を参照してください。

#### 4.8.9 特権コード

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

##### 4.8.9.1 SYS\$LKWSET と SYS\$LKWSET\_64 の使用

プログラム自身をメモリへロックするのに SYS\$LKWSET または SYS\$LKWSET\_64 を使用している場合には、これらを (OpenVMS Version 8.2 で導入された) LIB\$LOCK\_IMAGE という新しいライブラリ・ルーチンに置き換えることをお勧めします。同様に、SYS\$ULWSET および SYS\$ULWSET\_64 も LIB\$UNLOCK\_IMAGE という新しいライブラリ・ルーチンに置き換えます (ただし、VAX システムで実行されるプログラムは、これらの新しいライブラリ・ルーチンを使用することはできません)。

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

VAX システムでは通常、プログラムで明示的に参照されるコードとデータのロックだけが必要でした。Alpha では、プログラムで参照されるコード、データ、およびリンケージ・データをロックしなければなりません。I64 システムでは、コード、データ、ショート・データ、およびリンカで生成されたコードをロックする必要があります。ポーティングを容易にするため、また、ショート・データとリンカ生成データのアドレスはイメージ内で簡単に検出できないという理由から、Alpha と I64 の SYS\$LKWSET および SYS\$LKWSET\_64 システム・サービスには変更が加えられています。



OpenVMS Version 8.2 では、SYS\$LKWSET および SYS\$LKWSET\_64 システム・サービスは、渡された最初のアドレスを調べます。このアドレスがイメージの内部にある場合は、これらのサービスはイメージ全体をワーキング・セット内にロックしようとしています。正常終了状態コードが返されると、システムが PGFIPLHI バグ・チェックでクラッシュすることなく、プログラムは IPL を 2 より大きな値に上げることができます。

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

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

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

#### 4.8.9.2 SYS\$LCKPAG および SYS\$LCKPAG\_64 の使用

メモリでコードをロックするためにアプリケーションで SYS\$LCKPAG あるいは SYS\$LCKAPG\_64 を使用している場合、このサービスの使用を再検討してください。I64 では、このサービスはワーキング・セットのイメージ全体はロックしません。

コードが IPL を上げ、ページ・フォルトを発生することなく実行できるように、ワーキング・セットのイメージをロックしようとしているのかもしれませんが。第 4.8.9.1 項を参照してください。LIB\$LOCK\_IMAGE および LIB\$UNLOCK\_IMAGE ルーチンについては『OpenVMS RTL Library (LIB\$) Manual』を参照してください。

#### 4.8.9.3 ターミナル・ドライバ

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

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

[http://www.hp.com/products1/evolution/alpha\\_retaintrust/openvms/resources](http://www.hp.com/products1/evolution/alpha_retaintrust/openvms/resources)

#### 4.8.9.4 保護されたイメージ・セクション

保護されたイメージ・セクションは、通常、ユーザ作成のシステム・サービスを実装するための共有イメージで使用されます。

これらのイメージ・セクションは、ソフトウェアおよびハードウェアの機能によって保護し、特権のないアプリケーションがこれらのセクションの完全性を危うくすることがないようにします。VAX および Alpha と I64 のハードウェアのページ保護機能の違いにより、わずかな制限が追加になりますので、保護されたイメージに対して変更が必要になる場合があります。

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

このように VAX と Alpha では例外がありましたが、I64 では、すべての保護された書き込み可能イメージ・セクションは、ユーザ・モードでの書き込みから保護されます。保護されたイメージ・セクションに対するユーザ書き込みを許可する場合は、\$SETPRT/\$SETPRT\_64 を使ってページ保護を変更する必要があります。

---

## OpenVMS I64 開発環境

この章では、OpenVMS I64 開発環境について以下の項目を説明します。

- I64 のネイティブ・コンパイラ
- その他の開発ツール
- リンカ
- デバッガ
- Librarian コーティリティ

---

### 5.1 I64 のネイティブ・コンパイラ

OpenVMS I64 では、以下の言語に対して I64 のネイティブ・コンパイラが提供されています。

- BASIC
- BLISS (Freeware CD で提供)
- C++
- COBOL V2.8
- Fortran V8.0 (Fortran 90)
- HP C V6.5
- HP Pascal
- Java
- VAX Macro-32

Alpha の Macro-64 アセンブラ・コードを OpenVMS I64 でコンパイルし実行するための機能は提供されません。

I64 コンパイラでは、VAX 浮動小数点データ・タイプを使用するためのコマンド・ライン修飾子が提供されます。VAX Macro-32 コンパイラを除き、これらのコンパイラの詳細については、第 6 章を参照してください。

### 5.1.1 VAX MACRO-32 Compiler for OpenVMS I64

OpenVMS I64 向けの新しいプログラムの開発では、高級言語を使用することをお勧めします。既存のVAX MACROコードを OpenVMS I64 システム上で動作するマシン・コードに変換するための VAX Macro-32 Compiler for OpenVMS I64 を提供します。

大部分のVAX MACROコードは、まったく変更せずにコンパイルできます。例外は以下のとおりです。

- OpenVMS I64 の呼び出し規則に準拠していないルーチンを呼び出すプログラム
- Macro-32 以外の言語で書かれたルーチンを JSB インストラクションを使用して呼び出すプログラム

VAX MACROプログラムの OpenVMS I64 システムへのポーティングの詳細については、『OpenVMS MACRO-32 Porting and User's Guide』を参照してください。

---

## 5.2 その他の開発ツール

I64 のネイティブ・アプリケーションの開発、デバッグ、および導入のために、コンパイラの他に複数のツールが提供されています。これらのツールについて、表 5-1 で概要を説明します。

表 5-1 OpenVMS の開発ツール

ツール	説明
OpenVMS リンカ	OpenVMS リンカは I64 のオブジェクト・ファイルから I64 のイメージを生成します。OpenVMS リンカの詳細については、第 5.3 節を参照してください。
OpenVMS デバッガ	OpenVMS I64 で動作する OpenVMS デバッガには、現在の OpenVMS Alpha デバッガと同じコマンド・インタフェースが用意されています。OpenVMS Alpha システムで提供されているグラフィカル・インタフェースは、今後のリリースで提供される予定です。OpenVMS I64 で提供される OpenVMS デバッガの詳細については、第 5.4 節を参照してください。
XDelta デバッガ	XDelta デバッガは、特権プロセッサ・モードや引き上げられた割り込み優先順位レベル (IPL) で動作するプログラムのデバッグ用に使用されるアドレス・ロケーション・デバッガです。XDelta デバッガは本リリースで提供されますが、Delta デバッガはまだ提供されていません。
OpenVMS Librarian ユーティリティ	OpenVMS Librarian ユーティリティは I64 ライブラリを作成します。
OpenVMS Message ユーティリティ	OpenVMS Message ユーティリティを使用すると、OpenVMS のシステム・メッセージに独自のメッセージを追加することができます。

(次ページに続く)

表 5-1 (続き) OpenVMS の開発ツール

ツール	説明
ANALYZE/IMAGE	Analyze/Image ユーティリティは I64 イメージを分析できます。
ANALYZE/OBJECT	Analyze/Object ユーティリティは I64 オブジェクトを分析できます。
DECset	DECset はさまざまな開発ツールを盛り込んだパッケージであり、LSE (Language Sensitive Editor)、DTM (Digital Test Manager)、CMS (Code Management System)、および MMS (Module Management System) が含まれています。この他に DECset には、PCA (Performance and Coverage Analyzer) と SCA (Source Code Analyzer) の 2 つのツールがありますが、これらのツールは将来のリリリースで提供される予定です。
Command Definition ユーティリティ	Command Definition ユーティリティ (CDU) を使うと、DCL コマンドに似た構文を持つコマンドを作成できます。
System Dump Analyzer (SDA)	SDA は拡張され、OpenVMS I64 システム固有の情報も表示するようになりました。
Crash Log Utility Extractor (CLUE)	CLUE は、クラッシュ・ダンプの履歴と各クラッシュ・ダンプのキー・パラメータを記録し、キー情報を抽出して要約を表示するためのツールです。

### 5.2.1 Alpha コードの変換

HP OpenVMS Migration Software for Alpha to Integrity Servers は、Alpha 用のユーザ・コード・イメージを OpenVMS I64 システムで動作させるためのバイナリ・トランスレータです。このツールは、ソース・コードがすでに入手できなくなっていたり、I64 でコンパイラが利用できない場合や、他社製の製品が OpenVMS I64 にポーティングされていない場合などに便利です。なお、他社製品に対して使用する場合は、ソフトウェア所有者の許可が必要です。

HP OpenVMS Migration Software for Alpha to Integrity Servers は、DECmigrate に似た機能を提供します。DECmigrate は、OpenVMS VAX のイメージを OpenVMS Alpha システムで動作させるためのポーティングの際に使用されたバイナリ・トランスレータです。

詳細は、第 4.1.2.1 項を参照してください。

## 5.3 モジュールのリンク

リンカの目的は、イメージ、つまりバイナリ・コードとデータが格納されているファイルを作成することです。OpenVMS I64 に移植されているリンカは、OpenVMS VAX および Alpha のリンカと異なり、OpenVMS I64 オブジェクト・ファイルを受け付け、OpenVMS I64 イメージを作成します。OpenVMS VAX および Alpha システムと同様に、作成されるイメージの主なタイプは実行イメージです。このイメージは、DCL コマンド・ラインで RUN コマンドを発行することにより起動できます。

OpenVMS I64 リンカは共有イメージも作成します。共有イメージは、symbol\_vector オプションでエクスポートされるプロシージャおよびデータ (それらは実行イメージや他の共有イメージから参照可能) の集合です。共有イメージは、実行イメージまたは共有イメージを作成するリンク操作で、入力ファイルとして指定されます。

モジュールをリンクする際にリンカの入力ファイルとなるのは、通常、オブジェクト・ファイルです。オブジェクト・ファイルは、コンパイラやアセンブラなどの言語プロセッサによって作成されます。これらのコンパイラで作成されるオブジェクト・ファイルは、Intel Itanium アーキテクチャ固有のものであるので、OpenVMS I64 でのモジュールのリンクには、OpenVMS VAX/Alpha の場合と異なる特徴がいくつかあります。一般に、OpenVMS I64 のリンカ・インタフェースをはじめ、その他の機能 (たとえば、シンボルの解決、仮想メモリの割り当て、イメージの初期化) は、OpenVMS VAX および Alpha システムの場合と類似しています。ここでは、リンカの相違点の概要を示し、OpenVMS I64 システムでプログラムをリンクする前に確認の必要がある項目についても説明します。説明する内容は次のとおりです。

- I64 システムでのリンクを VAX および Alpha システムでのリンクと比較した場合の相違点 (第 5.3.1 項)
- I64 用に拡張されたリンカ・マップ・ファイル情報 (第 5.3.2 項)
- 新しいリンカ修飾子とオプション (第 5.3.3 項)
- リンカ・オプションの引数における大文字小文字の混在 (第 5.3.4 項)

これらの機能および検討事項の詳細については、『HP OpenVMS Version 8.2 新機能説明書』を参照してください。

### 5.3.1 OpenVMS I64 システムでリンクする場合の相違点

OpenVMS I64 システムでのリンクは OpenVMS Alpha システムでのリンクに類似していますが、いくつかの相違点があります。OpenVMS I64 リンカは、以下の修飾子あるいはオプションを無視します。

- /REPLACE
- /SECTION\_BINDING
- /HEADER
- /DEMAND\_ZERO=PER\_PAGE の per\_page キーワード (per\_page キーワードは、意味するところがわずかに異なりますが、将来のリリースではサポートされます。)
- DZRO\_MIN
- ISD\_MAX

以下の修飾子およびオプションは、OpenVMS I64 リンカでは使用できません。

- /SYSTEM

- /DEBUG=file\_spec でのファイル名キーワード
- CLUSTER=cluster\_name,base\_address ... で、ベース・アドレス・キーワードはヌルでなければなりません。
- BASE=
- UNIVERSAL=

以下の新しい修飾子が OpenVMS I64 リンカでサポートされます。

- /BASE\_ADDRESS
- /SEGMENT\_ATTRIBUTE
- /FP\_MODE
- /EXPORT\_SYMBOL\_VECTOR
- /PUBLISH\_GLOBAL\_SYMBOLS
- /FULL に対する GROUP\_SECTIONS および SECTIONS\_DETAILS キーワード

これらの修飾子およびオプションのいくつかについての詳細は、『HP OpenVMS Version 8.2 新機能説明書』を参照してください。

#### 5.3.1.1 CLUSTER オプションでのベース・アドレスの指定

VAX および Alpha では、CLUSTER オプションでのベース・アドレスの指定が可能です。VAX では共有イメージであってもベース・クラスタを含めることが可能ですが、Alpha ではメイン・イメージのみがそのようなクラスタを構成することが可能です。I64 では、CLUSTER オプションでのベース・アドレスの指定はどのイメージに対しても認められていません。

#### 5.3.1.2 OpenVMS I64 での初期化済みオーバーレイ・プログラム・セクションの取り扱い

Alpha および VAX システムでは、オーバーレイ・プログラム・セクションの一部に対する初期化が可能でした。同じ部分に対して複数回初期化を行うと、前のモジュールで行われた初期化が上書きされます。リンクされるイメージで、それぞれのバイトごとに、最後に実行された初期化の内容が、そのバイトの最終値として適用されます。

I64 システムの ELF (Executable and Linkable Format) オブジェクト言語は、セクションの一部を初期化するという Alpha および VAX オブジェクト言語の機能を実装していません。I64 システムで初期化を行うと、セクション全体が初期化されます。このセクションのその後の初期化は、ゼロ以外の部分が値で一致する場合のみ実行されます。

たとえば以下の条件の場合は、OpenVMS I64 システムと Alpha システムで異なる結果が発生します。

それぞれが2つのロングワードを宣言する2つのプログラム・セクション (ELF では単にセクションと呼ぶ) がオーバーレイされた場合。最初のプログラム・セクションが最初のロングワードを初期化し、2番目のプログラム・セクションは2番目のロングワードをゼロ以外の値で初期化します。

OpenVMS Alpha システムでは、リンカは、最初のロングワードと2番目のロングワードが初期化されたイメージ・セクションを作成します。VAX および Alpha オブジェクト言語は、セクションを初期化するために、セクション・サイズと TIR (Text Information Relocation) コマンドをリンカに与えます。

OpenVMS I64 システムでは、リンカは、コンパイラからの初期化データを含む事前に初期化されたイメージ・セクションを取得します。ELF には TIR コマンドがないため、リンカは初期化は行わず、また初期化についての情報を持っていません。その後リンカは、最後に処理されたセクションを使用して初期化のためのセグメントを生成します。つまり、イメージの最初のロングワードあるいは2番目のロングワードのどちらかがゼロ以外の値に初期化されるかどうかは、モジュールがリンクされた順番によります。I64 システムでは、オーバーレイされたセクションをリンカが読み取り、互換性 (初期化が同一であるか、または矛盾がないか) をチェックします。2つのオーバーレイ・セクションがゼロ以外の値で同一であれば、それらは互換性があります。初期化が一致していない場合は、リンカは次のようなエラーを出力します。

```
%ILINK-E-INVORINI, incompatible multiple initializations for overlaid section
section: <section name>
module: <module name for first overlaid section>
file: <file name for first overlaid section>
module: <module name for second overlaid section>
file: <file name for second overlaid section>
```

上記のメッセージでは、リンカは、ゼロ以外の初期化を行った最初のモジュール、およびそれと互換性のない初期化を行う最初のモジュールをリストします。なおこれは、互換性のないすべての初期化を含む完全なリストではありません。リンカが遭遇する最初のものにすぎません。

リンカ・マップのプログラム・セクション構文では、ゼロ以外の初期化を伴う各モジュールには“Initializing Contribution”というフラグが付けられます。この情報を使用して、すべての互換性のない初期化を識別し、解決してください。初期化済みのオーバーレイ・セクションの処理に関するさらに詳しい例と説明については、『HP OpenVMS Version 8.2 新機能説明書』を参照してください。

#### 5.3.1.3 ELF 共通シンボルをリンクする際の動作の違い

オブジェクト・モジュール内の ELF common シンボル (Alpha の relaxed ref/def シンボルに相当) を、同じシンボルを含む共有イメージと/SELECTIVE\_SEARCH 修飾子とともにリンクする場合、I64 リンカの動作は Alpha と異なります。Alpha では、リンカは、オブジェクト・モジュール側の relaxed ref/def シンボルから定義を取得します。詳細は『HP OpenVMS Version 8.2 新機能説明書』を参照してください。



### 5.3.2 拡張されたマップ・ファイル情報

リンカ・マップ・ファイル内の情報は、OpenVMS I64 システム用に拡張されています。

- Alpha システムで Object Module Synopsis というタイトルだったマップ・セクションは、I64 システムでは Object and Image Synopsis に変更され、拡張されたオブジェクト情報とイメージ情報が格納されています。
- Alpha システムで Image Section Synopsis というタイトルだったマップ・セクションは、I64 システムでは Cluster Synopsis と Image Segment Synopsis の2つのセクションに分割されました。情報はそれぞれ適切なセクションに移動されています。
- Program Section Synopsis の PIC 属性と NOPIC 属性は、I64 システムでは無効であるため、削除されました。
- Symbol Cross Reference セクションの外部シンボルの指定は変更されました。Alpha で使用されていたプレフィックスまたはサフィックスは、再配置可能 (R) および外部 (X) を意味する RX でした。しかし、リンカには、外部シンボルが再配置可能かどうかはわかりません。そのため、I64 システムでは、プレフィックスまたはサフィックスは X (外部) に変更されました。
- Symbols By Value セクションの Keys for Special Characters は変更または拡張されています。たとえば、UxWk キーによって示される UNIX スタイルの弱いシンボル (おそらく C++ でのみ使用される) が OpenVMS I64 に新たに追加されています。

この新しい情報を示すリンカ・マップの例については『HP OpenVMS Version 8.2 新機能説明書』を参照してください。

### 5.3.3 OpenVMS I64 用の新しいリンカ修飾子とオプション

OpenVMS I64 システムでのリンクをサポートするために、新しいリンカ・オプションと修飾子が追加されました。ここでは、これらの機能について説明します。

#### 5.3.3.1 新しい /BASE\_ADDRESS 修飾子

I64 システムで新しい修飾子 /BASE\_ADDRESS が提供されています。ベース・アドレスは、リンカが実行イメージに対して割り当てる起点となるアドレスです。この修飾子の目的は、ブート・プロセスなどの、OpenVMS イメージ・アクティベータで起動されないイメージに対して仮想アドレスを割り当てることです。OpenVMS イメージ・アクティベータは、リンカが割り当てたスタート・アドレスを無視します。この修飾子は、主にシステム開発者によって使用されます。

/BASE\_ADDRESS 修飾子は、OpenVMS I64 では不正となる CLUSTER=[base-address] オプションの置き換えにはなりません。詳細は第 5.3.1.1 項を参照してください。

この修飾子についての詳細は『HP OpenVMS Version 8.2 新機能説明書』を参照してください。

#### 5.3.3.2 新しい/SEGMENT\_ATTRIBUTE 修飾子

OpenVMS I64 リンカは、2つのキーワード SHORT\_DATA=WRITE および DYNAMIC\_SEGMENT = P0 あるいは P1 を指定できる新しい/SEGMENT\_ATTRIBUTE 修飾子を提供します。DYNAMIC\_SEGMENT キーワードは通常は必要ありません。詳細は『HP OpenVMS Version 8.2 新機能説明書』を参照してください。

SHORT\_DATA=WRITE キーワードにより、最大 65,535 バイトの未使用読み取り専用スペースを再要求して、読み取り専用と読み書き両用のショート・データ・セクションを統合した単一のセグメントを作ることができます。SHORT\_DATA=WRITE を設定すると、以前は読み取り専用だったデータ・セグメントに書き込みが禁止されないという危険があります。このため、この修飾子はセクションの必要サイズが通常のショート・データ・セクションの上限である 4 MB を超えてしまった場合のみ使用することをお勧めします。

#### 5.3.3.3 新しい/FP\_MODE 修飾子

OpenVMS I64 リンカは、main プログラムの転送アドレスを提供するモジュールが提供する浮動小数点モードを使用して、プログラムの初期浮動小数点モードを決定します。メイン転送アドレスを提供するモジュールが初期浮動小数点モードを提供しない場合のみ、/FP\_MODE 修飾子を使用して浮動小数点の初期モードを設定してください。/FP\_MODE 修飾子は、メイン転送モジュールが提供する浮動小数点の初期モードを上書きしません。OpenVMS I64 リンカは、浮動小数点モードの設定に次のキーワードを使用できます。

- D\_FLOAT, G\_FLOAT—VAX 浮動小数点モードを設定します。
- IEEE\_FLOAT[=ieee\_behavior]—IEEE 浮動小数点モードをデフォルトあるいは特定の動作に設定します。

OpenVMS I64 リンカは、次の ieee\_behavior キーワードを使用できます。

- FAST
- UNDERFLOW\_TO\_ZERO
- DENORM\_RESULTS (デフォルト)
- INEXACT

OpenVMS I64 リンカは、浮動小数点モード動作リテラルも使用できます。浮動小数点初期モードについての詳細は、『OpenVMS Calling Standard』を参照してください。

#### 5.3.3.4 新しい/EXPORT\_SYMBOL\_VECTOR および/PUBLISH\_GLOBAL\_SYMBOLS 修飾子

/EXPORT\_SYMBOL\_VECTOR および/PUBLISH\_GLOBAL\_SYMBOLS 修飾子は、SYMBOL\_VECTOR オプションでどのシンボルをエクスポートするかを知らないユーザが共有イメージを作成するのを支援するためにリンクに追加されています。UNIX からアプリケーションをポーティングしているがどのシンボルをエクスポートするか良くわからないような場合、あるいは、C++ でコーディングしていてマングル名が何になるかわからない場合に役に立ちます。

OpenVMS I64 リンカは、オブジェクト・モジュール中のすべてのグローバル・シンボルをシンボル・ベクタ・オプションへエクスポートするようにモジュールにマークを付ける、新しい/PUBLISH\_GLOBAL\_SYMBOLS 修飾子を提供します。さらに、OpenVMS I64 リンカは、シンボル・ベクタ・オプションをエクスポートすることを指定し、またその出力ファイル名を設定するための新しい/EXPORT\_SYMBOL\_VECTOR 修飾子も提供します。

どちらの修飾子も/SHAREABLE 修飾子を指定している場合のみ有効です。/EXPORT\_SYMBOL\_VECTOR はコマンド行でのみ使用可能です。/PUBLISH\_GLOBAL\_SYMBOLS はオプション・ファイルでも使用可能です。/PUBLISH\_GLOBAL\_SYMBOLS 修飾子がないのに/EXPORT\_SYMBOL\_VECTOR がある場合、リンクは警告を出します。

/EXPORT\_SYMBOL\_VECTOR がある場合、オプション・ファイルが書き込まれますがイメージ・ファイルは生成されません。生成されるオプション・ファイルは、開発者が GSMATCH 情報を書き加えて完成させる必要があります。

/PUBLISH\_GLOBAL\_SYMBOLS 修飾子は位置修飾子で、オブジェクト・ファイルとライブラリで使用されます。/INCLUDE および/SELECTIVE 修飾子と互換性があります。次に例を示します。

```
$ link/SHARE public/PUBLISH,implementation/EXPORT=public
$ link/SHARE plib/LIBRARY/PUBLISH/INCLUDE=public/EXPORT=public
```

/EXPORT\_SYMBOL\_VECTOR 修飾子は位置修飾子で、オプションの出力ファイル指定を受け付けます。ファイル名が指定されていない場合リンクは最後の入力ファイル名を使用し、デフォルトのファイル・タイプは.OPT です。

この修飾子についての詳細は『HP OpenVMS Version 8.2 新機能説明書』を参照してください。

#### 5.3.3.5 PSECT\_ATTRIBUTE オプションのための新しいアラインメント

PSECT\_ATTRIBUTE オプションは、アラインメント属性として整数 5, 6, 7, 8 を受け付けるようになりました。これらの整数は、2 のべき乗として示されるバイト・アラインメントを表します (たとえば、2 \*\* 6 は 64 バイト・アラインメントを表します)。2 \*\* 5 (つまり、32 バイト・アラインメントまたは 16 ワード・アラインメント) を表すために、キーワード HEXA (16 ワードの意味) が追加されました。

#### 5.3.3.6 /FULL 修飾子の新しいキーワード GROUP\_SECTIONS および SECTION\_DETAILS

OpenVMS I64 リンカは/FULL 修飾子で 2 つの新しいキーワードを使用できます。最初のキーワード GROUP\_SECTIONS は、マップで使用されているすべてのグループを出力します。現在、グループを利用できるコンパイラは C++ のみです。他の言語でこのキーワードを使用しても効果はありません。

/FULL=NOSECTION\_DETAILS が指定されている場合、OpenVMS I64 リンカは、長さゼロの Program Section Synopsis データをマップにリストしません。/FULL 修飾子を使用されている場合、デフォルトは/FULL=SECTION\_DETAILS となり、VAX、Alpha、および I64 のリンカ・マップは Program Section Synopsis にすべてのモジュールをリストします。

#### 5.3.4 リンカ・オプションの引数における大文字小文字の区別

OpenVMS I64 システムでは、コンパイラが発行する名前は大文字と小文字が混在する場合があります。オプション・ファイルで大文字と小文字が混在する名前を操作する必要がある場合のため (たとえばライブラリに/INCLUDE を指定していて、大文字と小文字が混在するモジュール名がある場合など)、リンカは、デフォルトの動作 (名前はすべて大文字) ではなく、大文字/小文字を区別して名前を処理するオプションをすでに持っています。次に示す CASE\_SENSITIVE オプションを使います。

```
CASE_SENSITIVE=YES
```

CASE\_SENSITIVE オプションが YES に設定されている場合、等号の右側のすべての文字 (オプション引数など) の大文字/小文字の違いは維持されます。つまり、これらの文字は変更なくそのまま使用されます。ファイル名、モジュール名、シンボル名、およびキーワードがこれに含まれます。オプション行全体を大文字にするというリンカのデフォルトの動作に戻すには、次のように CASE\_SENSITIVE オプションに NO キーワードを指定します。

```
CASE_SENSITIVE=NO
```

NO キーワードは大文字で指定しなければなりません。大文字でないとリンカが認識しません。

VAX および Alpha との互換性のため、このオプションは必要な場合のみ使用することをお勧めします。

詳細については『HP OpenVMS Version 8.2 新機能説明書』を参照してください。

## 5.4 OpenVMS I64 システムでのデバッグ機能

OpenVMS では、広範囲にわたるデバッグ機能を実行できる複数のデバッガが提供されます。

- OpenVMS デバッガ (単にデバッガとも呼びます)
- Delta (OpenVMS I64 では利用できません)
- XDelta
- ソース・コード・デバッガ (SCD) (OpenVMS I64 では利用できません)

OpenVMS 8.2 では、OpenVMS デバッガと XDelta デバッガが提供されます。以下の表は、これらのデバッガの機能を示しています。

カテゴリ	OpenVMS デバッガ	XDelta
オペレーティング・システムのデバッグ	不可	可能
アプリケーションのデバッグ	可能	不可
シンボリック	可能	不可
0 より大きな IPL	不可	可能
プロセス・コンテンツ	可能	不可
ユーザ・モード	可能	不可†
スーパーバイザ・モード	不可	不可†
エグゼクティブ・モード	不可	可能
カーネル・モード	不可	可能

†IPL 0, 1, および 2 のみ

以降の各項では、OpenVMS I64 システムで動作する OpenVMS デバッガおよび XDelta デバッガの機能について説明します。OpenVMS デバッガで解決された問題点、制限事項、および報告されている問題点の詳細については、『HP OpenVMS Version 8.2 リリース・ノート[翻訳版]』を参照してください。

### 5.4.1 OpenVMS デバッガ

OpenVMS I64 で提供されるデバッガは、OpenVMS VAX および Alpha のデバッガと異なります。OpenVMS I64 デバッガのコマンド・インタフェースやデバッグ機能は OpenVMS VAX/Alpha デバッガに類似していますが、いくつかの相違点があります。ここでは、OpenVMS VAX および Alpha システムのデバッガについて十分理解しているものとして、OpenVMS I64 デバッガの機能について説明します。

#### 5.4.1.1 アーキテクチャのサポート

OpenVMS I64 デバッガは以下のハードウェア・レジスタをサポートします。

- 汎用レジスタ R0 ~ R127
- 浮動小数点レジスタ F0 ~ F127
- 分岐レジスタ B0 ~ B7
- プレディケート・レジスタ P0 ~ P63 を表す 64 ビット値 PRED
- アプリケーション・レジスタ: AR16 (RSC), AR17 (BSP), AR18 (BSPSTORE), AR19 (RNAT), AR25 (CSD), AR26 (SSD), AR32 (CCV), AR36 (UNAT), AR64 (PFS), AR65 (LC), AR66 (EC)
- ハードウェア IP レジスタと, PSR レジスタの ri フィールドから合成されるプログラム・カウンタ PC
- その他のレジスタ: CFM (current frame marker), UM (user mask), PSP (previous stack pointer), IIPA (previously executed bundle address)

#### 5.4.1.2 言語のサポート

OpenVMS I64 デバッガは、以下の言語で書かれたプログラムをサポートします。

- BLISS
- C
- C++ (制限あり)
- COBOL (制限あり)
- Macro-32
- Fortran
- Intel® Assembler (IAS)

サポートされる言語を OpenVMS I64 でデバッグする場合、いくつかの問題点があります。これらの問題点およびその対処法については、『HP OpenVMS Version 8.2 リリース・ノート[翻訳版]』を参照してください。

本リリースでは、C++ と COBOL のサポートには制限があります。参照、クラス、クラス・メンバなどの基本的な C++ の機能はサポートされますが、一般に、C++ に対するデバッガのサポートは、C と共通のプログラミング構造に制限されています。

C++ のデータ宣言に対して、SHOW SYMBOL/ADDRESS コマンドと SHOW SYMBOL/TYPE コマンドは使用しないでください。これらの問題やその他の問題の具体例については、『HP OpenVMS Version 8.2 リリース・ノート[翻訳版]』を参照してください。

Macro-32 は OpenVMS I64 でのコンパイル言語です。Itanium アーキテクチャでのハードウェア・レジスタの使い方は、Alpha や VAX の場合と異なるため、IMACRO コンパイラは Alpha および VAX のレジスタへのソース・コードの参照を、Itanium

で互換性のあるレジスタ参照に変換しなければなりません。Macro-32 で使用されるレジスタ・マッピングの詳細については、『HP OpenVMS Version 8.2 リリース・ノート[翻訳版]』を参照してください。

#### 5.4.1.3 機能とコマンド

提供されている機能とコマンドは以下のとおりです。

- 画面モード: ソース表示, インストラクション表示, 出力表示
- インストラクションのデコーディング
- ブレークポイント
- ウォッチポイント (非静的)
- ステップ (すべてのフォーム)
- DEBUG/KEEP コマンド (保持デバッガ構成)
- RUN/DEBUG コマンド (通常のデバッガ構成)
- マルチスレッド・プログラム
- CALL コマンド
- 共有イメージ内のコードのシンボリック・デバッグ
- 考えられるすべての静的データ・ロケーションのシンボル化
- 浮動小数点レジスタ内の値は, IEEE T-floating 値として表示
- 浮動小数点変数の確認と代入
- EXAMINE/ASCII コマンド

#### 5.4.1.4 まだ移植されていない機能

以下の機能は OpenVMS I64 へのポーティングがまだ完了していません。

- DECwindows グラフィカル・ユーザ・インタフェース
- ヒープ・アナライザ
- 画面モード: レジスタ表示
- SHOW STACK コマンド

### 5.4.2 XDelta デバッガ

OpenVMS I64 システムの XDelta デバッガは, 基本的には OpenVMS Alpha システムの XDelta と同じ動作をしますが, いくつかの制限事項があります。ここでは, OpenVMS 用に追加された新機能と, OpenVMS I64 システムと OpenVMS Alpha システムの XDelta の相違点について説明します。

#### 5.4.2.1 OpenVMS I64 での XDelta の機能

OpenVMS I64 システム用に以下の新機能が追加されました。

- XDelta は以下の Itanium レジスタをサポートします。
  - 汎用レジスタ: R0 ~ R127
  - 浮動小数点レジスタ: FP0 ~ FP127
  - アプリケーション・レジスタ: AR0 ~ AR127
  - 分岐レジスタ: BR0 ~ BR7
  - 制御レジスタ: CR0 ~ CR63
  - その他のレジスタ: PC, PS, および CFM
  - Alpha レジスタのソフトウェアでのインプリメント
- ;D メモリ・ダンプ・コマンド
- ;T 割り込みスタック表示コマンド

本リリースでのこれらレジスタおよび機能の詳細については、『HP OpenVMS Version 8.2 新機能説明書』を参照してください。このドキュメントでは、;D および ;T コマンドについてと、一部の制限事項についても説明しています。

#### 5.4.2.2 OpenVMS I64 システムと OpenVMS Alpha システムの XDelta の相違点

ここでは、I64 システムと Alpha システムでの XDelta の動作の違いについて説明します。

実行中のシステムへの割り込み

I64 で実行中のシステムに割り込みをかけるには、システム・コンソールで Ctrl/P を押します。その場合、あらかじめ XDelta をロードしておく必要があります。Ctrl/P を押すと、システムは現在の PC および現在の IPL で停止します。Alpha システムの場合と異なり、IPL が 14 より低くなるのを待つための遅延は発生しません。

シンボル化の機能向上

X レジスタは、イメージやモジュールのベース・アドレスなど、頻繁に使用される値を保持するためにプログラマが使用します。ブレークポイントやその他のアドレス値を表示する場合、XDelta はこれらの値を、最も近い X レジスタ値を基準にした相対値として出力するようになりました。以前は、特定の値だけが X レジスタ値との相対値を出していました。

---

## 5.5 OpenVMS I64 Librarian ユーティリティ

OpenVMS I64 の Librarian ユーティリティは、OpenVMS Alpha の Librarian と同じ機能を提供しますが、一部の機能が変更されており、制限事項もあります。ここでは、I64 の Librarian に固有の機能について説明します。制限事項やその他の一時的な条件については、『HP OpenVMS Version 8.2 リリース・ノート[翻訳版]』を参照してください。



### 5.5.1 I64 Librarian を使用する場合の検討事項

DCL の LIBRARY コマンド (または Librarian LBR ルーチン) を使用すると、I64 (ELF) オブジェクト・ライブラリ、I64 (ELF) 共有イメージ・ライブラリ、マクロ・ライブラリ、ヘルプ・ライブラリ、テキスト・ライブラリなどのライブラリを作成できます。ライブラリ内のモジュールの保守や、ライブラリとそのモジュールに関する情報の表示が可能です。I64 の Librarian で Alpha および VAX のオブジェクトや共有イメージを作成したり、処理することはできません。I64 Librarian のアーキテクチャは Intel Itanium です。

I64 ライブラリ用のアーキテクチャ・スイッチはありません。以下の修飾子を指定した場合、Librarian は OpenVMS ELF オブジェクト・ライブラリおよびイメージ・ライブラリを取り扱います。

- /OBJECT — OpenVMS ELF オブジェクト・ライブラリを使用します (デフォルト)。
- /SHARE — OpenVMS ELF 共有イメージ・ライブラリを使用します。
- /CREATE — /OBJECT 修飾子あるいは /SHARE 修飾子のどちらを指定したかに応じて、オブジェクト・タイプまたは共有イメージ・タイプの OpenVMS ELF ライブラリを作成します。

/OBJECT 修飾子も /SHARE 修飾子も指定しなかった場合、作成されるデフォルトのライブラリ・タイプはオブジェクト・ライブラリです。

### 5.5.2 LBR\$ルーチンの変更点

I64 システムでは、LBR\$OPEN ルーチンに対して 2 つの新しいライブラリ・タイプが追加されました。

LBR\$C\_TYP\_ELFOBJ (9) — ELF オブジェクト・ライブラリを表します。

LBR\$C\_TYP\_ELFSHSTB (10) — ELF 共有イメージ・ライブラリを表します。

さらに、LBR\$OPEN ルーチンに対する以下のライブラリ・タイプは、I64 の Librarian ではサポートされません。これらのライブラリ・タイプを使用して、OpenVMS Alpha または VAX のオブジェクト・ライブラリと共有イメージ・ライブラリを作成したり、開いたりすることはできません。

LBR\$C\_TYP\_OBJ (1) — VAX オブジェクト・ライブラリを表します。

LBR\$C\_TYP\_SHSTB (5) — VAX 共有イメージ・ライブラリを表します。

LBR\$C\_TYP\_EOBJ (7) — Alpha オブジェクト・ライブラリを表します。

LBR\$C\_TYP\_ESHSTB (8) — Alpha 共有イメージ・ライブラリを表します。

### 5.5.3 UNIX スタイルの弱いシンボルを取り扱う I64 ライブラリ形式

Intel C++ コンパイラの要件により、I64 ライブラリの形式が拡張され、新しい UNIX スタイルの弱いシンボルを取り扱うようになりました。新しい UNIX スタイルの弱いシンボルのキー名に一致する複数のモジュールが、同じライブラリに存在できるようになりました。以前の動作と同様に、Librarian は OpenVMS スタイルの弱いシンボル定義を無視します。

UNIX スタイルの弱いシンボル定義は、OpenVMS での弱い転送アドレスと同じ方法で動作します。つまり、その定義は一時的です。より強力なバインディング・タイプの定義がリンク操作で検出されない場合は、一時的な定義が最終的な定義として指定されます。

#### 5.5.3.1 弱いシンボルに対する新しい ELF タイプ

2 種類の弱いシンボル定義を区別するために、新しい ELF (Executable and Linkable Format) タイプが用意されました。ABI バージョン 2 のモジュール (コンパイラが使用する最も一般的なバージョン) の場合は、以下のようになります。

- タイプ STB\_WEAK は、UNIX スタイルの弱いシンボルを表します (以前は、ABI バージョン 1 の ELF 形式の場合は OpenVMS スタイルの弱いシンボル定義でした)。
- タイプ STB\_VMS\_WEAK は、OpenVMS スタイルの弱いシンボル定義を表します。

Librarian は同じライブラリ内で、ELF ABI バージョン 1 およびバージョン 2 の両方のオブジェクト・ファイルおよびイメージ・ファイルの形式をサポートします。

---

#### 注意

新しいライブラリ形式 (バージョン 6.0) は、ELF オブジェクト・ライブラリおよび共有イメージ・ライブラリにだけ適用されます。その他のライブラリはバージョン 3.0 の形式のままです。現在定義されているライブラリ・サービス・インタフェースを介してライブラリを参照するアプリケーションの場合、動作が変化することはありません。

---

#### 5.5.3.2 バージョン 6.0 ライブラリのインデックスの形式

新しいバージョン 6.0 のライブラリを使用することをお勧めします。旧バージョン (バージョン 3.0) のライブラリは、Library Services で開くことができますが、変更することはできません。旧バージョンのライブラリはバージョン 4.0 に変換できますが、制限事項があります。詳細については、『HP OpenVMS Version 8.2 新機能説明書』を参照してください。

#### 5.5.3.3 新しいグループ・セクション・シンボル

グループという ELF エンティティ内に含まれるセクションの集まりにシンボルを関連付けることができます。これらのグループと、グループに関連付けられたシンボルは、新しい UNIX スタイルの弱いシンボル定義と同様の動作をします。つまり、これらは一時的な定義です。Librarian は現在、ライブラリのシンボル名インデックスで重複したシンボル定義を認めています。

#### 5.5.3.4 弱いシンボルとグループ・シンボルに関する現在のライブラリの制限事項

ライブラリ・シンボル・エントリは、それが定義されたモジュールに関連付けられます。OpenVMS I64 では、複数のモジュールが UNIX スタイルの弱いシンボルまたはグループ・シンボルを定義できます。そのため、I64 Librarian は定義しているモジュールの優先順位リストを管理しなければなりません。このリストがあれば、優先順位の高い関連付けが削除されても、優先順位が低い関連付けを使用できます。優先順位の規則の詳細については、『HP OpenVMS Version 8.2 リリース・ノート[翻訳版]』を参照してください。

現時点では、『HP OpenVMS Version 8.2 リリース・ノート[翻訳版]』に説明されている制限事項があるため、UNIX スタイルの弱いシンボルを使うモジュールを含むライブラリに対してだけ、挿入操作だけを実行するようにしてください。ライブラリ内のモジュールの削除や置換を行う場合は、ライブラリを再ビルドして、UNIX スタイルの弱い定義の関連付けが正しく維持されるようにする必要があります。



## アプリケーションのポータリングの準備

この章では、OpenVMS I64 向けに提供される主要なコンパイラに関して、ポータリング上の検討事項の概要を示します。表 6-1 は、I64 バージョンへのポータリングに使用された Alpha コンパイラのバージョンの対応関係を示しています。

表 6-1 Alpha コンパイラのバージョンと I64 コンパイラのバージョンの対応関係

コンパイラ	OpenVMS Alpha	OpenVMS I64	参照先
BASIC	V1.6	V1.6	第 6.2 節を参照
BLISS	V1.11-004	V1.12-067	第 6.3 節を参照
COBOL	V2.8	V2.8	第 6.4 節を参照
Fortran 77	—	提供されません <sup>1</sup>	第 6.5.2 項を参照
Fortran 90	V7.5	V8.0	第 6.5 節を参照
GNAT Pro Ada 95	他社製	—	第 6.1 節を参照
HP Ada 83	V3.5A	提供されません <sup>1</sup>	—
HP C	V6.5	V7.1	第 6.6 節を参照
HP C++	V6.5	7.1	—
HP Pascal	V5.9	V5.9	—
Java	1.4.2	1.4.2-1	第 6.8 節を参照
Macro-32	V4.1-18	T1.0-77	第 6.9 節を参照
Macro-64	V1.2	提供されません <sup>1</sup>	—

<sup>1</sup>コンパイラはプラットフォーム固有です (Alpha のみ)。

大部分の OpenVMS I64 コンパイラには、以下に示す共通の特徴があります。

- デフォルトでは 32 ビット・コードを生成します。64 ビット・プログラムをビルドするには、コンパイラ・オプションを指定する必要があります。
- デフォルトは IEEE 浮動小数点データ・タイプであり、VAX 浮動小数点データ・タイプではありません。

ホワイトペーパー『Intel® Itanium®アーキテクチャにおける OpenVMS 浮動小数点演算について』では、I64 で浮動小数点に関する問題にどのように対処しているかについて説明されています。このホワイトペーパーと、OpenVMS から Itanium アーキテクチャへの移行に関するその他のリソースが入手できる Web サイトについては、「まえがき」の「関連資料」の項を参照してください。

これらの新しいコンパイラで発生する問題を振り落とすために、OpenVMS I64 にポータリングされているバージョンのコンパイラを使って、まず OpenVMS Alpha 上でアプリケーションをコンパイルすることをお勧めします (新しいバージョンのコンパイラでは、既存のコンパイラ標準より厳密な解釈が適用されたり、新しい厳密な標準が適用されていることがあります)。新しいコンパイラを使って OpenVMS Alpha システム上でアプリケーションをコンパイルし、リンクし、実行してエラーが発生しなければ、そのアプリケーションは OpenVMS I64 へのポータリングが可能です。

---

## 6.1 ADA

OpenVMS I64 Version 8.2 では ADA 95 が使用できます。ADA 83 は、I64 ではサポートされません。

---

## 6.2 BASIC

Alpha と I64 では同じ BASIC がサポートされます。OpenVMS I64 上で BASIC を使用する場合は注意事項については、BASIC のリリース・ノートを参照してください。

OpenVMS Alpha の BASIC コンパイラのデフォルトは/REAL\_SIZE=SINGLE (VAX F-float) で、OpenVMS I64 では/REAL\_SIZE=SFLOAT がデフォルトです。

BASIC コンパイラは/IEEE\_MODE 修飾子をサポートしません。コンパイラと RTL は、例外処理とラウンディング処理に関しては I64 上でも Alpha 版と同じようにランタイム環境を設定します。

---

## 6.3 BLISS コンパイラ

ここでは、OpenVMS Alpha と OpenVMS I64 の BLISS コンパイラの相違点について説明します。

BLISS-32EN と BLISS-64EN は、OpenVMS Alpha システム用のコードを生成する OpenVMS Alpha のネイティブ・コンパイラです。

BLISS-32IN と BLISS-64IN は、OpenVMS I64 システム用のコードを生成する OpenVMS I64 のネイティブ・コンパイラです。

BLISS-32xxコンパイラは 32 ビット操作を実行します (つまり、BLISS の値はロングワードです)。デフォルトは 32 ビットです。この章では、これらのコンパイラを総称して「32 ビット・コンパイラ」と呼びます。

BLISS-64xxは64ビット操作を実行します(つまり、BLISSの値はクォードワードです)。デフォルトは64ビットです。この章では、これらのコンパイラを総称して「64ビット・コンパイラ」と呼びます。

コンパイラを起動するには、以下のコマンドを使用します。

プラットフォーム	コンパイラ	コマンド
Alpha	BLISS-32EN	BLISS/A32 または BLISS
Alpha	BLISS-64EN	BLISS/A64
I64	BLISS-32IN	BLISS/I32 または BLISS
I64	BLISS-64IN	BLISS/I64

### 6.3.1 BLISS ファイル・タイプとファイル位置のデフォルト

ここでは、BLISS コンパイラのデフォルトのファイル・タイプと、出力ファイルの位置について説明します。

#### ファイル・タイプ

OpenVMS コンパイラのオブジェクト・ファイルのデフォルトのファイル・タイプは.OBJ です。

ライブラリ・ファイルの出力ファイルのデフォルトのファイル・タイプは、BLISS-32EN と BLISS-32IN の場合は.L32、BLISS-64EN と BLISS-64IN の場合は.L64 です。これらのライブラリ・ファイルの間に互換性はありません。

BLISS-32EN の検索リストは以下のとおりです。

ソース・コード: .B32E , .B32 , .BLI  
リクワイア・ファイル: .R32E , .R32 , , REQ  
ライブラリ・ファイル: .L32E , .L32 , .LIB

BLISS-64EN の検索リストは以下のとおりです。

ソース・コード: .B64E , .B64 , .BLI  
リクワイア・ファイル: .R64E , .R64 , .REQ  
ライブラリ・ファイル: .L64E , .L64 , .LIB

BLISS-32IN の検索リストは以下のとおりです。

ソース・コード: .B32I , .B32 , .BLI  
リクワイア・ファイル: .R32I , .R32 , .REQ  
ライブラリ・ファイル: .L32I , .L32 , .LIB

BLISS-64IN の検索リストは以下のとおりです。

ソース・コード: .B64I , .B64 , .BLI

## アプリケーションのポーティングの準備

### 6.3 BLISS コンパイラ

リクワイア・ファイル: .R64I , .R64 , .REQ

ライブラリ・ファイル: .L64I , .L64 , .LIB

出力ファイルのデフォルトの場所

OpenVMS コンパイラの出力ファイルの場所は , Alpha で動作する場合も I64 で動作する場合も , 出力修飾子がコマンド・ラインのどこに指定されているかに応じて決まります。

Alpha の BLISS でも I64 の BLISS でも , 入力ファイル指定の後に /OBJECT , /LIST , /LIBRARY などの出力ファイル修飾子が指定されていて , 出力ファイル指定がない場合は , 出力ファイルのデフォルトは , 直前に指定されている入力ファイルのデバイス , ディレクトリ , およびファイル名に設定されます。以下の例を参照してください。

```
$ BLISS /A32 [FOO]BAR/OBJ      ! Puts BAR.OBJ in directory FOO
$ BLISS /I32 [FOO]BAR/OBJ      ! Puts BAR.OBJ in directory FOO
$
$ BLISS /A32 /OBJ [FOO]BAR      ! Puts BAR.OBJ in default directory
$ BLISS /I32 /OBJ [FOO]BAR      ! Puts BAR.OBJ in default directory
$
$ BLISS /A32 [FOO]BAR/OBJ=[]    ! Puts BAR.OBJ in default directory
$ BLISS /I32 [FOO]BAR/OBJ=[]    ! Puts BAR.OBJ in default directory
```

#### 6.3.2 提供されない Alpha BLISS の機能

ここでは , OpenVMS I64 BLISS ではサポートされていない Alpha BLISS の機能について説明します。

Alpha BLISS のマシン固有の組み込み関数

以下の Alpha BLISS のマシン固有の組み込み関数は , I64 ではサポートされなくなりました。

CMP\_STORE\_LONG (CMP\_SWAP\_LONG に変更)  
CMP\_STORE\_QUAD (CMP\_SWAP\_QUAD に変更)  
CMPBGE  
DRAINT  
RPCC  
TRAPB  
WRITE\_MBX  
ZAP  
ZAPNOT

CMP\_SWAP\_LONG と CMP\_SWAP\_QUAD の詳細については , 比較 (Compare) およびスワップ (Swap) 組み込み関数を参照してください。



#### Alpha BLISS PALcode 組み込み関数

以下の Alpha BLISS PALcode 組み込み関数は、I64 ではサポートされなくなりました。

CALL_PAL	PAL_MFPR_PCBB	PAL_MTPR_SIRR
PAL_BPT	PAL_MFPR_PRBR	PAL_MTPR_SSP
PAL_BUGCHK	PAL_MFPR_PTBR	PAL_MTPR_TBIA
PAL_CFLUSH	PAL_MFPR_SCBB	PAL_MTPR_TBIAP
PAL_CHME	PAL_MFPR_SISR	PAL_MTPR_TBIS
PAL_CHMK	PAL_MFPR_SSP	PAL_MTPR_TBISD
PAL_CHMS	PAL_MFPR_TBCHK	PAL_MTPR_TBISI
PAL_CHMU	PAL_MFPR_USP	PAL_MTPR_USP
PAL_DRAIN	PAL_MFPR_VPTB	PAL_MTPR_VPTB
PAL_HALT	PAL_MFPR_WHAMI	PAL_PROBER
PAL_GENTRAP	PAL_MTPR_ASTEN	PAL_PROBEW
PAL_IMB	PAL_MTPR_ASTSR	PAL_RD_PS
PAL_LDQP	PAL_MTPR_DATFX	PAL_READ_UNQ
PAL_MFPR_ASN	PAL_MTPR_ESP	PAL_RSCC
PAL_MFPR_ASTEN	PAL_MTPR_FEN	PAL_STQP
PAL_MFPR_ASTSR	PAL_MTPR_IPIR	PAL_SWPCTX
PAL_MFPR_ESP	PAL_MTPR_IPL	PAL_SWASTEN
PAL_MFPR_FEN	PAL_MTPR_MCES	PAL_WRITE_UNQ
PAL_MFPR_IPL	PAL_MTPR_PRBR	PAL_WR_PS_SW
PAL_MFPR_MCES	PAL_MTPR_SCBB	PAL_MTPR_PERFMON

PALCALL 組み込み関数用のマクロ定義が STARLET.REQ に含まれています。

OpenVMS はサポーティング・コードを提供しています。特権 CALL\_PAL はエグゼクティブ・ルーチン呼び出し、非特権 CALL\_PAL はシステム・サービス呼び出しします。

#### Alpha BLISS のレジスタ名

以下のレジスタは、I64 のデフォルトでは下記に示すような機能で使われているため、REGISTER、GLOBAL REGISTER、および EXTERNAL REGISTER での名前の指定や、LINKAGE 宣言に対するパラメータとしてはサポートされません。

R0	ゼロ・レジスタ
R1	グローバル・ポインタ
R2	揮発性および GEM スクラッチ・レジスタ
R12	スタック・ポインタ
R13	スレッド・ポインタ
R14-R16	揮発性および GEM スクラッチ・レジスタ
R17-R18	揮発性スクラッチ・レジスタ

#### INTERRUPT リンケージと EXCEPTION リンケージ

INTERRUPT リンケージと EXCEPTION リンケージはサポートされません。

#### BUILTIN Rn

BUILTIN キーワードに I64 のレジスタ名を指定することはできません。

### 6.3.3 OpenVMS I64 BLISS の機能

OpenVMS I64 BLISS では、OpenVMS I64 をサポートするのに必要な既存の Alpha BLISS の機能だけが提供されます。Alpha BLISS では、OpenVMS 以外のオペレーティング・システムの機能もサポートしていますが、I64 BLISS コンパイラでは、このような機能は提供されません。

OpenVMS I64 BLISS の組み込み関数

I64 の BLISS コンパイラでは、OpenVMS I64 が正常に動作するのに必要な組み込み関数だけがサポートされています。

共通の BLISS 組み込み関数 既存の共通の BLISS 組み込み関数で、サポートされている関数は以下のとおりです。

ABS	CH\$FIND_NOT_CH	CH\$WCHAR
ACTUALCOUNT	CH\$FIND_SUB	CH\$WCHAR_A
ACTUALPARAMETER	CH\$GEQ	MAX
ARGPTR	CH\$GTR	MAXA
BARRIER	CH\$LEQ	MAXU
CH\$ALLOCATION	CH\$LSS	MIN
CH\$a_RCHAR	CH\$MOVE	MINA
CH\$a_WCHAR	CH\$NEQ	MINU
CH\$COMPARE	CH\$PLUS	NULLPARAMETER
CH\$COPY	CH\$PTR	REF
CH\$DIFF	CH\$RCHAR	SETUNWIND
CH\$EQL	CH\$RCHAR_A	SIGN
CH\$FAIL	CH\$SIZE	SIGNAL
CH\$FILL	CH\$TRANSLATE	SIGNAL_STOP
CH\$FIND_CH	CH\$TRANSTABLE	

RETURNADDRESS 組み込み関数 新しい組み込み関数 RETURNADDRESS は、呼び出し元の関数を呼び出した関数の PC を返します。

この組み込み関数には引数がなく、形式は次のとおりです。

RETURNADDRESS()

マシン固有の組み込み関数 I64 BLISS コンパイラでは、Alpha BLISS の以下のマシン固有の組み込み関数がサポートされています。

BARRIER  
ESTABLISH  
REVERT  
  
ROT  
SLL  
SRA  
SRL  
UMULH  
  
AdaWI  
  
ADD\_ATOMIC\_LONG    AND\_ATOMIC\_LONG    OR\_ATOMIC\_LONG  
ADD\_ATOMIC\_QUAD    AND\_ATOMIC\_QUAD    OR\_ATOMIC\_QUAD

組み込み関数xxx\_ATOMIC\_xxxでは、オプションの入力引数 retry-count はサポートされなくなりました。詳細については、不可分な操作のための ADD, AND, OR 組み込み関数を参照してください。

TESTBITSSI TESTBITCC TESTBITCS  
TESTBITCCI TESTBITSS TESTBITSC

TESTBITxxインストラクションでは、オプションの入力引数 retry-count およびオプションの出力引数 success-flag はサポートされません。詳細については、不可分な操作のための TESTBITxxI および TESTBITxx 組み込み関数を参照してください。

ADDD	DIVD	MULD	SUBD	CMPD
ADDF	DIVF	MULF	SUBF	CMPF
ADDG	DIVG	MULG	SUBG	CMPG
ADDS	DIVS	MULS	SUBS	CMPS
ADDT	DIVT	MULT	SUBT	CMPT
CVTDF	CVTFD	CVTGD	CVTSF	CVTTD
CVTDG	CVTFG	CVTGF	CVTSI	CVTTG
CVTDI	CVTFI	CVTGI	CVTSL	CVTTI
CVTDL	CVTFL	CVTGL	CVTSQ	CVTTL
CVTDQ	CVTFQ	CVTGQ	CVTST	CVTTQ
CVTDT	CVTFS	CVTGT		CVTTS
CVTID	CVTLD	CVTQD		
CVTIF	CVTLF	CVTQF		
CVTIG	CVTLG	CVTQG		
CVTIS	CVTLS	CVTQS		
CVTIT	CVTLT	CVTQT		
CVTRDL	CVTRDQ			
CVTRFL	CVTRFQ			
CVTRGL	CVTRGQ			
CVTRSL	CVTRSQ			
CVTRTL	CVTRTQ			

#### 新しいマシン固有の組み込み関数

OpenVMS I64 BLISS に追加された多くの新しい組み込み関数では、オペレーティング・システムで利用できるシングル I64 インストラクションにアクセスする機能が提供されます。

シングル I64 インストラクションに対する組み込み関数 以下の一覧で大文字で示されている関数は、指定可能な新しい組み込み関数です。括弧内の小文字の名前は、実際に実行される I64 のインストラクションです。これらのインストラクションに対する引数 (および関連の BLISS 組み込み関数名) の詳細については、『Intel IA-64 Architecture Software Developer's Manual』を参照してください。

## アプリケーションのポーティングの準備

### 6.3 BLISS コンパイラ

BREAK	(break)	LOADRS	(loadrs)	RUM	(rum)
BREAK2	(break)*	PROBER	(probe.r)	SRLZD	(srlz.d)
FC	(fc)	PROBEW	(probe.w)	SRLZI	(srlz.i)
FLUSHRS	(flushrs)	PCTE	(ptc.e)	SSM	(ssm)
FWB	(fwb)	PCTG	(ptc.g)	SUM	(sum)
INVALAT	(invala)	PCTGA	(ptc.ga)	SYNCI	(sync.i)
ITCD	(itc.d)	PTCL	(ptc.l)	TAK	(tak)
ITCI	(itc.i)	PTRD	(ptr.d)	THASH	(thash)
ITRD	(itr.d)	PTRI	(ptr.i)	TPA	(tpa)
ITRI	(itr.i)	RSM	(rsm)	TTAG	(ttag)

---

#### 注意

---

BREAK2 組み込み関数には、2つのパラメータが必要です。最初のパラメータはコンパイル時リテラルでなければならず、BREAK インストラクションの21ビットの即値 (immediate value) を指定します。2番目のパラメータには任意の式を指定でき、その値が、BREAK インストラクションを実行する直前に IPF 汎用レジスタ R17 に移動されます。

---

プロセッサ・レジスタへのアクセス OpenVMS I64 BLISS コンパイラでは、IPF インプリメンテーションの多くのさまざまなプロセッサ・レジスタへの読み書きアクセスのための組み込み関数が提供されています。これらの組み込み関数は以下のとおりです。

- GETREG
- SETREG
- GETREGIND
- SETREGIND

これらの組み込み関数は mov.i インストラクションを実行します。このインストラクションの詳細については、『Intel IA-64 Architecture Software Developer's Manual』を参照してください。2つの GET 組み込み関数は、指定されたレジスタの値を返します。

レジスタを指定するには、特殊なエンコードの整数定数を使用します。この定数は Intel C ヘッド・ファイルに定義されています。このファイルの内容については、『Intel IA-64 Architecture Software Developer's Manual』を参照してください。

PALcode 組み込み関数 以下の Alpha BLISS PALcode 組み込み関数は I64 でもサポートされます。

PAL_INSQHIL	PAL_REMQHIL
PAL_INSQHILR	PAL_REMQHILR
PAL_INSQHIQ	PAL_REMQHIQ
PAL_INSQHIQR	PAL_REMQHIQR
PAL_INSQTIL	PAL_REMQTIL
PAL_INSQTILR	PAL_REMQTILR
PAL_INSQTIQ	PAL_REMQTIQ
PAL_INSQTIQR	PAL_REMQTIQR
PAL_INSQUEL	PAL_REMQUEL
PAL_INSQUEL_D	PAL_REMQUEL_D
PAL_INSQUEQ	PAL_REMQUEQ
PAL_INSQUEQ_D	PAL_REMQUEQ_D

24 のキュー操作 PALcall はそれぞれ、OpenVMS の実行時ルーチン SYSSPAL\_xxxx の呼び出しとして、BLISS でインプリメントされています。

#### BLISCALLG

VAX の CALLG( .AP, ...) は、OpenVMS Alpha BLISS ではアセンブリ・ルーチン BLISCALLG(ARGPTR(), .RTN) に変更されています。このルーチンはもともと OpenVMS Alpha BLISS 向けに定義されたものですが、I64 アーキテクチャ向けに書き直され、OpenVMS I64 BLISS でサポートされています。

#### I64 のレジスタ

I64 の汎用レジスタは R3 ~ R11 および R19 ~ R31 であり、REGISTER、GLOBAL REGISTER、および EXTERNAL REGISTER に名前を指定でき、LINKAGE 宣言に対するパラメータとして指定することもできます。さらに、パラメータ・レジスタ R32 ~ R39 は、LINKAGE 宣言のみでパラメータとして指定できます。

現在のところ、I64 の汎用レジスタ R40 ~ R127 の名前を指定する機能をサポートする予定はありません。

REGISTER、GLOBAL REGISTER、EXTERNAL REGISTER、および LINKAGE 宣言によって、I64 の浮動小数点レジスタ、プレディケート・レジスタ、分岐レジスタ、およびアプリケーション・レジスタの名前を指定する機能は提供されていません。

特定のレジスタに対するユーザ要求を満たすことができない場合は、レジスタ競合のメッセージが出力されます。

#### ALPHA\_REGISTER\_MAPPING スイッチ

OpenVMS I64 BLISS では、新しいモジュール・レベル・スイッチ ALPHA\_REGISTER\_MAPPING が提供されています。

このスイッチは、MODULE 宣言または SWITCHES 宣言に指定できます。このスイッチを使用すると、この項で説明するように、Alpha のレジスタ番号を I64 のレジスタ番号に再マップできます。

REGISTER, GLOBAL REGISTER, および EXTERNAL REGISTER に指定されたレジスタ番号, またはリンケージ宣言の GLOBAL, PRESERVE, NOPRESERVE, または NOT USED に対するパラメータとして指定されたレジスタ番号 (0 ~ 31 の範囲) は, 以下の IMACRO マッピング・テーブルに従って再マッピングされます。

0 = GEM_TS_REG_K_R8	16 = GEM_TS_REG_K_R14
1 = GEM_TS_REG_K_R9	17 = GEM_TS_REG_K_R15
2 = GEM_TS_REG_K_R28	18 = GEM_TS_REG_K_R16
3 = GEM_TS_REG_K_R3	19 = GEM_TS_REG_K_R17
4 = GEM_TS_REG_K_R4	20 = GEM_TS_REG_K_R18
5 = GEM_TS_REG_K_R5	21 = GEM_TS_REG_K_R19
6 = GEM_TS_REG_K_R6	22 = GEM_TS_REG_K_R22
7 = GEM_TS_REG_K_R7	23 = GEM_TS_REG_K_R23
8 = GEM_TS_REG_K_R26	24 = GEM_TS_REG_K_R24
9 = GEM_TS_REG_K_R27	25 = GEM_TS_REG_K_R25
10 = GEM_TS_REG_K_R10	26 = GEM_TS_REG_K_R0
11 = GEM_TS_REG_K_R11	27 = GEM_TS_REG_K_R0
12 = GEM_TS_REG_K_R30	28 = GEM_TS_REG_K_R0
13 = GEM_TS_REG_K_R31	29 = GEM_TS_REG_K_R29
14 = GEM_TS_REG_K_R20	30 = GEM_TS_REG_K_R12
15 = GEM_TS_REG_K_R21	31 = GEM_TS_REG_K_R0

レジスタ番号 16 ~ 20, 26 ~ 28, 30 ~ 31 のマッピングは, OpenVMS I64 BLISS では, 無効な指定であると解釈されるレジスタに変換されます (Alpha BLISS のレジスタ名および I64 のレジスタを参照)。ALPHA\_REGISTER\_MAPPING が指定されているときに, これらのレジスタを含む宣言を行うと, 以下のようなエラーが発生します。

```
      r30 = 30,  
      ^  
.....  
%BLS64-W-TEXT, Alpha register 30 cannot be declared, invalid mapping to  
IPF register 12 at line number 9 in file ddd:[xxx]TESTALPHAREGMAP.BLI
```

ソース行にはレジスタ番号 30 が指定されていますが, エラー・テキストにはレジスタ 12 に問題があることが示されています。レジスタ 12 はレジスタ番号 30 が変換された結果であり, 指定するとエラーになります。

Alpha のレジスタ R16 ~ R21 がリンケージ I/O パラメータとして指定されている場合は, これらのレジスタに対して特殊なマッピング・セットが用意されています。

---

#### 注意

---

リンケージ I/O パラメータのみの場合は, R16 ~ R21 のマッピングは次のようになります。

```
16 = GEM_TS_REG_K_R32
17 = GEM_TS_REG_K_R33
18 = GEM_TS_REG_K_R34
19 = GEM_TS_REG_K_R35
20 = GEM_TS_REG_K_R36
21 = GEM_TS_REG_K_R37
```

---

#### ALPHA\_REGISTER\_MAPPING と "NOTUSED"

ALPHA\_REGISTER\_MAPPING を指定すると、IA64 のスクラッチ・レジスタにマッピングされリンク宣言で NOTUSED が指定された Alpha レジスタは、PRESERVE セットに配置されます。

これにより、このレジスタを NOTUSED と宣言しているルーチンを実行するときにレジスタが退避され、ルーチンから抜けるときに復元されます。

#### /ANNOTATIONS 修飾子

OpenVMS I64 BLISS コンパイラでは、新しいコンパイル修飾子/ANNOTATIONS がサポートされます。この修飾子は、コンパイル時にコンパイラが行っている (または行っていない) 最適化に関する情報をソース・リストに追加します。

この修飾子は、以下のキーワードにより、リストする情報を選択できます。

- ALL
- NONE
- CODE — マシン・コード・リストの注釈に使用されます。現在は NOP インストラクションにのみ、注釈が付けられます。
- DETAIL — 他のキーワードと組み合わせて使用し、詳細情報を提供します。

他のキーワードは GEM 最適化の情報を選択できます。

```
INLINING
LINKAGES
LOOP_TRANSFORMS
LOOP_UNROLLING
PREFETCHING
SHRINKWRAPPING
SOFTWARE_PIPELINING
TAIL_CALLS
TAIL_RECURSION
```

ALL と NONE を除き、他のすべてのキーワードは否定形にすることができます。修飾子自体を否定することもできます。デフォルトでは、修飾子はコマンド・ラインに指定されません。

/ANNOTATIONS 修飾子だけを指定し、パラメータを指定しなかった場合、デフォルトは ALL になります。

#### /ALPHA\_REGISTER\_MAPPING 修飾子

OpenVMS I64 BLISS コンパイラでは、ソースを変更せずに ALPHA\_REGISTER\_MAPPING を有効にすることができる新しいコンパイル修飾子がサポートされます。これは位置依存修飾子です。モジュールのコンパイル行にこの修飾子を指定すると、モジュール・ヘッダに ALPHA\_REGISTER\_MAPPING スイッチを設定した場合と同じ効果があります。

/ALPHA\_REGISTER\_MAPPING 情報メッセージ OpenVMS I64 BLISS では、3 つの新しい情報メッセージが追加されています。

- /ALPHA\_REGISTER\_MAPPING 修飾子をコマンド・ラインに指定すると、以下のメッセージが表示されます。

```
%BLS64-I-TEXT, Alpha Register Mapping enabled by the command line
```

- スイッチ ALPHA\_REGISTER\_MAPPING をモジュール・ヘッダに指定するか、または SWITCH 宣言の引数として指定すると、以下のメッセージが表示されます。

```
MODULE SIMPLE (MAIN=TEST, ALPHA_REGISTER_MAPPING)=  
.....^  
%BLS64-I-TEXT, Alpha Register Mapping enabled
```

- スイッチ NOALPHA\_REGISTER\_MAPPING をモジュール・ヘッダに指定するか、または SWITCH 宣言の引数として指定すると、以下のメッセージが表示されます。

```
MODULE SIMPLE (MAIN=TEST, NOALPHA_REGISTER_MAPPING)=  
.....^  
%BLS64-I-TEXT, Alpha Register Mapping disabled
```

#### 不可分な操作のための ADD, AND, OR 組み込み関数

OpenVMS I64 BLISS では、不可分 (atomic) なメモリ更新のために、ADD\_ATOMIC\_XXXX, AND\_ATOMIC\_XXXX, および OR\_ATOMIC\_XXXX 組み込み関数がサポートされます。これらの組み込み関数の一覧については、マシン固有の組み込み関数を参照してください。

これらの組み込み関数をサポートする I64 インストラクションは、操作が正常終了するまで待つため、オプションの retry-count 入力パラメータは除外されています。これらの組み込み関数の形式は以下のように変更されました。

*option*\_ATOMIC\_size(*ptr*, *expr* [:*old\_value*] ) !Optional output

ここで、

*option* は AND, ADD, OR のいずれかです。

*size* は LONG または QUAD です。

*ptr* は自然なアラインメントのアドレスでなければなりません。



返される値は 0 (操作失敗) または 1 (操作成功) です。

操作は、式 `expr` の、`ptr` によって示されるデータ・セグメントへの不可分な加算 (または AND もしくは OR) です。

オプションの出力パラメータ `old_value` は、`ptr` によって示されるデータ・セグメントの以前の値に設定されます。

OpenVMS Alpha BLISS のオプションの `retry-count` パラメータを使用すると、構文エラーになります。

不可分な操作のための `TESTBITxxl` および `TESTBITxx` 組み込み関数

OpenVMS I64 BLISS では、不可分な操作のために `TESTBITxxI` および `TESTBITxx` 組み込み関数がサポートされます。サポートされる組み込み関数の一覧については、マシン固有の組み込み関数を参照してください。

これらの組み込み関数をサポートする I64 インストラクションは、操作が正常終了するのを待機するので、オプションの入力パラメータ `retry-count` とオプションの出力パラメータ `success_flag` は取り除かれています。これらの組み込み関数の形式は以下のとおりです。

```
TESTBITxxx( field )
```

OpenVMS Alpha BLISS のオプションのパラメータ `retry-count` や `success_flag` を使用すると、構文エラーになります。

ロングワード書き込みとクォドワード書き込みの単位

OpenVMS I64 BLISS では、ここで説明する `/GRANULARITY=keyword` 修飾子、スイッチ `DEFAULT_GRANULARITY=n`、およびデータ属性 `GRANULARITY(n)` がサポートされます。

ユーザはコマンド・ライン修飾子 `/GRANULARITY=keyword`、スイッチ `DEFAULT_GRANULARITY=n`、およびデータ属性 `GRANULARITY(n)` を使用することにより、ストアとフェッチの単位 (Granularity) を制御できます。

コマンド・ライン修飾子に指定するキーワードは、`BYTE`、`LONGWORD` または `QUADWORD` でなければなりません。値 `n` は、0 (バイト)、2 (ロングワード) または 3 (クォドワード) でなければなりません。

これらを組み合わせて使用した場合、優先順位が最も高いのはデータ属性です。`SWITCHES` 宣言でスイッチを使用した場合、同じスコープ内でその後に宣言されるデータの単位 (Granularity) を設定します。スイッチはモジュール・ヘッダでも使用できます。最も優先順位が低いのはコマンド・ライン修飾子です。

#### シフト (Shift) 組み込み関数

OpenVMS I64 BLISS では、既知の方向にシフトを行う組み込み関数がサポートされます。これらの組み込み関数の一覧については、マシン固有の組み込み関数を参照してください。これらの関数は、0..%BPVAL-1 の範囲で行われるシフトに対してだけ有効です。

#### 比較 (Compare) およびスワップ (Swap) 組み込み関数

OpenVMS Alpha と OpenVMS I64 の両方の BLISS で、以下の新しい比較およびスワップ組み込み関数がサポートされます。

- CMP\_SWAP\_LONG(addr, comparand, value)
- CMP\_SWAP\_QUAD(addr, comparand, value)

これらの関数は、addr にあるロングワードまたはクォドワードを comparand と比較し、等しい場合は値を addr にストアするという、連動した操作を実行します。操作の成功 (1) または失敗 (0) を示すインジケータが返されます。

#### I64 固有のマルチメディア・インストラクション

I64 固有のマルチメディア・タイプのインストラクションへのアクセスをサポートする計画はありません。

#### リンケージ

CALL リンケージ OpenVMS Alpha BLISS に関してここで説明する CALL リンケージは、OpenVMS I64 BLISS でもサポートされます。

32 ビット・コンパイラでコンパイルされたルーチンは、64 ビット・コンパイラでコンパイルされたルーチン呼び出すことができ、その逆も可能です。64 ビットから 32 ビットに短縮される場合はパラメータが切り捨てられ、32 ビットから 64 ビットに拡張される場合は符号拡張されます。

デフォルト設定では、CALL リンケージは引数の数を渡します。この設定は、NOCOUNT リンケージ・オプションを使用することで無効にすることができます。

引数はクォドワードで渡されますが、32 ビット・コンパイラは下位 32 ビットしか「確認」できません。

JSB リンケージ OpenVMS I64 BLISS コンパイラには JSB リンケージ・タイプがあります。JSB リンケージで宣言されたルーチンは、OpenVMS I64 用に開発された JSB ルールに従います。

#### /[NO]TIE 修飾子

この修飾子のサポートは、OpenVMS I64 でも継続されます。デフォルトは/NOTIE です。

コンパイルされたコードから変換されたイメージを呼び出したり、変換されたイメージからコンパイルされたコードを呼び出す必要がある場合には、TIE を使用すると、コンパイルされたコードと変換されたイメージとを組み合わせ使用できます。

特に、TIE は以下の操作を行います。

- コンパイルされたプログラムにプロシージャ・シグネチャ情報を挿入します。この結果、サイズが大きくなる可能性があり、リンク時とイメージ起動時に処理される再配置の回数も多くなる可能性があります、その他の点ではパフォーマンスへの影響はありません。
- プロシージャ値の呼び出し (間接呼び出しまたは演算呼び出しとも呼ばれます) が、サービス・ルーチン (OTSS\$CALL\_PROC) を使用してコンパイルされます。このルーチンは、ターゲット・プロシージャが IPF のネイティブ・コードなのか、変換されたイメージ内にあるのかを判断し、結果に従って操作を続行します。

/ENVIRONMENT=([NO]FP) および ENVIRONMENT([NO]FP)

OpenVMS Alpha BLISS では、/ENVIRONMENT=([NO]FP) 修飾子と ENVIRONMENT([NO]FP) スイッチが提供されており、これらを使用すると、コンパイラは特定の整数除算に対して浮動小数点レジスタの使用を禁止します。

しかし、OpenVMS I64 BLISS では、I64 のアーキテクチャ上の機能により、/ENVIRONMENT=NOFP コマンド修飾子および ENVIRONMENT(NOFP) スイッチは、浮動小数点レジスタの使用を完全に禁止するわけではありません。ソース・コードでは浮動小数点演算は使用しないように制限されますが、特定の演算 (特に整数乗算と整数除算、およびそれを暗黙に含む構造) 用に生成されたコードでは、一部の浮動小数点レジスタだけを使用するように制限されます。特に、このオプションを指定した場合、コンパイラは f6 ~ f11 だけを使用するように制限され、『Intel Itanium Processor-Specific Application Binary Interface』で説明している ELF EF\_IA\_64\_REDUCEFP オプションを設定します。

/ENVIRONMENT=FP コマンド修飾子と ENVIRONMENT(FP) スイッチには影響ありません。

#### 浮動小数点のサポート

ここでは、BLISS コンパイラを使った浮動小数点演算のサポートについて説明します。

**浮動小数点組み込み関数** BLISS では、浮動小数点数値に関して、高いレベルのサポートは提供されません。浮動小数点リテラルを作成する機能はサポートされます。また、浮動小数点演算および変換操作を行うためのマシン固有の組み込み関数も用意されています。サポートされる組み込み関数の一覧については、マシン固有の組み込み関数を参照してください。

どの浮動小数点組み込み関数もオーバーフローは検出しないため、値を返しません。

**浮動小数点リテラル** OpenVMS I64 BLISS でサポートされる浮動小数点リテラルは、OpenVMS Alpha BLISS の場合と同じで、%E、%D、%G、%S、および%T がサポートされます。

**浮動小数点レジスタ** I64 浮動小数点レジスタの直接使用はサポートされていません。

浮動小数点パラメータを受け渡しする BLISS 以外のルーチンの呼び出し 値渡しで浮動小数点パラメータを受け取り，浮動小数点値または複素数値を返す BLISS 以外の標準ルーチン呼び出すことができます。

OpenVMS I64 の BLISS では，以下の標準関数がサポートされます。

- %FFLOAT
- %DFLOAT
- %GFLOAT
- %SFLOAT
- %TFLOAT

新しいレキシカルと拡張されたレキシカル

OpenVMS I64 のコンパイラ BLISS32I と BLISS64I をサポートするために，以下の新しいコンパイラ状態レキシカルが BLISS に追加されました。

- %BLISS は BLISS32V，BLISS32E，BLISS64E，BLISS32I，および BLISS64I を認識するようになりました。  
%BLISS(BLISS32) は，すべての 32 ビット BLISS コンパイラの場合に真になります。  
%BLISS(BLISS32V) は，VAX BLISS (BLISS-32) の場合だけ真になります。  
%BLISS(BLISS32E) は，32 ビット Alpha コンパイラの場合に真になります。  
%BLISS(BLISS64E) は，64 ビット Alpha コンパイラの場合に真になります。  
%BLISS(BLISS32I) は，32 ビット I64 コンパイラの場合に真になります。  
%BLISS(BLISS64I) は，64 ビット I64 コンパイラの場合に真になります。
- レキシカル%BLISS32I と%BLISS64I が追加されました。その動作は，%BLISS に対する新しいパラメータと同様です。
- %HOST および%TARGET レキシカルのキーワードにおける Intel Itanium アーキテクチャのサポートが，OpenVMS I64 BLISS で追加されました。

OpenVMS I64 BLISS での IPF ショート・データ・セクションのサポート

IPF 呼び出し規則では，8 バイト以下のサイズのグローバル・データ・オブジェクトはすべて，ショート・データ・セクションに割り振ることが要求されています。

ショート・データ・セクションのアドレスは，GP ベース・レジスタの内容に 22 ビット・リテラルを加算する効率のよいコード・シーケンスによって設定されます。このコード・シーケンスは，すべてのショート・データ・セクションを組み合わせたサイズを制限します。ショート・データ・セクションに割り当てられたデータ容量の合計が  $2^{**}22$  バイトを超えると，リンカ・エラーが発生します。IPF のコンパイラは，ショート・グローバル・データ・セクションおよびショート・エクスターナル・データ・セクションにアクセスするときに，GP 相対アドレッシングを使用できます。

OpenVMS I64 BLISS は、PSECT 属性 GP\_RELATIVE および新しい PSECT 属性 SHORT に対して、新しい動作を実行します。この新しい属性はショート・データ・セクションの割り当てをサポートします。

PSECT 属性として GP\_RELATIVE キーワードを指定すると、PSECT はショート・データを含むセクションとしてラベルが付けられるため、リンクは GP ベース・アドレスの近くに PSECT を割り振ります。

SHORT 属性の構文は以下のとおりです。

SHORT ( psect-name )

SHORT 属性には、以下のルールが適用されます。

- SHORT 属性に指定された PSECT 名がまだ宣言されていない場合は、SHORT 属性にその名前が指定されることで宣言が成立します。SHORT 属性を含む PSECT の属性は、SHORT 属性に指定された PSECT の属性になりますが、SHORT 属性で宣言された PSECT 名は SHORT 属性を持たず、GP\_RELATIVE 属性を持ちます。
- SHORT 属性に指定された PSECT 名が以前に宣言されている場合は、PSECT の属性は変更されません。SHORT 属性に指定された PSECT に GP\_RELATIVE 属性がない場合には、警告メッセージが生成されます。
- ストレージ・クラスが OWN、GLOBAL、PLIT のいずれかのデータ・オブジェクトのサイズが 8 バイト以下であり、SHORT 属性を含む PSECT に割り当てられるように指定された場合、そのオブジェクトは、SHORT 属性に指定されている PSECT に割り当てられます。これは、1 ステップのプロセスであり、再帰的ではありません。ショート・データ・オブジェクトに SHORT 属性によって名前が変更された割り当て PSECT が含まれている場合、名前が変更された PSECT の SHORT 属性は、その後の名前変更の対象として考慮されません。
- 8 バイトより大きなサイズのデータ・オブジェクトは、SHORT 属性を無視します。
- CODE、INITIAL、および LINKAGE ストレージ・クラスのデータ・オブジェクトは、そのサイズに関係なく、SHORT 属性を無視します。
- SHORT 属性による PSECT 名の変更では、PLIT データの前にあるカウント・ワードのサイズは PLIT オブジェクトのサイズに含まれません。

例 BLISS コードで PSECT を使用する例を以下に示します。

PSECT

```
NODEFAULT = $GLOBAL SHORT$  
  (READ,WRITE,NOEXECUTE,NOSHARE,NOPIC,CONCATENATE,LOCAL,ALIGN(3),  
   GP_RELATIVE),
```

## アプリケーションのポーティングの準備

### 6.3 BLISS コンパイラ

```
! The above declaration of $GLOBAL_SHORT$ is not needed.  If the above
! declaration were deleted then the SHORT($GLOBAL_SHORT$) attribute in
! the following declaration would implicitly make an identical
! declaration of $GLOBAL_SHORT$.

GLOBAL = $GLOBAL$
  (READ,WRITE,NOEXECUTE,NOSHARE,NOPIC,CONCATENATE,LOCAL,ALIGN(3),
   SHORT($GLOBAL_SHORT$)),

NODEFAULT = MY_GLOBAL
  (READ,WRITE,NOEXECUTE,SHARE,NOPIC,CONCATENATE,LOCAL,ALIGN(3)),

PLIT = $PLIT$
  (READ,NOWRITE,NOEXECUTE,SHARE,NOPIC,CONCATENATE,GLOBAL,ALIGN(3),
   SHORT($PLIT_SHORT$));

GLOBAL
  X1,                      ! allocated in $GLOBAL_SHORT$
  Y1 : VECTOR[2, LONG],    ! allocated in $GLOBAL_SHORT$
  Z1 : VECTOR[3, LONG],    ! allocated in $GLOBAL$
  A1 : PSECT(MY_GLOBAL),   ! allocated in MY_GLOBAL
  B1 : VECTOR[3, LONG] PSECT(MY_GLOBAL), ! allocated in MY_GLOBAL
  C1 : VECTOR[3, LONG]
        PSECT($GLOBAL_SHORT$); ! allocated in $GLOBAL_SHORT$

PSECT GLOBAL = MY_GLOBAL;
! use MY_GLOBAL as default for both noshort/short

GLOBAL
  X2,                      ! allocated in MY_GLOBAL
  Y2 : VECTOR[2, LONG],    ! allocated in MY_GLOBAL
  Z2 : VECTOR[3, LONG],    ! allocated in MY_GLOBAL
  A2 : PSECT($GLOBAL$),    ! allocated in $GLOBAL_SHORT$
  B2 : VECTOR[3, LONG] PSECT($GLOBAL$); ! allocated in $GLOBAL$;

! Note that the allocations of A1, X2 and Y2 violate the calling
! standard rules.  These variables cannot be shared with other
! languages, such as C or C++.

PSECT GLOBAL = $GLOBAL$;
! back to using $GLOBAL$/ $GLOBAL_SHORT$ as default noshort/short

GLOBAL BIND
  P1 = UPLIT("abcdefghi"), ! allocated in $PLIT$
  P2 = PLIT("abcdefgh"),  ! allocated in $PLIT_SHORT$
  P3 = PSECT(GLOBAL) PLIT("AB"), ! allocated in $GLOBAL_SHORT$
  p4 = PSECT($PLIT_SHORT$)
        PLIT("abcdefghijklmn"), ! allocated in $PLIT_SHORT$
  P5 = PSECT(MY_GLOBAL) PLIT("AB"); ! allocated in MY_GLOBAL
```

---

#### 注意

---

- A1, X2, Y2, および P5 の割り当ては呼び出し規則に違反します。これらの変数は、C や C++ などの他の言語と共有できません。BLISS および MACRO で書かれたモジュールとの共有は可能です。

- 現在の OpenVMS I64 BLISS の設計では、EXTERNAL 変数参照で GP\_RELATIVE アドレッシング・モードはサポートされていません。しかし、EXTERNAL 変数で使用する通常の GENERAL アドレッシング・モードは、GP\_RELATIVE セクションを正しく参照します。現時点では、ADDRESSING\_MODE(GP\_RELATIVE) 属性を BLISS に追加する予定はありません。
- 

## 6.4 COBOL

COBOL Version 2.8 は、Alpha と I64 の両方でサポートされます。OpenVMS I64 で COBOL を使用する場合は制限事項と既知の問題点については、COBOL のリリース・ノートを参照してください。

### 6.4.1 浮動小数点演算

OpenVMS Alpha の COBOL コンパイラのデフォルトは/FLOAT=D\_FLOAT です。I64 の場合、/FLOAT=IEEE\_FLOAT がデフォルトです。

COBOL コンパイラは/IEEE\_MODE 修飾子をサポートしません。COBOL RTL は、例外処理とラウンディング処理の点で Alpha の COBOL ランタイム環境と同じ環境を I64 で設定します。

I64 上で浮動小数点を COBOL がどのように処理するかについては、COBOL のリリース・ノートとホワイト・ペーパー『Intel® Itanium®における OpenVMS 浮動小数点演算について』を参照してください。

このホワイト・ペーパーおよび関連の技術資料の入手方法については、本書のまえがきを参照してください。

### 6.4.2 /ARCH および/OPTIMIZE=TUNE 修飾子

「コンパイル・アンド・ゴー」の互換性を維持するために、/ARCH および /OPTIMIZE=TUNE 修飾子に対する Alpha 固有のキーワードがコンパイラ起動コマンドで受け付けられます。これらの値が無視されることを示す情報メッセージが出力されます。

/ARCH および/OPTIMIZE=TUNE 修飾子に対する I64 固有のキーワードは、OpenVMS I64 コンパイラで定義されていますが、これは開発用に提供されるものです。これらの修飾子の動作は予測できないため、使用すべきではありません。

---

## 6.5 Fortran

OpenVMS I64 では、Fortran 90 V8.0 がサポートされます。Fortran 77 は OpenVMS I64 ではサポートされません。詳細については、第 6.5.2 項を参照してください。

OpenVMS I64 システム向けの HP Fortran では、OpenVMS Alpha システム向けの HP Fortran と同じコマンド・ライン・オプションおよび言語機能が提供されますが、いくつかの例外があります。ここでは、これらの例外について説明します。

### 6.5.1 浮動小数点演算

Fortran のリリース・ノートおよびホワイトペーパー『Intel® Itanium®アーキテクチャにおける OpenVMS 浮動小数点演算について』では、HP Fortran for I64 で浮動小数点の問題がどのように処理されるかについて説明しています。

このホワイトペーパーと、OpenVMS から Itanium アーキテクチャへの移行に関するその他のリソースが入手できる Web サイトについては、「まえがき」の「関連資料」の項を参照してください。

ここでは、その要点をまとめます。

- デフォルトの浮動小数点データ・タイプは IEEE です (つまり、デフォルトは/FLOAT=IEEE\_FLOAT です)。

OpenVMS Alpha のコンパイラでは、デフォルトの浮動小数点データ・タイプは/FLOAT=G\_FLOAT です。OpenVMS I64 システムでは、IEEE 以外の浮動小数点表現はハードウェアでサポートされません。VAX 浮動小数点形式は、算術演算を実行する前に IEEE 形式に変換し、結果を適切な VAX 形式に戻す実行時コードを生成することで、コンパイラでサポートされています。このため、実行時のオーバーヘッドが増加し、Alpha (および VAX) のハードウェアで同じ演算を実行する場合と比較すると精度が少し低下します。

ソフトウェアでの VAX 形式のサポートは、ディスクに格納されているバイナリ形式の浮動小数点データを取り扱う必要のある特定のアプリケーションで互換性を維持するための重要な機能ですが、これの利用はお勧めできません。この変更は、デフォルトが VAX の/FLOAT=D\_FLOAT から Alpha の/FLOAT=G\_FLOAT に変更されたのに類似しています。

支障がなければ、最大限のパフォーマンスと精度のために/FLOAT=IEEE\_FLOAT (デフォルト) を使用してください。

/FLOAT のデフォルトが変更されたことにより、/CONVERT=NATIVE (デフォルト) の使用に影響があるので注意してください。このスイッチは、一般的な浮動小数点データ・タイプに一致するという仮定のもと、フォーマットされていないデータを未変換のまま入力に残します。



Alpha 上で/FLOAT=G\_FLOAT/CONVERT=NATIVE を指定して (明示的に G\_FLOAT および NATIVE を指定するか、どちらもデフォルトのままとして) 作成された Fortran アプリケーションが書き込んだファイルは、G\_FLOAT になります。このファイルは、/FLOAT=G\_FLOAT/CONVERT=NATIVE または/FLOAT=IEEE/CONVERT=VAXG で作成した I64 アプリケーションで読み込むことができます。しかし、/CONVERT=NATIVE で作成したアプリケーションでは、/FLOAT のタイプがデフォルトの IEEE 形式となるため、読み込むことができません。

- /IEEE\_MODE 修飾子のデフォルトは/IEEE\_MODE=DENORM\_RESULTS に設定されます。

このデフォルトは Alpha の場合と異なります。Alpha のデフォルトは /IEEE\_MODE=FAST です。"FAST"となっているにもかかわらず、/IEEE\_MODE=FAST は I64 (あるいは EV6 以降の Alpha プロセッサ) の実行時パフォーマンスにそれほど大きな効果がありません。

- ユーザは/FLOAT の値と/IEEE\_MODE の値を 1 つずつ選択して、アプリケーション全体でその値を使用する必要があります。これは、ハードウェアにアーキテクチャ上の違いがあるために、複合モード・アプリケーションが OpenVMS I64 システムでは (一般に) 動作しないからです。これは OpenVMS Alpha から変更された点です (OpenVMS Alpha では複合モードアプリケーションが動作します)。特に、ルーチンごと、ファイルごと、ライブラリごとのモードの設定は機能しません。

Alpha と同様に、/IEEE\_MODE は、ユーザが/FLOAT=IEEE を選択するか、それをデフォルトとして適用した場合にだけ設定できます。G\_FLOAT および D\_FLOAT で使用される IEEE\_MODE モードは、VAX 形式のサポートを実現する IEEE 形式の計算をサポートするのに適切のように、コンパイラが選択したものです。

- プログラムで浮動小数点例外モードを変更した場合は、例外ハンドラによる終了も含めて、ルーチンの終了時に例外モードを元に戻さなければなりません。元に戻さないと、コンパイラは、呼び出されたルーチンが現在のモードを変更していないものと考え、予測しない結果が発生することがあります。この要件については、ユーザ作成ライブラリの場合も対応しなければなりません。
- 例外ハンドラの開始時には、ハンドラがコンパイルされたときのモードではなく、例外が発生した時点で有効だった浮動小数点モードになります。

ホワイトペーパー『Intel® Itanium®アーキテクチャにおける OpenVMS 浮動小数点演算について』にあるように、I64 でアプリケーションの実行中に発生する浮動小数点例外の番号、種類、場所は、Alpha と同じではありません。特に、VAX 形式の浮動小数点で違いが顕著で、一般に、計算後に元の VAX 形式に変換する際にだけ例外が発生します。

VAX 形式の浮動小数点のユーザに関係しそうな影響には、以下の 3 つがあります。

- Alpha で例外が発生していた計算途中の値 (たとえば、" $X*Y + X/Y$ "での " $X/Y$ ") で、I64 では例外が発生しないことがあります。
- VAX 形式の浮動小数点に変換されない値 (たとえば " $IF (X/Y > 0.0)$ "での " $X/Y$ ") は、I64 では例外を発生しません。
- OpenVMS Alpha および VAX の Fortran アプリケーションは、  
/CHECK=UNDERFLOW 修飾子を指定してコンパイルすることによりアンダーフロー・トラップを有効にしていない限り、VAX 形式の浮動小数点演算に対してはアンダーフローを報告しません。これは、OpenVMS I64 システムの場合も同じです。ただし、以下の点に注意してください。

すべての I64 浮動小数点演算は IEEE 形式の演算で実装されているため、  
/CHECK=UNDERFLOW でアンダーフロー・トラップを有効にした場合、VAX 形式ではなく IEEE 形式の式でアンダーフローを評価すると例外処理が発生する場合があります。

つまり、計算値が VAX 形式では有効範囲に収まるが IEEE 形式ではデノーマル値の範囲となる場合があるため、Alpha あるいは VAX プログラムと比較すると、/CHECK=UNDERFLOW によりアンダーフロー例外が増える可能性があります。

### 6.5.2 サポートされるのは F90 コンパイラのみ

F77 コンパイラは、以前は/OLD\_F77 修飾子で起動されていましたが、現在は使用できなくなりました。Alpha F77 コンパイラで提供されていて、I64 および Alpha の F90 ではまだ提供されていない機能のうち、以下の機能は現在開発中です。

- CDD および FDML のサポートは F90 で実装されていますが、まだ十分なテストは行なわれていません。
- F77 コンパイラで以前コンパイルされていて、F90 コンパイラでコンパイルされないコードは、問題報告メカニズムを通じて報告してください。すべての相違点を排除することは不可能ですが、開発チームは F90 コンパイラを強化することで、これらの違いをできるだけ少なくすることに努めています。

---

#### 注意

---

/OLD\_F77 修飾子はサポートされませんが、これを/F77 修飾子と混同しないでください。/F77 修飾子は、FORTRAN-66 との間で互換性のないステートメントについて、コンパイラがFORTRAN-77の解釈ルールを使用することを示し、この修飾子は今後もサポートされます。

---

### 6.5.3 /ARCH および/OPTIMIZE=TUNE 修飾子

「コンパイル・アンド・ゴー」の互換性を維持するために、/ARCH および /OPTIMIZE=TUNE 修飾子に対する Alpha 固有のキーワードがコンパイラ起動コマンドで受け付けられます。これらの値が無視されることを示す情報メッセージが出力されます。

/ARCH および/OPTIMIZE=TUNE 修飾子に指定する I64 固有のキーワードは、OpenVMS I64 で定義されていますが、これは開発のためにだけ提供されるものです。これらの修飾子の動作は予測できないため、使用すべきではありません。

---

## 6.6 HP C/ANSI C

I64 システムでは、HP C Version 7.1 がサポートされます。ポーティングに関する注意事項については、HP C のリリース・ノートを参照してください。

### 6.6.1 I64 浮動小数点のデフォルト

Alpha のネイティブ・コンパイラのデフォルトは/FLOAT=G\_FLOAT です。I64 のデフォルトは/FLOAT=IEEE\_FLOAT/IEEE=DENORM です。

OpenVMS I64 では、IEEE 以外の浮動小数点表現はハードウェアでサポートされません。VAX 浮動小数点形式は、算術演算を実行する前に IEEE 形式に変換し、結果を適切な VAX 形式に戻す実行時コードを生成することで、コンパイラでサポートされています。このため、実行時のオーバーヘッドが増加し、Alpha (および VAX) のハードウェアで同じ演算を実行する場合と比較すると、精度が少し低下します。ソフトウェアでの VAX 形式のサポートは、ディスクに格納されているバイナリ形式の浮動小数点データを取り扱う必要のある特定のアプリケーションで、互換性を維持するための重要な機能ですが、この機能の利用はお勧めできません。この変更は、デフォルトが VAX の/FLOAT=D\_FLOAT から Alpha の/FLOAT=G\_FLOAT に変更されたのに類似しています。

また、/IEEE\_MODE のデフォルトは、OpenVMS Alpha では FAST ですが、OpenVMS I64 では DENORM\_RESULTS に変更されています。つまり、VAX 浮動小数点形式や/IEEE\_MODE=FAST を使用すると致命的な実行時エラーが発生していた浮動小数点演算で、デフォルトにより、Inf または Nan として出力される値が生成されるようになりました (業界標準の動作)。また、このモードでの非ゼロ最小値は以前よりはるかに小さくなります。これは、値が非常に小さくなって正規化した状態では表現できなくなったときに、結果がデノーマル範囲にあることが許されるからで、ただちに 0 になることはありません。

### 6.6.2 /IEEE\_MODE 修飾子のセマンティック

OpenVMS Alpha では、/IEEE\_MODE 修飾子は一般に、コンパイルで生成されるコードに対して最大の影響を与えます。異なる/IEEE\_MODE 修飾子を使用してコンパイルされた関数の間で呼び出しを行うと、各関数の動作は、それがコンパイルされたときのモードに従います。

OpenVMS I64 では、/IEEE\_MODE 修飾子は基本的にプログラム起動時のハードウェア・レジスタの設定にだけ影響します。一般に、各関数に対する/IEEE\_MODE の動作は、メイン・プログラムのコンパイル時に指定された/IEEE\_MODE オプションによって制御されます。コンパイラは、各コンパイルでできるオブジェクト・モジュールを、コンパイル時の修飾子で指定された浮動小数点制御でマークします。I64 リンカが実行イメージを作成する際、イメージの main エントリ・ポイントへの転送アドレスを提供したオブジェクト・モジュールから浮動小数点制御をそのイメージ自体へコピーします。この処理は、そのイメージに対するホール・プログラム浮動小数点モード (whole program floating-point mode) と呼ばれます。この後、実行のためにイメージを起動すると、このホール・プログラム浮動小数点モードに従ってハードウェアの浮動小数点制御が初期化されます。実行時に特定の制御設定が必要となるようなコードのセクションから制御が移る場合に、ホール・プログラム浮動小数点モードの設定がリストアされるよう、実行時に浮動小数点制御を修正するようなコードを作成してください。

/IEEE\_MODE 修飾子がメイン・プログラムを含まないコンパイルに適用される場合も、この修飾子には効果があります。浮動小数点定数式の評価に影響を与え、そのコンパイルからの呼び出しに対して、算術演算ライブラリで使用される EXCEPTION\_MODE を設定します。

/IEEE\_MODE 修飾子は、そのコンパイルでインライン・コードとして生成された浮動小数点演算の例外動作には影響を与えません。したがって、浮動小数点例外動作がアプリケーションにとって重要な場合は、メイン・プログラムを含むコンパイルも含めて、すべてのコンパイルで/FLOAT および/IEEE\_MODE を同じ設定にする必要があります。

Alpha でも、/IEEE\_MODE=UNDERFLOW\_TO\_ZERO に設定するには、実行時状態レジスタを設定する必要があります。したがって、この設定を他のコンパイルでも有効にするには、メイン・プログラムを含むコンパイルでこの設定を指定する必要があります。

最後に、/ROUNDING\_MODE 修飾子は/IEEE\_MODE と同じように影響を与え、ホール・プログラム浮動小数点モードに含まれ、また、VAX 浮動小数点演算は実際には IEEE インストラクションで実行されるため、VAX 形式の浮動小数点を使用するコンパイルは/IEEE\_MODE=DENORM/ROUND=NEAREST でコンパイルした場合と同じホール・プログラム浮動小数点モード設定で動作することに注意してください。

### 6.6.3 定義済みマクロ

I64 コンパイラでは、Alpha のネイティブ・コンパイラと同じ意味を持つ多くのマクロがあらかじめ定義されていますが、Alpha アーキテクチャを指定するマクロは定義されていません。その代わりに、`__ia64`マクロと`__ia64__`マクロが、I64 向けの Intel および gcc コンパイラと同じ手法で定義されています。G\_FLOAT から IEEE への浮動小数点表現の変更は、定義済みマクロでもデフォルトで反映されています。

一部のユーザは、`__ALPHA`マクロが定義されていない場合、ターゲットは VAX システムでなければならないという前提で書かれた条件付きソース・コードを簡単に扱う方法として、`/DEFINE` を使用するか、またはヘッダ・ファイルに記述して、明示的に `__ALPHA`マクロを定義しようとしていました。しかし、このような定義を行うと、CTRL ヘッダおよびその他の OpenVMS ヘッダは I64 に対して誤ったパスを選択します。このコンパイラを使用する場合、Alpha アーキテクチャの定義済みマクロを定義すべきではありません。

---

## 6.7 HP C++

OpenVMS I64 では、HP C++ Version 7.1 がサポートされます。ポーティングに関する注意事項については、HP C++ のドキュメントも参照してください。

### 6.7.1 浮動小数点と定義済みマクロ

C コンパイラの浮動小数点のデフォルト、`/IEEE_MODE` セマンティック、定義済みマクロに関する注意事項が、C++ コンパイラにもそのまま適用されます。

### 6.7.2 long double

`long double` タイプは常に 128 ビット IEEE 4 倍精度で表現されます。`/L_DOUBLE_SIZE=64` 修飾子は無視され、警告メッセージが表示されます。Alpha では C++ ライブラリの 64 ビット `long double` のサポートに制限がありました。64 ビット浮動小数点タイプが必要なコードでは、`long double` の代わりに `double` を使用していました。

### 6.7.3 オブジェクト・モデル

I64 の C++ コンパイラが使用するオブジェクト・モデルと名前マングリング・スキーマは、Alpha の場合と大きく異なります。ARM オブジェクト・モデルは使用できません。使用できるのは、ANSI/ISO C++ 標準をサポートするオブジェクト・モデルのみです。ただしこれは、Alpha で実装されている `/MODEL=ANSI` オブジェクト・モデルとも異なり、I64 のモデルは業界標準の I64 ABI の実装となっています。C++ オブジェクトの配置 (非 POD データ)、あるいは `.obj` ファイルにエンコードされた外部マングル名に依存するプログラムは、作成し直す必要があります。このようなプログラムは、本質的に実装に大きく依存します。しかし、実装に依存しない適切な方法

で標準の C++ 機能を使用するプログラムであれば、ポーティングは難しくありません。

#### 6.7.4 言語ダイアレクト

cfront ダイアレクトはサポートされません (Alpha 版からも削除されます)。  
/standard=cfront を使用してコンパイルした場合は、relaxed\_ansi ダイアレクトが使用されます。

#### 6.7.5 テンプレート

OpenVMS I64 では、.OBJ ファイルは EOBJ ではなく ELF 形式で実装されます。また I64 リンカと連動し、C++ テンプレートおよびインライン関数を使用した場合に発生するリンク時の重複定義の問題を解決するために Windows および UNIX プラットフォームの両方でよく使用される COMDAT セクションの概念をサポートします。Alpha では、リンカに単一の定義インスタンスを提示するリポジトリ・メカニズムによってこれらの問題を処理します。I64 ではこの処理にリポジトリ・メカニズムは必要なく、リンカが自動的に重複を解消します。このため、Alpha でサポートされるリポジトリ・ベースのテンプレート・インスタンス化オプションは、IPF ではサポートされません。リポジトリでオブジェクトを処理しようとする Alpha ビルド・プロシージャは I64 では使えないので、変更が必要になります (I64 ではリポジトリにはオブジェクトがなく、デマングラ・データベースです)。ほとんどの場合、コンパイラの COMDAT インスタンス化の使用により、ビルド・プロシージャでリポジトリを直接扱う必要はなくなります。

---

### 6.8 Java

OpenVMS I64 では、Java™ Version 1.4.2-1 がサポートされます。OpenVMS Alpha の Java と OpenVMS I64 の Java の間に相違点はありません。

---

### 6.9 MACRO-32

OpenVMS Alpha でコンパイルされた大部分の VAX MACRO プログラムは、まったく変更せずに OpenVMS I64 で再コンパイルできます。しかし、非標準戻り値を返すルーチン呼び出すプログラムや、JSB インストラクションを使用して、他の言語で作成されたルーチン呼び出すプログラムでは、ソース・ファイルに新しいディレクティブを追加する必要があります。詳細については、『OpenVMS MACRO-32 Porting and User's Guide』を参照してください。

## 6.10 HP Pascal

Alpha と I64 の両方で HP Pascal Version 5.9 が利用できます。

OpenVMS Alpha 上の Pascal と OpenVMS I64 上の Pascal では、明らかになっている言語上の違いはありません。唯一の違いは、HP のほかの I64 コンパイラと同様に、浮動小数点のデフォルト・データ・タイプが VAX 形式ではなく IEEE 形式である点です。

### 6.10.1 /ARCH および/OPTIMIZE=TUNE 修飾子

「コンパイル・アンド・ゴー」の互換性を維持するために、/ARCH および /OPTIMIZE=TUNE 修飾子に対する Alpha 固有のキーワードがコンパイラ起動コマンドで受け付けられます。これらの値が無視されることを示す情報メッセージが出力されます。

/ARCH および/OPTIMIZE=TUNE 修飾子に対する I64 固有のキーワードは、OpenVMS I64 コンパイラで定義されていますが、これは開発用に提供されるものです。これらの修飾子の動作は予測できないため、使用すべきではありません。





---

## その他の検討事項

この章では、ポーティングに関するその他の検討事項について説明します。

---

### 7.1 ハードウェアに関する検討事項

ここでは、Alpha、PA-RISC、および Itanium プロセッサ・ファミリに関して説明します。また、プロセッサに関連する開発上の問題点について説明し、C 言語とアセンブリ言語の両方を使用して、典型的なプロセッサ依存コードの例を示します。

#### 7.1.1 Intel Itanium プロセッサ・ファミリの概要

Itanium アーキテクチャは、エンジニアリング・ワークステーションおよび e コマース・サーバ向けの 64 ビット・プロセッサとして、HP と Intel が開発しました。Itanium プロセッサ・ファミリでは、EPIC (Explicitly Parallel Instruction Computing) という新しいアーキテクチャが採用されています。

EPIC の設計では、最も重要な目標として、インストラクション・レベルの並列処理、ソフトウェア・パイプライン化、スペキュレーション、プレディケーション、大きなレジスタ・ファイルなどが設定されました。これらの機能を実現することで、Itanium プロセッサは複数のインストラクションを同時に実行できます。EPIC は、これらの並列機能の多くの制御をコンパイラに委ねます (したがって、名前に "Explicitly" (明示的) という言葉が盛り込まれています)。この結果、コンパイラは複雑になりますが、これらの機能をコンパイラで直接利用することが可能になります。

EPIC は将来の世代のプロセッサがさまざまなレベルの並列処理を利用できるように設計されています。将来の拡張性が考慮されており、コンパイラが生成するマシン・コードについては、(並列に実行可能な) 1 つのグループ内の命令数には制限がありません。

Intel Itanium プロセッサ・ファミリ向けのアセンブリ・プログラミングの詳細については、『Intel® Itanium® Architecture Assembly Language Reference Guide』を参照してください。以下の Web サイトで提供されています。

[http://developer.intel.com/software/products/opensource/tools1/to1\\_whte2.htm](http://developer.intel.com/software/products/opensource/tools1/to1_whte2.htm)

### 7.1.2 Alpha プロセッサ・ファミリの概要

Alpha は 64 ビットのロード/ストア RISC アーキテクチャであり，パフォーマンスに最も影響を与える 3 つの要素に特に重点を置いて設計されています。3 つの要素とは，クロック速度，複数のインストラクションの発行，および複数のプロセッサです。

Alpha の設計者たちは，最新の理論上の RISC アーキテクチャ設計要素を調査，分析し，ハイパフォーマンスな Alpha アーキテクチャを開発しました。

Alpha アーキテクチャは，特定のオペレーティング・システムやプログラミング言語に偏らないように設計されています。Alpha ではオペレーティング・システムとして OpenVMS Alpha，Tru64 UNIX，および Linux がサポートされ，これらのオペレーティング・システムで動作するアプリケーションに対して簡単なソフトウェア移行がサポートされています。

Alpha は 64 ビット・アーキテクチャとして設計されました。すべてのレジスタのサイズが 64 ビットであり，すべての操作が 64 ビット・レジスタ間で実行されます。当初は 32 ビット・アーキテクチャとして開発され，後で 64 ビットに拡張されたものではありません。

インストラクションは非常に単純です。すべてのインストラクションが 32 ビットです。メモリ操作はロードとストアのいずれかです。すべてのデータ操作がレジスタ間で行われます。

Alpha アーキテクチャには特殊なレジスタや条件コードがないため，同じ操作の複数のインスタンスをパイプライン化するのは簡単です。

レジスタまたはメモリに書き込む 1 つの命令と，同じ場所から読み取りを行う別の命令とは相互に調整をします。このため，同時に複数の命令を実行する CPU の実装が簡単にできます。

Alpha では，複数のインプリメンテーション間でバイナリ・レベルの互換性を容易に維持でき，複数インストラクション発行のインプリメンテーションで完全な速度を簡単に維持できます。たとえば，インプリメンテーション固有のパイプライン・タイミングによる危険性，ロードディレイ・スロット，および分岐ディレイ・スロットはありません。

Alpha アーキテクチャでは，LDBU インストラクションと STB インストラクションを使用して，レジスタおよびメモリ間でバイトの読み書きを行います (Alpha では，LDWU および STW インストラクションによるワード読み書きもサポートされます)。バイト・シフトとマスキングは，通常の 64 ビット・レジスタ間インストラクションによって実行されます (インストラクション・シーケンスが長くないように工夫されています)。

第2のプロセッサ(I/O デバイスを含む)から見ると、1つのプロセッサが発行した一連の読み取りと書き込みは、インプリメンテーションによって任意の順序に変更される可能性があります。このため、インプリメンテーションではマルチバンク・キャッシュ、バイパスされた書き込みバッファ、書き込みマージ、エラー発生時にリトライするパイプライン書き込みなどを使用できます。2つのアクセスの順序を厳密に維持しなければならない場合は、プログラムにメモリ・バリア命令を明示的に挿入できます。

基本的なマルチプロセッサ・インターロック機構はRISCスタイルの `load_locked`、`modify`、`store_conditional` シーケンスです。割り込みや例外、あるいは別のプロセッサからの妨害的な書き込みが発生せずに、シーケンスが実行されると、条件付きストアは正常終了します。割り込みや例外などが発生すると、ストア操作は失敗し、プログラムは最終的に元に戻って、シーケンスをリトライしなければなりません。このスタイルのインターロックは非常に高速のキャッシュにも対応できるため、複数のプロセッサが搭載されたシステムを構築するためのアーキテクチャとして、Alpha を特に魅力的なものにしています。

PALcode (privileged architecture library) は、特定の Alpha オペレーティング・システム・インプリメンテーションに固有のサブルーチンの集合です。これらのサブルーチンは、コンテキスト・スイッチング、割り込み、例外、およびメモリ管理のためのオペレーティング・システム・プリミティブを提供します。PALcode は、パーソナル・コンピュータで提供される BIOS ライブラリのようなものです。

詳細については、『Alpha Architecture Handbook』(注文番号 EC-QD2KC-TE) を参照してください。

---

## 7.2 エンディアンに関する検討事項

OpenVMS Alpha または OpenVMS VAX から OpenVMS I64 へのポータリングを行う場合、エンディアンに関する問題について考慮する必要はありません。どのプラットフォームでも、OpenVMS はデータの格納と操作でリトル・エンディアンを使用します。この章では、エンディアンが異なるハードウェア・アーキテクチャ間でポータリングを行う際に、一般に考慮しなければならない問題点について、わかりやすく説明しています。

エンディアンとは、データがストアされる方法のことであり、マルチバイト・データ・タイプで各バイトを配置する方法を定義します。エンディアンが重要なのは、データを書き込んだマシンと異なるエンディアンのマシンでバイナリ・データを読み込もうとすると、異なる結果になるからです。データベースをエンディアンの異なるアーキテクチャに移動しなければならない場合には、このことが特に重要な問題になります。

エンディアンには、ビッグ・エンディアン (順方向バイト順, つまり MSF (most significant first)) とリトル・エンディアン (逆方向バイト順, つまり LSF (least significant first)) の 2 種類があります。エンディアンについて考える場合は, 「big end in」と「little end in」という覚え方をすると便利です。図 7-1 と図 7-2 は, 2 つのバイト順方式を比較しています。

図 7-1 ビッグ・エンディアン・バイト順

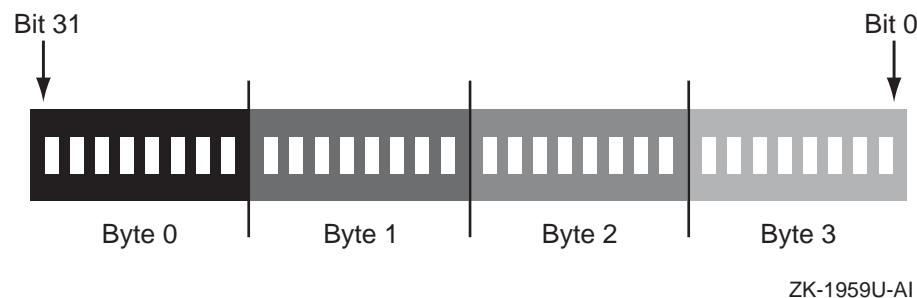
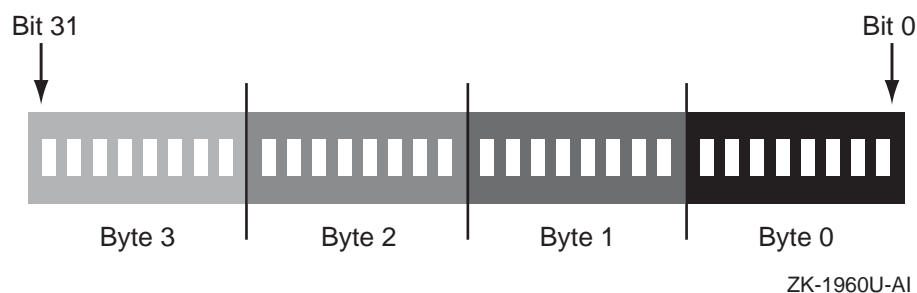


図 7-2 リトル・エンディアン・バイト順



Alpha マイクロプロセッサでは OpenVMS や Tru64 UNIX などのオペレーティング・システムでリトル・エンディアン環境を提供し, PA-RISC マイクロプロセッサでは HP-UX オペレーティング・システムでビッグ・エンディアン環境を提供しています。

OpenVMS などのリトル・エンディアン・オペレーティング・システムでは, リトル・エンド, つまり, 最下位有効バイトが最下位アドレスに格納されます。たとえば, 0x1234 という 16 進数は 0x34 0x12 としてメモリに格納されます。4 バイト値の場合も同様です。たとえば, 0x12345678 は 0x78 0x56 0x34 0x12 として格納されます。ビッグ・エンディアン・オペレーティング・システムではこれとは逆の順に格納されます。たとえば, 0x1234 は 0x12 0x34 としてメモリに格納されます。

4 バイト整数として格納されている 1025 (2 の 10 乗 + 1) という数値について考えてみましょう。メモリでは、この数値は以下のように表現されます。

リトル・エンディアン: 00000000 00000000 00000100 00000001  
ビッグ・エンディアン: 00000001 00000100 00000000 00000000

Intel Itanium プロセッサ・ファミリ・アーキテクチャでは、ビッグ・エンディアンとリトル・エンディアンの両方のメモリ・アドレッシングがサポートされます。Itanium プロセッサで動作する OpenVMS I64 と OpenVMS Alphaの間には、データの読み込みおよびデータ交換に関する問題はありません。Alpha または I64 システム上の OpenVMS と、PA-RISC または Itanium プロセッサ・ファミリの HP-UX の間でデータを交換する必要がある場合は、バイトの入れ換えが必要になることがあります。



---

## アプリケーション評価チェックリスト

計画およびポーティング作業の際には、このチェックリストを手引きとして利用してください。

## アプリケーション評価チェックリスト

### 全般的な情報

1. アプリケーションの目的は何ですか? (簡単に記入してください)

---

---

### 開発の履歴とプラン

2. a. アプリケーションは現在、他のオペレーティング・システムまたはハードウェア・アーキテクチャで動作していますか? ☐はい ☐いいえ
- b. 回答が「はい」の場合、アプリケーションは現在、最新の OpenVMS Alpha システムで動作していますか? ☐はい ☐いいえ

[アプリケーションがすでに複数のプラットフォームで動作している場合は、特定のプラットフォームに依存していない可能性が高いため、OpenVMS I64 へのポーティングは比較的簡単に行うことができると考えられます (しかし、アプリケーション・コードがプラットフォームごとに条件化されている場合は、多少の変更が必要になる可能性があります。詳細については、第 4.8.1 項を参照してください)。b の回答が「はい」の場合は、OpenVMS I64 へのアプリケーションのポーティングは比較的簡単に行うことができると考えられます。OpenVMS VAX で動作しているアプリケーションの場合は、最初に OpenVMS Alpha へのポーティングを行う必要があります (詳細については、『OpenVMS VAX から OpenVMS Alpha へのアプリケーションの移行』を参照してください)。]

3. 前回、アプリケーション環境をソースから完全に再コンパイルおよび再ビルドしたのはいつですか? \_\_\_\_\_
- a. アプリケーションの再ビルドは定期的に行っていますか? ☐はい ☐いいえ
- b. その頻度を記入してください。 \_\_\_\_\_

[あまり頻繁には再ビルドされていないアプリケーションの場合は、ポーティングの前に追加作業が必要になることがあります。たとえば、開発環境を変更すると、互換性の問題が発生することがあります。アプリケーションを移植する前に、最新バージョンの OpenVMS Alpha でビルドできるかどうか確認してください。]

4. アプリケーションについてよく理解している開発者が積極的にメンテナンス作業を行っていますか? ☐はい ☐いいえ

開発者の名前と連絡先を記入してください。

開発者	連絡先
_____	_____
_____	_____
_____	_____

5. a. 新しくビルドされたアプリケーションの動作は、どのような方法でテストまたは検証していますか?

---

---

- b. 最適化に役立つパフォーマンス評価ツールを使用していますか? ☐はい ☐いいえ



c. 回答が「はい」の場合は、使用しているツールとバージョン番号を記入してください。

ツール	バージョン
_____	_____
_____	_____
_____	_____

d. 使用しているソース・コード構成管理ツールとバージョン番号を記入してください。

ツール	バージョン
_____	_____
_____	_____
_____	_____

6. 運用システムとは別に、開発およびテストのための環境を用意していますか？ ☐はい ☐いいえ

7. 新バージョンのアプリケーションを運用環境に導入する際の手順を記入してください。

\_\_\_\_\_

\_\_\_\_\_

8. OpenVMS I64 へのポータリング完了後、アプリケーションの今後の開発計画はどのようになっていますか。

a. 開発終了	<input type="checkbox"/> はい	<input type="checkbox"/> いいえ
b. 保守リリースのみ	<input type="checkbox"/> はい	<input type="checkbox"/> いいえ
c. 機能の追加や変更	<input type="checkbox"/> はい	<input type="checkbox"/> いいえ
d. OpenVMS I64 と OpenVMS Alpha のソースを個別に管理	<input type="checkbox"/> はい	<input type="checkbox"/> いいえ

[a の回答が「はい」の場合は、利用するアプリケーションのバイナリ・トランスレーションをお勧めします。 b または c の回答が「はい」の場合は、バイナリ・トランスレーションも可能ですが、アプリケーションの再コンパイルと再リンクを行う場合の利点を評価する価値があります。 OpenVMS I64 と OpenVMS Alpha のソースを個別に管理する場合、つまり、 d の回答が「はい」の場合は、相互運用性と整合性に関する問題点を考慮する必要があります。特に、同じアプリケーションの複数のバージョンが同じデータベースにアクセスする場合は、この検討が必要です。]

#### アプリケーションの構成

移植するアプリケーションのサイズとコンポーネントについて検討します。

9. アプリケーションのサイズを記入してください。

モジュールの数を記入してください。

コードの行数、ディスク・ブロック数、または容量 (MB) を記入してください。

必要なディスク容量を記入してください。

[この情報は、ポータリングに必要な作業量とリソースの「サイズ」を判断するのに役立ちます。最も役立つ情報はコードのディスク・ブロック数です。]

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

## アプリケーション評価チェックリスト

10. a. アプリケーションを構成するすべてのソース・ファイルにアクセスできますか? ☐はい ☐いいえ

- b. HP サービスの利用を検討している場合、HP のスタッフがこれらのソース・ファイルおよびビルド・プロセスにアクセスすることが可能ですか? ☐はい ☐いいえ

[a の回答が「いいえ」の場合、不足しているソース・ファイルについては、バイナリ・トランスレーションによるポーティングしか手段がありません。ユーザ・モードで実行される OpenVMS Alpha イメージを OpenVMS I64 イメージに変換してください。

OpenVMS VAX イメージを移植する場合は、最初に OpenVMS Alpha イメージに変換してください。OpenVMS I64 イメージに直接変換することはできません。

b の回答が「はい」の場合は、広い範囲で HP ポーティング・サービスを利用できます。]

11. a. アプリケーションを作成するのに使用した言語を記入してください (複数の言語を使用している場合は、それぞれの割合も記入してください)。

言語	パーセンテージ
_____	_____
_____	_____
_____	_____

- b. OpenVMS Alpha で MACRO-64, PL/I, Ada 83, Fortran 77 を使用していますか? ☐はい ☐いいえ

[OpenVMS Alpha で MACRO-64 を使用している場合は、OpenVMS I64 で MACRO-64 コンパイラが提供されないため、コードを書き直す必要があります。可能であれば、ドキュメントに記載されているシステム・インタフェースを使用して、コードを高級言語で書き直すことをお勧めします。アセンブリ言語で記述しなければならないマシン固有のコードの場合は、MACRO-64 コードを OpenVMS I64 アセンブリ・コードで書き直す必要があります。

同様に、PL/I, Fortran 77, および Ada 83 も OpenVMS I64 ではサポートされません。PL/I で作成されたコードがアプリケーションに含まれている場合は、C や C++ などの別の言語で書き直すことをお勧めします。Ada 83 で書かれているコードは Ada 95 に、Fortran 77 で書かれているコードは Fortran 90 に更新してください。コードを分析して、新しいコンパイラでコードの変更が必要かどうかを確認します。

一般に、OpenVMS I64 でコンパイラが提供されていない場合は、バイナリ・トランスレータを使用するか、または別の言語でコードを書き直す必要があります。

コンパイラとトランスレータが提供されているかどうかについては、第 5 章と第 6 章を参照してください。]

12. アプリケーションのドキュメントが存在しますか? ☐はい ☐いいえ

[回答が「はい」の場合は、アプリケーションの変更が必要になったときに、ドキュメントの更新も必要になることに注意してください。]

外部依存性

アプリケーションの開発、テスト、および実行に必要なシステム構成、環境、ソフトウェアについて検討します。

13. アプリケーションの開発環境を設定するのに必要なシステム構成 (CPU、メモリ、ディスク) を記入してください。

---



---

[この情報は、ポーティングに必要なリソースを見積もるのに役立ちます。]

14. アプリケーションの典型的なユーザ環境 (インストール検証プロシージャ、リグレーション・テスト、ベンチマーク、作業負荷など) を設定するのに必要なシステム構成 (CPU、メモリ、ディスク) を記入してください。

---



---

[この情報は、環境全体が OpenVMS I64 で提供されるかどうかを判断するのに役立ちます。]

15. アプリケーションは特殊なハードウェアに依存していますか? ☐はい ☐いいえ

[この情報は、ハードウェアが OpenVMS I64 で使用できるかどうか、およびアプリケーションにハードウェア固有のコードが含まれているかどうかを判断するのに役立ちます。]

16. アプリケーションが現在動作している OpenVMS Alpha のバージョンを記入してください。 

---

アプリケーションは OpenVMS Alpha Version 7.3-2 または 7.3-1 で動作していますか? ☐はい ☐いいえ

[アプリケーションを OpenVMS I64 へ移植する前に、最新バージョンの OpenVMS Alpha で実行できるようにすることをお勧めします。]

17. アプリケーションでレイヤード・ソフトウェアや他社製品を実行することが必要ですか?

a. HP 製品 (コンパイラ・ランタイム・ライブラリ以外): ☐はい ☐いいえ

HP 製レイヤード・ソフトウェアとそのバージョンを記入してください。

HP 製品	バージョン
<hr/>	<hr/>
<hr/>	<hr/>
<hr/>	<hr/>

b. 他社製品: ☐はい ☐いいえ

他社製品とそのバージョンを記入してください。

他社製品	バージョン
<hr/>	<hr/>
<hr/>	<hr/>
<hr/>	<hr/>

## アプリケーション評価チェックリスト

[aの回答が「はい」で、HP 製レイヤード・ソフトウェアが OpenVMS I64 で提供されるかどうか分からない場合は、HP のサポート担当者にお問い合わせください。ソフトウェアのポーティングに対する HP の取り組みについては、第 3.4 節を参照してください。b の回答が「はい」の場合は、その製品を提供する会社にお問い合わせください。]

18. a. アプリケーション用のリグレッション・テスト・ツールを用意していますか? ☐はい ☐いいえ

b. 回答が「はい」の場合は、HP Digital Test Manager など、特定のソフトウェア製品が必要ですか? ☐はい ☐いいえ

[aの回答が「はい」の場合は、リグレッション・テスト・ツールのポーティングを検討する必要があります。bの回答が「はい」の場合は、本リリースの OpenVMS で HP Digital Test Manager が DECset 製品セットの一部として提供されています。テスト用に他のツールが必要な場合は、HP のサポート担当者にお問い合わせください。また、第 5 章も参照してください。運用環境でリグレッションをチェックするために、適切なテスト・プロセスを使用することをお勧めします。]

### OpenVMS Alpha アーキテクチャへの依存性

OpenVMS Alpha と OpenVMS I64 との相違点について検討します。以下の質問は、最も重要な相違点を明らかにするのに役立ちます。これらの相違点によって必要になるアプリケーション・コンポーネントの変更点を検討してください。OpenVMS Alpha アーキテクチャに特別に依存しているユーザ作成コードやアセンブリ・コードは書き直す必要があります。

19. アプリケーションで OpenVMS VAX 浮動小数点データ・タイプを使用していますか? ☐はい ☐いいえ

[回答が「はい」の場合は、OpenVMS I64 のコンパイラのデフォルトが IEEE 浮動小数点データ・タイプであることに注意してください。OpenVMS I64 のコンパイラでは、OpenVMS VAX 浮動小数点データ・タイプから IEEE 浮動小数点データ・タイプに自動的に変換することで、VAX 浮動小数点形式をサポートしています。その結果、わずかな精度の差が発生することがあります。さらに、アプリケーションでどの程度の浮動小数点演算が実行されるかに応じて、実行時のパフォーマンスが少し低下します (変換せずに、IEEE 浮動小数点データ・タイプを直接使用した場合と比較して)。浮動小数点データ・タイプの詳細については、第 4.8.4 項を参照してください。]

20. データは自然な境界に配置されていますか? ☐はい ☐いいえ

[データが自然な境界に配置されていない場合は、データ参照のパフォーマンスを向上するために、データが自然な境界に配置されるようにしてください。自然な境界に配置されていないデータの場合、データ参照のパフォーマンスが大幅に低下します。特定の状況では、データを自然な境界に配置する必要があります。データ構造とアラインメントの詳細については、第 4.8.7 項と『OpenVMS Programming Concepts Manual』を参照してください。]

21. アプリケーションで呼び出す OpenVMS システム・サービスは、次の操作を行いますか?

a. ワーキング・セットを変更しますか? ☐はい ☐いいえ

- b. SYSSGOTO\_UNWIND を使用してプログラムの制御の流れを変更しますか? ☐はい ☐いいえ

[a の回答が「はい」の場合は、SYSSLKWSET および SYSSLKWSET\_64 システム・サービスの呼び出しに必要な入力パラメータが正しく指定されているかどうか確認してください。

b の回答が「はい」の場合は、SYSSGOTO\_UNWIND の呼び出しを SYSSGOTO\_UNWIND\_64 に変更してください。OpenVMS I64 では SYSSGOTO\_UNWIND はサポートされません。

詳細については、第 4 章を参照してください。]

22. a. アプリケーションで複数の協調動作プロセスを使用していますか? ☐はい ☐いいえ  
回答が「はい」の場合:

b. プロセスの数を記入してください。 \_\_\_\_\_

c. 使用しているプロセス間通信の方法を記入してください。 \_\_\_\_\_

- ☐ \$CRMPSC      ☐ メールボックス      ☐ SCS      ☐ その他  
☐ DLM      ☐ SHM, IPC      ☐ SMGS      ☐ STRS

d. グローバル・セクション (\$CRMPSC) を使用して他のプロセスとデータを共有している場合、データ・アクセスの同期をとる方法を記入してください。 \_\_\_\_\_

[この情報は、明示的な同期化を使用する必要があるかどうかと、アプリケーションの要素間で同期化を保証するのに必要な作業レベルを判断するのに役立ちます。一般に、高度な同期化方式を使用すると、アプリケーションのポーティングは最も簡単に行うことができます。]

23. アプリケーションは現在、マルチプロセッサ (SMP) 環境で動作していますか? ☐はい ☐いいえ

[回答が「はい」の場合は、アプリケーションですでに適切なプロセス間同期化方式が使用されていると考えられます。]

24. アプリケーションで AST (非同期システム・トラップ) 機能を使用していますか? ☐はい ☐いいえ

[回答が「はい」の場合は、AST とメイン・プロセスがプロセス空間でデータへのアクセスを共有しているかどうかを判断する必要があります。共有している場合は、このようなアクセスは明示的に同期化しなければならないことがあります。これは Alpha 上のアプリケーションにも同じく当てはまる点に注意してください。AST についての詳細は、『OpenVMS Programming Concepts Manual』を参照してください。]

25. a. アプリケーションに条件ハンドラが含まれていますか? ☐はい ☐いいえ  
b. ハンドラでメカニズム・アレイ内のデータを変更しますか? ☐はい ☐いいえ

[OpenVMS Alpha と OpenVMS I64 のメカニズム・アレイの形式は大きく異なります。メカニズム・アレイ内のレジスタ値を変更するアプリケーションは必ず変更してください。詳細は、『OpenVMS Guide to Upgrading Privileged-Code Applications』を参照してください。]

## アプリケーション評価チェックリスト

26. a. アプリケーションは特権モードで動作するか、または SYSSBASE\_IMAGE に対してリンクされていますか? ☐はい ☐いいえ  
 回答が「はい」の場合は、その理由を記入してください。  
 b. アプリケーションは OpenVMS の内部データ構造またはインタフェースに依存していますか? ☐はい ☐いいえ  
 [OpenVMS エグゼクティブに対してリンクされているアプリケーションや特権モードで動作するアプリケーションの場合は、ポーティングで追加作業が必要になることがあります。SYSSBASE\_IMAGE でドキュメントに記載されていないインタフェースは、OpenVMS I64 で変更されている可能性があります。  
 OpenVMS の内部データ構造定義 (SYSSLIBRARY:LIB.INCLUDE, SYSSLIBRARY:LIB.L32, SYSSLIBRARY:LIB.MLB, SYSSLIBRARY:LIB.R64, SYSSLIBRARY:LIB.REQ, および SYSSLIBRARY:SYSSLIB\_C.TLB に定義) に依存するアプリケーションの場合も、一部の内部データ構造が OpenVMS I64 で変更されている可能性があるため、ポーティングで追加作業が必要になることがあります。]
27. アプリケーションで接続/割り込み機能 (connect-to-interrupt) を使用していますか? ☐はい ☐いいえ  
 回答が「はい」の場合は、その機能を記述してください。  
 [接続/割り込み機能は OpenVMS I64 システムではサポートされません。この機能が必要な場合は、HP の担当者にお問い合わせください。]
28. アプリケーションでマシン・インストラクションを生成または変更しますか? ☐はい ☐いいえ  
 [OpenVMS I64 では、インストラクション・ストリームに書き込まれたインストラクションが正しく実行されることを保証するには、細心の注意を払う必要があります。特定のマシン・インストラクションを取り扱うコードは、書き直す必要があります。]
29. OpenVMS Alpha アーキテクチャに特に依存しているその他のユーザ作成コードやアセンブラ・コードがアプリケーションに含まれていますか? ☐はい ☐いいえ  
 [回答が「はい」の場合は、このようなコードは書き直してください。できれば、アーキテクチャに依存しないように書き直してください。]
30. アプリケーションのソース・コードやビルド・ファイルに、条件付きステートメントや VAX システムまたは OpenVMS Alpha システムで動作することを前提としたロジックが含まれていますか? ☐はい ☐いいえ  
 [回答が「はい」の場合は、条件付きディレクティブを多少変更する必要があります。たとえば、「IF ALPHA」を「IF ALPHA OR IPF」に変更したり、「IF NOT ALPHA」を「IF VAX」に変更したりします。詳細については、第 4.8.1 項を参照してください。]
31. アプリケーションの中でパフォーマンスに最も影響を与えやすい要素を記述してください。たとえば、I/O、浮動小数点、メモリ、リアルタイム (つまり、割り込み待ち時間) などの要素です。

[この情報は、アプリケーションの各要素に対する作業の優先順位を判断するのに役立ちます。この情報をもとに、HP はお客様にとって最も意味のあるパフォーマンス強化手段を計画することができます。]

32. アプリケーションで OpenVMS Alpha 呼び出し規則ルーチンを使用していますか? ☐はい ☐いいえ

[回答が「はい」の場合は、LIBICB という呼び出し規則データ構造を使用するアプリケーションや、LIB\$GET\_CURRENT\_INVO\_CONTEXT、LIB\$GET\_PREVIOUS\_INVO\_CONTEXT、LIB\$PUT\_INVO\_CONTEXT、および LIB\$GET\_INVO\_HANDLE などの呼び出し規則ルーチンを使用するアプリケーションは変更する必要があります。OpenVMS I64 では、LIBICB データ構造が変更された結果、これらのルーチンの名前も変更されました。さらに、呼び出し規則も変更されています。OpenVMS Alpha 呼び出し規則に関する情報を直接使用するアプリケーションは、OpenVMS I64 では変更する必要があります。OpenVMS Alpha と OpenVMS I64 の呼び出し規則の相違点の詳細については、第 2.1.1 項を参照してください。OpenVMS の呼び出し規則の詳細については、『OpenVMS Calling Standard』を参照してください。]

33. アプリケーションで非標準ルーチン・リンケージ宣言を使用していますか? ☐はい ☐いいえ

[回答が「はい」の場合は、非標準リンケージを含むアプリケーションは OpenVMS Alpha の呼び出し規則に従っているため、OpenVMS I64 の呼び出し規則に準拠するように変更する必要があります。]

34. a. アプリケーションはオブジェクト・ファイルの形式または内容に依存していますか? ☐はい ☐いいえ  
b. アプリケーションは実行イメージの形式または内容に依存していますか? ☐はい ☐いいえ  
c. アプリケーションはイメージのデバッグ・シンボル・テーブルの内容に依存していますか? ☐はい ☐いいえ

[いずれかの回答が「はい」の場合は、これらのイメージおよびオブジェクト・ファイルのデータ構造に依存しているアプリケーションは変更する必要があります。OpenVMS I64 では、ELF という業界標準のオブジェクト・ファイルおよびイメージ・ファイルのレイアウトが採用されています。また、DWARF という業界標準のデバッグ・シンボル・テーブルの形式も採用されています。OpenVMS Alpha のオブジェクト・ファイル、イメージ・ファイル、またはデバッグ・シンボル・テーブルの形式に依存しているアプリケーションは、変更する必要があります。詳細については、第 4.8.3.4 項を参照してください。]

35. アプリケーションで C の asm ステートメントを使用していますか? ☐はい ☐いいえ

[C の asm ステートメントを使用しているアプリケーションは Alpha インストラクションに依存するので、回答が「はい」の場合は、変更する必要があります。OpenVMS I64 向けの HP C コンパイラでは、asm ステートメントを使用する必要性をできるだけ少なくするために、多くの新しい組み込み関数が追加されています。]

## アプリケーション評価チェックリスト

36. アプリケーションで BLISS レジスタ・ビルトインを使用していますか? ☐はい ☐いいえ
- [回答が「はい」の場合は、BLISS の BUILTIN ステートメントを使用して組み込みレジスタ名を宣言するアプリケーションは変更する必要があります。]

### アプリケーションのポーティング

評価と計画段階が終了したら、実際のポーティング作業を行います。

37. OpenVMS Alpha システムを、提供されている最新バージョンにアップグレードしましたか? ☐はい ☐いいえ
- [回答が「いいえ」の場合は、アプリケーションのポーティングを開始する前に、OpenVMS Alpha システムをアップグレードしてください。]
38. a. 最新バージョンのコンパイラを使用して、最新バージョンの OpenVMS Alpha でアプリケーションをコンパイルしましたか? ☐はい ☐いいえ
- b. コンパイル時に検出された問題を修正しましたか? ☐はい ☐いいえ
- [回答が「いいえ」の場合は、最新バージョンのコンパイラを使用して、最新バージョンの OpenVMS Alpha でアプリケーションをコンパイルしてください。a の回答が「はい」であっても、コンパイル後にコードの一部を変更している場合は、再度コンパイルしてください。この段階が終了し、このチェックリストでこれまで詳しく検討した他のすべての項目に回答し、第 4 章から第 7 章までの検討事項にも対応したら、アプリケーションは OpenVMS I64 にポーティング可能な状態になっています。]
39. アプリケーション・ソース・モジュールとすべての関連コードを OpenVMS I64 システムにコピーしてください。モジュールと関連コードを OpenVMS I64 にコピーしましたか? ☐はい ☐いいえ
40. OpenVMS I64 システムでアプリケーションをコンパイルおよびリンクして、実行します。  
アプリケーションをコンパイルおよびリンクして、実行していますか? ☐はい ☐いいえ
- [詳細については、第 5 章を参照してください。]
41. OpenVMS I64 システムで単体テストを行ってください。  
単体テストは完了しましたか? ☐はい ☐いいえ
- [詳細については、第 5 章を参照してください。]
42. OpenVMS I64 システムでシステム・テストを行ってください。  
システム・テストは完了しましたか? ☐はい ☐いいえ
- [詳細については、第 5 章を参照してください。]
43. OpenVMS I64 システムでリグレッション・テストを行ってください。  
リグレッション・テストは完了しましたか? ☐はい ☐いいえ
- [詳細については、第 5 章を参照してください。]



44. テストは成功しましたか?

☐はい

☐いいえ

[回答が「いいえ」の場合は、問題を解決した後、アプリケーションのコンパイル、リンク、テストを再度行います。詳細については、第5章を参照してください。]



---

## サポート対象外のレイヤード・プロダクト

いくつかのレイヤード・プロダクトについては OpenVMS I64 にポーティングする計画がありません。表 B-1 に、そのようなレイヤード・プロダクトと、弊社が推奨する代替製品を示します。

表 B-1 OpenVMS I64 にポーティングされないレイヤード・プロダクト

製品名	推奨する代替製品名
BASEstar Classic	BASEstar Open
HP Ada Compiler for OpenVMS Alpha Systems	GNAT Ada for OpenVMS Integrity Servers—Ada Core Technologies 社製
Pathworks 32	Advanced Server



## アプリケーション固有のスタック切り換えコードの I64 へのポーティング

1 つのプロセスのコンテキストの内部で複数のタスクを実行するために、スタックを切り換える小さなプライベート・スレッド・パッケージが、多くのアプリケーションでサポートされています。これらのパッケージの多くは、アセンブリ言語 (VAX では Macro-32, Alpha では Macro-64) で作成された 1 つ以上の小さなルーチンを使用して、スタック・ポインタを操作することで、単一プロセスの内部でコンテキストの切り換えを行います。通常、タスクのストールと再起動を行う機能も必要です。置き換えられるスタックは、要求されたタスクにとって適切なモードであれば、どのモードでもかまいません。

I64 では、スタックの切り換えははるかに困難です。I64 アーキテクチャには 2 つのスタックがあります。メモリ・スタックとレジスタ・スタック・エンジン・バッキング・ストア (一般には RSE スタックや単にレジスタ・スタックとも呼びます) です。RSE スタックはハードウェアで管理され、アーキテクチャはスタックの非同期操作をサポートする機能を提供します。また、I64 アーキテクチャには、コンテキストの切り換えの前後で維持しなければならない、多くの専用ハードウェア・レジスタ (制御レジスタおよびアプリケーション・レジスタ) もあります。さらに、IAS アセンブラは OpenVMS ディストリビューションには含まれていません。

これらのタイプのアプリケーションに対応するために、OpenVMS では KP サービスというシステム・ルーチン・セットが提供されています。KP サービスはもともと高級言語で作成されたデバイス・ドライバをサポートするように作成されていますが、どのモードでも、どの IPL でも動作するように拡張されています。KP モデルがすべてのプライベート・スタック切り換えコードの要件を満たすわけではありませんが、このモデルに移行することを強くお勧めします。

KP サービスは Alpha と I64 の両方で提供されているため、共通のコードを使用できます。インプリメントの面ではいくつかの相違点がありますが、一方のアーキテクチャでのみ有効なオプションは、可能な限り、もう一方のアーキテクチャでは無視されるようになっています。

---

## C.1 KP サービスの概要

KP モデルについて説明する場合、任意のストール機能と再開機能を備えた共通ルーチン・モデルとして説明するのが最も正確です。プロセス・コンテキストまたはシステム・コンテキストで動作するコード・ストリームは、既存のスタック・コンテキストを妨害せずに、同じモードで他のストリーム (KP ルーチン) を起動しようとしします。そのとき、KP ルーチンは独自の異なるコンテキストを維持しながら、必要に応じてストールと再開を実行できます。KP ルーチンの再開は、KP ルーチンを起動した元のコード・ストリームと非同期に行うことができます。

KP ルーチンは、プライベート・スタック (I64 では 2 つ 1 組のスタック) で動作するルーチンであると考えると最も簡単です。ルーチンは、ストールおよび再開することができます。コンパイラにとっては、起動するためのルーチン、ストールするためのルーチン、再開するためのルーチンは、単に OpenVMS 呼び出し規則に準拠した外部プロシージャ呼び出しに過ぎません。状態の保存とスタック切り換えは完全に KP ルーチンのコンテキストの内部で行われます。コード・ストリームが制御を放棄すると、再開ポイントでは、制御を切り換えた呼び出しが完了したかのように見えます。

ベース・サポートには、必要なスタックとデータ構造の割り当てを支援するルーチン、および KP ルーチンの起動、ストール、再起動、終了を行うルーチンが含まれています。

KP ルーチンは別の KP ルーチンを起動できます。

基本的な KP サポート・ルーチンは次のとおりです。

- EXESKP\_START— KP ルーチンを起動します。
- EXESKP\_STALL\_GENERAL— 現在のルーチンをストールし、EXESKP\_START または EXESKP\_RESTART を最後に呼び出したルーチンに戻ります。
- EXESKP\_RESTART— EXESKP\_STALL\_GENERAL の呼び出しによってストールされた KP ルーチンを再起動します。
- EXESKP\_END— KP ルーチンを終了します。

各ルーチンと必要なデータ構造については、第 C.2.2 項を参照してください。

### C.1.1 用語

メモリ・スタックとは、スタック・ポインタ・レジスタを参照するスタックです。Alpha システムには、メモリ・スタックしかありません。

レジスタ・スタックとは、レジスタ・スタック・エンジン (RSE) バッキング・ストアの略称です。I64 アーキテクチャには 96 個のスタック・レジスタがあり、ルーチン間で引数を受け渡したり、ルーチン内でのローカル・ストレージとして使用したり

できます。ハードウェアはスタック・レジスタの状態を維持管理し、必要に応じてレジスタをバッキング・ストアに書き出します。

KP ルーチンとは、1 つ以上のプライベート・スタックで動作する一連のコードです。EXESKP\_START の呼び出しによって起動され、EXESKP\_END の明示的な呼び出しまたは暗黙の呼び出しによって終了します。

KPBとは、1 つ以上のスタックを記述し、KP ルーチンの状態およびコンテキスト情報を保持するデータ構造です。

EXESKP\_START を呼び出すことで KP ルーチンをアクティブにするために KPB が使用されると、KPB は有効になります。EXESKP\_END は KPB を無効としてマークします。KPB の初期状態は無効です。

EXESKP\_START を呼び出すことで KP ルーチンが起動されるか、EXESKP\_RESTART を呼び出すことで再起動されると、KPB はアクティブになります。EXESKP\_STALL\_GENERAL と EXESKP\_END は、KPB を非アクティブとしてマークします。KPB の初期状態は非アクティブです。

### C.1.2 スタックとデータ構造

I64 では、3 つのメモリ領域が KP ルーチンに関連付けられています。3 つのメモリ領域とは、メモリ・スタック、RSE スタック、KPB です。KPB はこれらの 3 つのメモリ領域全部を結びつけるデータ構造です。Alpha には RSE スタックがないため、RSE スタックに関連するパラメータとフィールドはすべて無視されます。

KPB とスタックはそれぞれ、適切な領域から割り当てる必要があり、KP ルーチンが実行されるモードと対応するように、適切なモード、保護、オーナーシップを選択する必要があります。既存のアプリケーションをポータリングする場合、アプリケーションはあらかじめ適切なメモリ・スタックを割り当てていると考えられます。既存のメモリ・スタック割り当てルーチンは、KP API に適応できます。以前のアーキテクチャと同様に、メモリ・スタックは最上位アドレス (スタックのベース) から最下位アドレスへの向きでアクセスされます。

レジスタ・スタック・エンジンはまったく新しい概念であり、以前の 32 ビット・コードに依存しないため、RSE スタックは通常、64 ビット空間から割り当てられます。一般的なアプリケーション・ニーズの大部分に対応できる多くの割り当てルーチンが提供されています。RSE スタックは最下位アドレスから最上位アドレスへの向きにアクセスされます。

表 C-1 は、アプリケーションのモードとスコープ別に、割り当てのガイドラインを示しています。

## アプリケーション固有のスタック切り換えコードの I64 へのポータリング

### C.1 KP サービスの概要

表 C-1 モードおよびスコープ別割り当てガイドライン

モード — スコープ <sup>1</sup>	KPB	メモリ・スタック	レジスタ・スタック
カーネル — システム EXESKP_ALLOC_ KPB <sup>2</sup>	非ページング・プール KW EXESALONONPAGED	S0/S1 KW EXESKP_ALLOC_ MEM_STACK	S2 KW EXESKP_ALLOC_ RSE_STACK <sup>3</sup>
カーネル — プロセス	非ページング・プール または P1 KW EXESALONONPAGED または EXESALOP1PROC	P1— パーマネント KW \$CREATE_REGION /\$CRETVA	P2— パーマネント KW EXESKP_ALLOC_ RSE_STACK_P2
カーネル — イメージ	P1 KW EXESALOP1IMAG	P1— 非パーマネント KW \$CREATE_REGION /\$CRETVA	P2— 非パーマネント KW \$CREATE_REGION /\$CRETVA
Exec— プロセス	P1 EW EXESALOP1PROC	P1— パーマネント EW \$CREATE_REGION /\$CRETVA	P2— パーマネント EW EXESKP_ALLOC_ RSE_STACK_P2
Exec— イメージ	P1 EW EXESALOP1IMAG	P1— 非パーマネント EW \$CREATE_REGION /\$CRETVA	P2— 非パーマネント EW \$CREATE_REGION /\$CRETVA
Super— プロセス	P1 SW EXESALOP1PROC	P1— パーマネント SW \$CREATE_REGION /\$CRETVA	P2— パーマネント SW EXESKP_ALLOC_ RSE_STACK_P2
Super— イメージ	P1 SW EXESALOP1IMAG	P1— 非パーマネント SW \$CREATE_REGION /\$CRETVA	P2— 非パーマネント SW \$CREATE_REGION /\$CRETVA
ユーザ — イメージ	P0 UW Heap/Malloc /LIB\$GET_VM	P0— 非パーマネント <sup>4</sup> UW EXESKP_ALLOC_ MEM_STACK_USER	P2— 非パーマネント UW EXESKP_ALLOC_ RSE_STACK_P2

<sup>1</sup> イメージ・スコープはイメージの終了時に終了します。プロセス・スコープはプロセスの終了時に終了し、イメージがランダウンしても継続します。システム・スコープはプロセス・コンテキストを必要としません。

<sup>2</sup> EXESKP\_ALLOC\_KPB は、1 回の呼び出しでカーネル・モード KPB とカーネル・モード・スタックを割り当てます。

<sup>3</sup> EXESKP\_ALLOC\_RSE\_STACK\_P2 はパーマネント領域を作成します。

<sup>4</sup> パーマネント・メモリ領域をユーザ・モードで作成することはできません。

### C.1.3 KPB

KPB とは、KP ルーチンを起動するコード・ストリームと KP ルーチンの間で必要なコンテキストを保持するために使用されるデータ構造です。KPB は半透過的です。一部のフィールドはアプリケーションで管理され、一部は KP ルーチンで管理され、一部は両方で共有されます。KP ルーチンでは、KPB は割り当て時に 0 に初期化されるので、0 以外のフィールドには適切なデータが格納されていると解釈されます。



KPB の構造定義は、Macro-32 では \$KPBDEF マクロで、C では KPBDEF.H で定義されます。KPB 定義はシステムの内部的な定義であると考えられるため、LIB.MLB と SYS\$LIB\_C.TLB で提供されます。BLISS の場合は、LIB.REQ または LIB.L32/LIB.L64 に KPB 定義が格納されています。

KPB は多くの領域またはサブ構造で構成される可変長構造です。すべての領域が必須なわけではありません。領域は以下のとおりです。

- ベース領域
- スケジューリング領域
- VEST 領域
- スピンロック領域
- デバッグ領域
- ユーザ・パラメータ領域

ベース領域は必須です。この領域には、標準構造ヘッダ、スタック・サイズとベース・アドレス、フラグ (他にどの領域が存在するかの情報を含む)、非アクティブ・コード・ストリームのメモリ・スタック・ポインタ、他の領域を参照するポインタ、ベース KP ルーチンが必要とする追加フィールドが格納されます。

スケジューリング領域には、ストール、再起動、終了を取り扱うルーチンを参照するポインタ、フォーク・ブロック、追加フォーク・ブロックを参照するポインタが格納されます。終了ルーチンを除き、他のルーチンの大部分は、高い IPL で実行されるドライバ・レベルのコードでのみ必要とされます。EXESKP\_USER\_ALLOC\_KPB を呼び出すルーチンは、割り当てられたメモリの必要なクリーンアップを実行する終了ルーチンを指定する必要があります。

VEST 領域とスピンロック領域は、主にドライバ・コードで使用されます。

デバッグ領域は、ドライバ・サポート・ルーチンでインプリメントされる、制限されたトレース機能を提供します。

ユーザ・パラメータ領域は、単に他の領域に連続的に割り当てられる未定義ストレージです。アプリケーションはそれぞれの要件に応じて、このメモリを自由に使用できます。

#### C.1.4 提供される KPB 割り当てルーチン

OpenVMS オペレーティング・システムでは、KPB および関連スタックを割り当てるために、2 つの標準的な割り当てルーチンを提供しています。Alpha で提供されていたカーネル・モードのドライバ・レベル・インタフェースは、そのまま変更されずに提供されるため、KP インタフェースを使用するデバイス・ドライバは、この部分に関してソースを変更する必要がありません。さらに、モードに依存しないルーチンも提供されます。モードに依存しないルーチンは、アプリケーションで指定されたル

ーチン呼び出して、KPB と各スタックを割り当てます。大部分の新規アプリケーションや、KP API にポーティングされるアプリケーションは、モードに依存しないルーチンを使用します。

カーネル・モード・ルーチンも、モードに依存しないルーチンも、KPB を初期化します。提供されるすべてのルーチンの C プロトタイプは、ヘッダ・ファイル EXE\_ROUTINES.H に格納されています。

### C.1.5 カーネル・モードの割り当て

カーネル・モードの割り当ての形式は次のとおりです。

EXE\$KP\_ALLOCATE\_KPB kpb, stack\_size, flags, param\_size

C プロトタイプ

```
status = EXE$KP_ALLOCATE_KPB( KPB_PPS kpb,  
    int stack_size,  
    int flags,  
    int param_size)
```

カーネル・モードでのみ使用します。このルーチンのプロトタイプは、元の Alpha ルーチンと同じです。

I64 では、RSE スタック・サイズ=メモリ・スタック・サイズです。

---

#### 注意

---

スタック・サイズはバイト数で表します。

---

#### パラメータ

- KPB— 割り当てられたデータ構造のアドレスが返されるロングワードのアドレス。OpenVMS API の規則では、32 ビット・ポインタは 32 ビット・アドレス (struct KPB を参照するショート・ポインタを参照するショート・ポインタ) 渡しされます。
- STACK\_SIZE— 割り当てられるスタックのサイズをバイト数で示す 32 ビット値。このパラメータは値渡しされます。渡される値は、ハードウェア・ページ・サイズの倍数に切り上げられます。どの場合も、割り当てられる最小ページ・サイズが SYSGEN パラメータ KSTACKPAGES より小さくなることはありません。スタックはページ境界で割り当てられ、両端にはマッピングされないガード・ページが配置されます。メモリ・スタックは 32 ビット S0/S1 アドレス空間に割り当てられます。

I64 システムでは、RSE スタックはメモリ・スタックと同じサイズに設定され、64 ビット S2 アドレス空間から割り当てられます。

- **FLAGS**— フラグのロングワード・ビットマスク。表 C-2 はこのパラメータに対して定義されているフラグを示しています。

表 C-2 カーネル・モード割り当てフラグ

フラグ	説明
KP\$M_VEST	OpenVMS システム KPB。一般に、このフラグは設定する必要があります。
KP\$M_SPLOCK	KPB の内部にスピンロック領域を割り当てます。
KP\$M_DEBUG	KPB の内部にデバッグ領域を割り当てます。
KP\$M_DEALLOC_AT_END	カーネル・プロセス・ルーチンの終了時に、KPB の割り当てを自動的に解除しなければならないことを指定します。
KP\$M_SAVE_FP (IA64 のみ)	汎用レジスタだけでなく、浮動小数点コンテキストも保存します。I64 での整数乗算や除算などの特定の演算は、浮動小数点演算を使用してインプリメントできます。これらの演算では、最小浮動小数点レジスタ・セットを使用しますが、定義では、これらは保存されるレジスタ・セットに含まれていません。アプリケーションで最小浮動小数点レジスタ・セットだけを使用する場合は、このビットを設定する必要はありません。アプリケーションで浮動小数点データを使用する場合は、このビットを設定して、スタック切り換えの前後で正しい浮動小数点コンテキストを保存する必要があります。
KP\$M_SET_STACK_LIMITS	スタック切り換えのたびに \$SETSTK_64 を呼び出します。条件処理では正確なスタック・リミット値が必要なため、プロセス・スコープ・アプリケーションは常にこのフラグを設定する必要があります。

- **PARAM\_SIZE**— 値渡しされるロングワード。KPB 内のユーザ・パラメータ領域のサイズをバイト数で示します。パラメータ領域が必要ない場合は、0 を渡します。

戻り値 (状態):

SS\$\_NORMAL  
SS\$\_INSFMEM  
SS\$\_INSFARG  
SS\$\_INSFRPGS

### C.1.6 モードに依存しない割り当て

モードに依存しない割り当ての構文は次のとおりです。

```
EXE$KP_USER_ALLOC_KPB kpb, flags, param_size, *kpb_alloc, mem_stack_bytes,
*memstk_alloc, rse_stack_bytes, *rsestk_alloc, *end_rtn
```

## C プロトタイプ

```
status = EXE$KP_USER_ALLOC_KPB( KPB_PPS kpb, int flags,
    int param_size,
    int (*kpb_alloc)(),
    int mem_stack_bytes,
    int(*memstk_alloc)(),
    int rse_stack_bytes,
    int(*rsestk_alloc)(),
    void(*end_rtn)())
```

### パラメータ

- KPB— 割り当てられたデータ構造のアドレスが返されるロングワードのアドレス。OpenVMS API の規則では、32 ビット・ポインタは 32 ビット・アドレス (struct KPB を参照するショート・ポインタを参照するショート・ポインタ) 渡しされます。
- FLAGS— フラグのロングワード・ビットマスク。値渡しされます。表 C-3 はこのパラメータに対して定義されているフラグを示しています。

表 C-3 モードに依存しない割り当てのフラグ

フラグ	説明
KPSM_VEST	OpenVMS システム KPB。一般に、このフラグは設定する必要があります。
KPSM_SPLOCK	KPB の内部にスピンロック領域を割り当てます。
KPSM_DEBUG	KPB の内部にデバッグ領域を割り当てます。
KPSM_DEALLOC_AT_END	カーネル・プロセス・ルーチンの終了時に、KPB の割り当てを自動的に解除しなければならないことを指定します。
KPSM_SAVE_FP	(I64 のみ、Alpha では無視されます) 汎用レジスタだけでなく、浮動小数点コンテキストも保存します。I64 システムでの整数乗算や除算などの特定の演算は、浮動小数点演算を使用してインプリメントできます。これらの演算では、最小浮動小数点レジスタ・セットを使用しますが、定義では、これらは保存されるレジスタ・セットに含まれていません。アプリケーションで最小浮動小数点レジスタ・セットだけを使用する場合は、このビットを設定する必要はありません。アプリケーションで浮動小数点データを使用する場合は、このビットを設定して、スタック切り換えの前で正しい浮動小数点コンテキストを保存する必要があります。
KPSM_SET_STACK_LIMITS	スタック切り換えのたびに\$SETSTK_64 を呼び出します。条件処理では正確なスタック・リミット値が必要なため、プロセス・スコープ・アプリケーションは常にこのフラグを設定する必要があります。

- PARAM\_SIZE – 値渡しされるロングワード。KPB 内のユーザ・パラメータ領域のサイズをバイト数で示します。パラメータ領域が必要ない場合は、0 を渡します。

- KPB\_ALLOC – KPB を割り当てるルーチンのプロシージャ記述子のアドレス。割り当てルーチンは、2 つのパラメータを指定して呼び出されます。1 つは長さパラメータ、もう 1 つは割り当てられた KPB のアドレスが返されるロングワードです。長さパラメータはバイト数で指定し、参照渡しされます。割り当てルーチンは、KP ルーチンが実行されるモードにとって適切な 32 ビット空間から、少なくとも要求されたバイト数を割り当てなければなりません。アドレスは少なくともクォードワード境界でアラインメントされる必要があり、実際の割り当てサイズを長さ引数に書き戻す必要があります。
- MEM\_STACK\_BYTES – 割り当てられるメモリ・スタックのサイズをバイト数で示す 32 ビット値。値渡しされます。返される値は、ハードウェア・ページ・サイズの倍数に切り上げられます。
- MEMSTK\_ALLOC – メモリ・スタックを割り当てるルーチンのプロシージャ記述子のアドレス。第 C.1.7 項はこのルーチンの形式と必要な動作を示しています。
- RSE\_STACK\_BYTES – 割り当てられるメモリ・スタックのサイズをバイト数で示す 32 ビット値。値渡しされます。返される値は、ハードウェア・ページ・サイズの倍数に切り上げられます。
- RSESTK\_ALLOC – メモリ・スタックを割り当てるルーチンのプロシージャ記述子のアドレス。第 C.1.7 項はこのルーチンの形式と必要な動作を示しています。
- END\_RTN – 終了ルーチンのプロシージャ記述子のアドレス。終了ルーチンは、EXESKP\_END を呼び出すか、または呼び出しルーチンに戻ることで、KP ルーチンが終了するときに呼び出されます。終了ルーチンは、スタックと KPB をキャッシングしたり、割り当てを解除したりするのに必要です。

### C.1.7 スタック割り当て API

スタック割り当てルーチンの API は、メモリ・スタック割り当ての場合も RSE スタック割り当ての場合も同じです。ルーチンを呼び出すときは、割り当てられている KPB の 64 ビット・アドレスと、割り当てられるハードウェア固有のページ数 (ページレット数ではない) を整数で指定します。

スタック割り当てルーチンを指定する構文は次のとおりです。

status = alloc-routine (KPB\_PQ kpb, const int stack\_pages)

- KPB—すでに割り当てられている KPB の 64 ビット・アドレス。64 ビット参照渡しされます。
- STACK\_PAGES—割り当てるページ数 (整数)。32 ビットの値渡しされます。

割り当てルーチンは、ページ境界でアラインメントされたアドレス空間を割り当てるものと考えられます。厳密に必須というわけではありませんが、両端にアクセスされないガード・ページを配置することで、スタックを保護することを強くお勧めします。また、最小スタック・サイズは、少なくとも SYSGEN パラメータ KSTACKPAGES (グローバル・セル SGN\$GL\_KSTACKPAG) の値になるようにして

ください。このように設定すると、アプリケーションを再コンパイルしなくても、スタック・サイズに関して特定の制御を行うことができます。また、I64 でのスタックの使い方は、従来のアーキテクチャとは大きく異なっているため、前に割り当てられていたスタック・サイズが適切でない可能性もあります。

メモリ・スタック割り当てルーチンでは、次の KPB フィールドを次のように設定する必要があります。

- `KPBSIS_STACK_SIZE`— ガード・ページを除く、スタックのサイズ (バイト数)。
- `KPBSPQ_STACK_BASE`— スタックのベース・アドレス。メモリ・スタックの場合は、割り当てられたスタックの末尾の次のバイトのアドレスです。つまり、割り当てられたアドレスに、割り当てサイズ (バイト数) を加算した値です。

メモリ・スタック割り当てルーチンでは、次の KPB フィールドを次のように設定できます。

- `KPBSQ_MEM_REGION_ID`— `$CREATE_REGION` システム・サービスから返された領域 ID。この情報は、スタックの割り当てを解除するときに必要です。

RSE スタック割り当てルーチンでは、次の KPB フィールドを次のように設定する必要があります。

- `KPBSIS_STACK_SIZE`— ガード・ページを除く、スタックのサイズ (バイト数)。
- `KPBSPQ_STACK_BASE`— スタックのベース・アドレス。RSE スタックの場合は、割り当てられた最下位アドレスです。

RSE スタック割り当てルーチンでは、次の KPB フィールドを次のように設定できます。

- `KPBSQ_REGION_ID`— `$CREATE_REGION` システム・サービスから返された領域 ID。この情報は、スタックの割り当てを解除するときに必要です。

どちらのルーチンも状態を呼び出しルーチンに返す必要があります。

### C.1.8 システムで提供される割り当てルーチンと割り当て解除ルーチン

本システムでは、多くの標準割り当てルーチンが提供されます。これらのルーチンは、KP 割り当て API に準拠しており、アプリケーションの要件を満たす場合は、ユーザ作成ルーチンの代わりに使用できます。次の割り当てルーチンが提供されています。

- `EXESKP_ALLOC_MEM_STACK` (カーネル・モード, S0/S1 空間)
- `EXESKP_ALLOC_MEM_STACK_USER` (ユーザ・モード, P0 空間)
- `EXESKP_ALLOC_RSE_STACK` (呼び出しルーチンのモード, S2 空間)
- `EXESKP_ALLOC_RSE_STACK_P2` (呼び出しルーチンのモード, P2 空間)

次の割り当て解除ルーチンが提供されています。すべての割り当て解除ルーチンに、1 つの引数 (KPB のアドレス) が必要です。

- EXESKP\_DEALLOCATE\_KPB
  - EXESKP\_ALLOCATE\_KPB が割り当てた KPB のみ。
  - スタックと KPB の割り当てを解除します。
- EXESKP\_DEALLOC\_MEM\_STACK
- EXESKP\_DEALLOC\_MEM\_STACK\_USER
- EXESKP\_DEALLOC\_RSE\_STACK
- EXESKP\_DEALLOC\_RSE\_STACK\_P2

### C.1.9 終了ルーチン

終了ルーチンは、EXESKP\_END が明示的に呼び出されるか、または KP ルーチンの最後に到達したときに、KP サービスから呼び出されます。EXESKP\_USER\_ALLOC\_KPB には任意の割り当てルーチンを指定できるので、終了ルーチンは、アプリケーションで将来使用するために KPB をキャッシュに保存しておくか、またはスタックおよび KPB に対して必要な割り当て解除ルーチンを呼び出す必要があります。

終了ルーチンを呼び出すには、次に示すように、2 つのパラメータを指定します。

void end\_routine (KPB\_PQ KPB, int status)

- KPB—すでに割り当てられている KPB の 64 ビット・アドレス。64 ビット参照渡しされます。
- STATUS—省略可能な状態値。EXESKP\_END が明示的に呼び出された場合は、状態値は、EXESKP\_END の 2 番目の引数に指定された値になります。2 番目の引数が省略されている場合は、SS\$\_NORMAL になります。呼び出しルーチンに戻ることによって KP ルーチンが終了する場合は、状態値は KP ルーチンの戻り値になります。

---

## C.2 KP 制御ルーチン

KPB とスタックが割り当てられた後、KP ルーチンの状態を判断する 4 つのルーチンを使用できます。

### C.2.1 概要

KP ルーチンは、EXESKP\_START を呼び出すことで開始されます。KP ルーチンは、実行中に EXESKP\_STALL\_GENERAL を呼び出すことで、制御を放棄することができます。ストールされた KP ルーチンは、EXESKP\_RESTART を呼び出すことで再開できます。KP ルーチンは、EXESKP\_END を明示的に呼び出すか、またはルーチンから戻ることで終了します。ルーチンから戻る場合は、KP サービスは EXESKP\_END を暗黙に呼び出します。

KP ルーチンが起動されると、現在の実行状態スレッドは現在のスタックに保存され、KP スタックがロードされ、KP ルーチンが呼び出されます。

KP ルーチンがストールされると、KP ルーチンのコンテキストは KP スタックに保存され、スタックは元のスタックに切り換えられ、メイン・ルーチンのコンテキストがスタックからロードされます。その結果、ストールによって、EXESKP\_START または EXESKP\_RESTART の最後の呼び出しから元のルーチンに戻ります。

KP ルーチンが再起動されると、現在のコンテキストは現在のスタックに保存され、スタックは KP ルーチンのスタックに切り換えられ、KP ルーチンのコンテキストがスタックから復元されます。EXESKP\_STALL\_GENERAL の最後の呼び出しから KP ルーチンに戻ります。

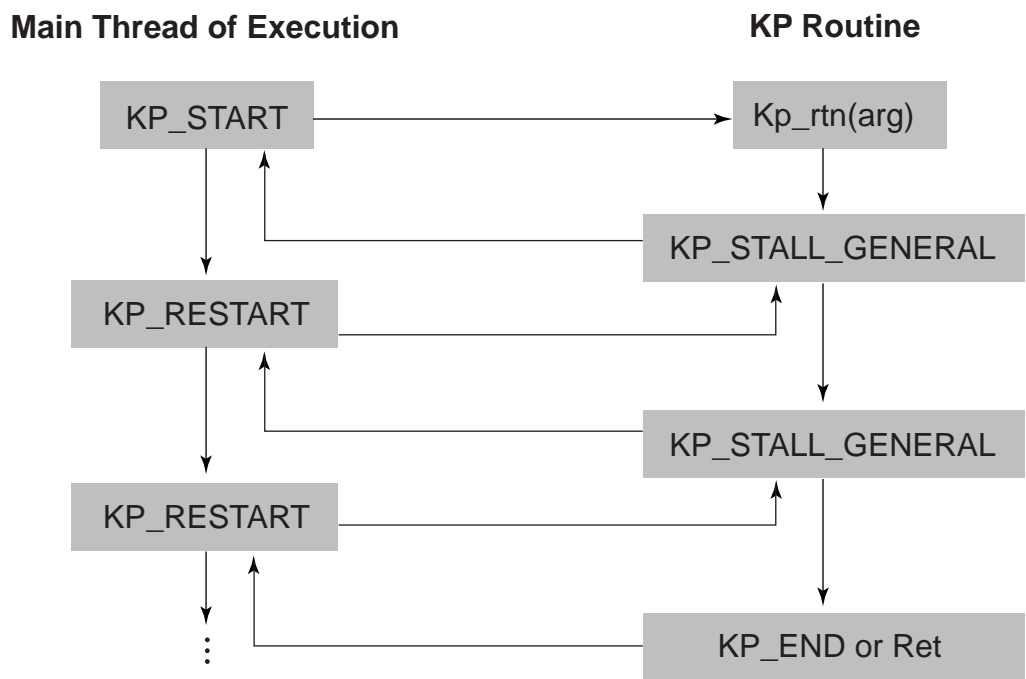
ストールと再開のシーケンスは 0 回以上実行できます。KP ルーチンをあらかじめストールしなければならないわけではありません。ストールされたルーチンは、再開されるまでストールできません。実行中の KP ルーチンは、ストールされるまで再起動できません。完全にチェックする SYSTEM\_PRIMITIVES.EXE で、これらのルールが適用されています。これらのルールに従わないと、KP ルーチンが実行されているモードに応じて、KP\_INCONSTATE バグチェックが発生することがあります。

EXESKP\_END を明示的に呼び出すか、呼び出しルーチンに戻ることによって、KP ルーチンが終了する場合、現在のコンテキストは保存されず、スタックは切り換えられ、元のスレッド・コンテキストが復元されます。この時点で、DEALLOCATE\_AT\_END フラグが設定されているか (カーネル・モードのみ)、終了ルーチンのアドレスが指定されている場合は、適切な処理が実行されます。KP ルーチンを起動または再起動した呼び出しから、元のスレッドに戻ります。

図 C-1 は全体的なコード・フローを示しています。



図 C-1 KP ルーチンの実行



VM-1170A-AI

---

注意

---

この図で、メイン実行スレッドは連続ストリームとして示されていますが、実際の実行スレッドには非同期コンポーネントが含まれることがあります。アプリケーションは必要な KPB のスコープを維持するだけでなく、必要なすべての同期化も実行しなければなりません。

---

## C.2.2 ルーチン説明

ここでは、各ルーチンについて説明します。

### C.2.2.1 EXE\$KP\_START

構文:

status = EXE\$KP\_START(kpb, routine, reg-mask)

- kpb— 前に割り当てられ、初期化されている KPB のアドレス。32 ビット参照渡しされます。
- routine— KP ルーチンのアドレス。
- reg-mask— レジスタ保存マスク。ロングワードです。値渡しされます。この引数は Alpha でのみ読み取られます。I64 インタフェースでは、レジスタの保存に関して OpenVMS 呼び出し規則だけがサポートされます。Alpha での高級言語からの呼び出しに対しては、KPBDEF に定義されている定数 KPREG\$K\_HLL\_REG\_MASK を使用して、呼び出し規則のレジスタ・マスクを指定できます。

このルーチンは、現在の実行スレッドを中断し、新しいスタックにスワップし、指定されたルーチンを呼び出します。KP ルーチンの呼び出しには、割り当てられている KPB の 32 ビット・アドレスを引数として指定します。KPB は無効かつ非アクティブでなければなりません。

### C.2.2.2 EXE\$KP\_STALL\_GENERAL

構文:

status = EXE\$KP\_STALL\_GENERAL(kpb)

- kpb— このルーチンの起動時に渡される KPB のアドレス。32 ビット参照渡しされます。

このルーチンは、現在の実行スレッドをストールし、コンテキストを KP スタックに保存し、このルーチンを最後に起動または再起動した呼び出しに戻ります。KPB は有効かつアクティブでなければなりません。

このルーチンからの戻り値 status は、このプロシージャを再起動したルーチンから渡されます。

#### C.2.2.3 EXE\$KP\_RESTART

構文:

EXE\$KP\_RESTART(kpb [, thread\_status])

- kpb— このルーチンをストールするために最後に使用された KPB のアドレス。32 ビット参照渡しされます。
- thread\_status – KP ルーチンを最後にストールした EXE\$KP\_STALL\_GENERAL の呼び出しの戻り値として渡される状態値。値渡しされます。省略可能です。このパラメータを省略した場合は、SS\$\_NORMAL が返されます。

このルーチンを呼び出すと、EXE\$KP\_STALL\_GENERAL の最後の呼び出しから戻ること、ストールされていたルーチンが再起動されます。この処理は、KP ルーチンを起動した元の実行スレッドとは完全に非同期に行われる操作です。KPB は有効かつ非アクティブでなければなりません。

#### C.2.2.4 EXE\$KP\_RESTART

構文:

status = EXE\$KP\_END(kpb [, status])

- kpb— このルーチンを起動または再起動するために最後に使用された KPB のアドレス。32 ビット参照渡しされます。
- status— KPB に指定されていた場合は、終了ルーチンに渡される状態値。値渡しされます。省略可能です。このパラメータを省略した場合は、SS\$\_NORMAL が返されます。

このルーチンは、KP ルーチンを終了し、KP ルーチンを起動または再起動した最後の実行スレッドに制御を返します。KPB は有効かつアクティブでなければなりません。KPB は無効かつ非アクティブとしてマークされるため、最初に EXE\$KP\_RESTART を呼び出して別の KP ルーチンを起動しない限り、この後の EXE\$KP\_RESTART や EXE\$KP\_STALL\_GENERAL の呼び出しで KPB を使用することはできません。

EXE\$KP\_END を呼び出す代わりに、KP ルーチンは呼び出しルーチンに戻ることもできます。呼び出しルーチンに戻る場合、KP コードは KP\_END を自動的に呼び出します。その場合、KP プロシージャからの戻り値 status は、状態引数として使用されます。

## C.3 設計上の考慮点

- KP ルーチンは単一プロセス内で動作します。複数の CPU を搭載したシステムで並列処理しても、KP サービス自体にメリットはありません。
- EXESKP\_STALL\_GENERAL, EXESKP\_RESTART, EXESKP\_END の呼び出しはすべて、EXESKP\_START の呼び出しと同じモードで実行する必要があります。呼び出しの前後でモードを変更してもかまいませんが、呼び出しはすべて、同じモードで行わなければなりません。
- 複数の KPB が同時に有効になることがあります。
- 複数の KPB が同時にアクティブになることがあります。これは、KP ルーチンが別の KPB を起動または再起動したことを示します。最も単純な場合を除き、アクティブ KPB の数が増加するにつれて、コード・フローは非常に複雑になります。このような場合、ワーク・キュー・モデルとディスパッチャを使用すると、複数のアクティブ KPB が必要になるという状況を回避できる可能性が高くなります。
- スタックの割り当てと割り当て解除には、システム時間およびリソースが大量に必要となるため、アプリケーションの内部である程度、KPB のキャッシングを検討する必要があります。
- カーネル・モードのプロセス・コンテキストで KP サービスを使用するアプリケーションは、スタックをワーキング・セットにロックする必要があります。

### ABI

Application Binary Interface の略。プログラムが特定のオペレーティング・システムの下で動作するために従わなければならないすべてのルールや規則の総称。たとえば、OpenVMS I64 ABI には、呼び出し規則、レジスタの使用、割り込みの処理、その他の関連トピックが含まれています。

Intel から発行されている『Itanium Software Conventions and Run-Time Architecture Guide』には、「ABI は、API、システム固有の規則、ハードウェアに関する記述、実行時アーキテクチャで構成される」と記述されています。

### ACPI

Advanced Configuration and Power Interface の略。ACPI は電源状態管理、システム・デバイス識別、およびハードウェア構成のためのインタフェースを提供します。

- ACPI 仕様:

<http://www.acpi.info/spec.htm>

- Intel の ACPI に関する Web サイト:

<http://developer.intel.com/technology/iapc/acpi/>

### ALAT

Advanced Load Address Table の略。Itanium プロセッサ上のルックアップ・テーブルであり、スペキュレーションによるメモリ・アクセスを追跡し、CPU が競合を取り扱うことができるようにします。

### AML

ACPI Machine Language の略。ACPI 互換のオペレーティング・システムでサポートされる仮想マシン用の擬似コードであり、ACPI 制御メソッドとオブジェクトはこの言語で書かれています。ACPI も参照してください。

### アプリケーション・レジスタ

さまざまな機能のために使用される 128 個の専用レジスタ。ar0 ~ ar127 のラベルが付けられています。たとえば、ar17 はバッキング・ストア・ポインタというレジスタです。

## Aries

HP Precision Architecture (PA) アプリケーション・コード向けの HP エミュレータ/トランスレータ。VEST (VAX から Alpha へのトランスレータ) と異なり、Aries は実行時にだけエミュレーションを行い、変換されたイメージは生成しません。イメージの終了時に、変換されたコードはすべて破棄されるので、プログラムを実行するたびに再作成する必要があります。

## ASL

ACPI Source Language の略。AML用のプログラミング言語。ASL ソース・コードは AML イメージにコンパイルされます。

## ASM

Address Space Match の略。仮想メモリ管理の一部です。

## 分岐レジスタ

Itanium の 64 ビット・レジスタで、br0 ~ br7 のラベルが付けられ、間接分岐でターゲット・アドレスを指定するために使用されます。分岐レジスタは呼び出し/戻り分岐を単純化します。

## バンドル

Itanium アーキテクチャ用のインストラクションの基本的な形式。バンドルは 128 ビット構造であり、3 つの 41 ビット・インストラクションおよび 5 ビットのテンプレートで構成されます。

## CISC

Complex Instruction Set Computing の略。RISC より複雑な、多くの可変長マシン・インストラクションが含まれます。RISC および VLIW も参照してください。

## COW

Copy-on-write の略。複数のプロセスによってマッピングされ、各プロセスが個別に書き込みを行わないメモリのことです。プロセスがあるページに書き込むと、そのページはプロセスのプライベート・ロケーションにコピーされ、再マッピングされます。元の共有ページは変更されません。COW は一般に、メモリ・マッピングされた共有ライブラリのローダで使用されます。

## CVF

Compaq Visual Fortran の略。

## DSDT

Differentiated System Description Table の略。ACPI サブシステムの一部。DSDT には Differentiated Definition Block (差別化定義ブロック) が含まれています。このブロックは、ベース・システムに関するインプリメンテーションおよび構成情報を提供します。

## DVD

Digital Versatile Disk の略。

## DWARF

デバッグおよびトレースバック情報 (ELF に埋め込まれます)。

## EFI

Extensible Firmware Interface の略。EFI の目的は、ハードウェアを初期化し、オペレーティング・システムのブートを開始することです。

詳細については、Intel の EFI ページを参照してください。

<http://developer.intel.com/technology/efi/index.htm>

## ELF

ELF (Executable and Linkable Format) は、Itanium プロセッサでのオブジェクト・ファイル、共有ライブラリ、および実行イメージのファイル形式を定義します。ELF は Itanium アーキテクチャ固有の形式ではなく、さまざまな UNIX システムで数年にわたって使用されています。

詳細については、以下の Web サイトで SCO が提供している ELF 仕様を参照してください。

<http://stage.caldera.com/developer/gabi/>

## EPIC

Explicitly Parallel Instruction Computing の略。CISCやRISCと対比して、Itanium アーキテクチャを記述するために Intel が考案した用語。EPIC は、CPU 内の並列処理へのアーキテクチャ上での目に見えるアクセスを可能にします。たとえば、データのスペキュレーションやレジスタ名の変更は、Alpha ではチップでインプリメンテーションされていて、ユーザには見えないようになっていますが、Intel Itanium プロセッサ・ファミリでは、これらの機能はアーキテクチャの一部として定義されています。EPIC は VLIW (Very Long Instruction Word) コンピューティング・アーキテクチャに類似しています。

## FADT

Fixed ACPI Description Table の略。このテーブルには、ACPI ハードウェア・レジスタ・ブロックのインプリメンテーションおよび構成の詳細情報や、DSDT の物理アドレスが格納されます。

## FAT

File Allocation Table の略。Microsoft のさまざまなオペレーティング・システムおよび Itanium EFI オペレーティング・システム・ローダで使用するディスク・ボリューム構造。FAT には、FAT8、FAT12、FAT16、FAT32 など、さまざまなバージョンがあります。各バージョンは、可能な合計ボリューム・サイズが異なり、その他にも違いがあります。

## フィル

スピル・アンド・フィルを参照してください。

## FIT

Firmware Interface Table の略。

## GEM

大部分の OpenVMS Alpha および I64 コンパイラで使用されるバックエンド・コード・ジェネレータ。

## 汎用レジスタ

Itanium プロセッサには 128 個の 64 ビット汎用レジスタがあり，r0 ~ r127 の番号が付けられています (gr0 ~ gr127 と呼ばれます)。しかし，通常は 32 個のレジスタだけが直接使用されます。他の 96 個のレジスタは引数の受け渡しや，現在実行中のプロシージャのローカル変数を格納するために使用されます。レジスタ r0 ~ r31 は静的ですが，レジスタ r32 以上は一般に RSE (Register Stack Engine) で管理され，プロシージャ呼び出し間で名前が変更される可能性があります。

## グローバル・ポインタ (GP)

現在のグローバル・データ・セグメントのベース・アドレス。ベース・アドレスが定義されていると，マシン・コードでコンパクトな GP 相対アドレスを使用できます。Itanium インストラクションの最大即値は 22 ビットなので，グローバル・データ・セグメントの長さは 4 MB です。現在の GP は汎用レジスタ 1 に格納されます。

## HP OpenVMS Migration Software for Alpha to Integrity Servers

Alpha 用の実行イメージを I64 用の実行イメージに変換するツール。

## HWPCB

Hardware Process Control Block の略。プロセスが再スケジューリングされるときに，ハードウェアで格納および取得されるプロセス・コンテキストの部分。

## HWRPB

Hardware Restart Parameter Block の略。これらのデータ構造は，システム起動時および再起動時に，コンソールと OpenVMS の間でシステム・レベルの構成情報を伝達するために使用されます。

## IA

Intel Architecture の略。

## IA-32

Intel の 32 ビット CISC マイクロプロセッサ・アーキテクチャ。たとえば，Pentium®，Xeon™，80386 などがインプリメントされています。

## IA-64

Itanium プロセッサ・ファミリ・アーキテクチャの以前の名称。IA-64 という名称は公式には使用されなくなりました。しかし，過去に発行されたドキュメントではこの名称が使用されており，一部のコードでも IA64 がまだ使用されています。

## Integrity

Itanium アーキテクチャに基づく HP サーバ製品ラインの商標。たとえば，rx2600 サーバは HP Integrity rx2600 として販売されています。



## ILP

Instruction Level Parallelism の略。同じサイクルで並列に複数のインストラクションを実行できる機能。

## インストラクション・グループ

明示的なストップまたは分岐によって区切られている，一連の 1 つ以上のインストラクション。インストラクション・グループ内のインストラクションには，RAWまたはWAW依存関係があってはなりません。インストラクション・グループは，バンドル境界を越えてもかまいません。

## Intel® Itanium®

64 ビット・アーキテクチャ (Itanium プロセッサ・ファミリ)，およびそのアーキテクチャをインプリメントした最初のチップ (Itanium プロセッサ) の両方の名前。

## Intel® Itanium® 2

Itanium プロセッサ・ファミリの第 2 世代。rx2600は Itanium-2 システムです。

## IPINT

Inter-Processor Interrupt の略。単に IPI とも呼ばれます。

## IPF

Itanium Processor Family の略。

## IPMI

Intelligent Platform Management Interface の略。サーバの管理に使用される API の集合。温度，電圧，ファン，シャーシ・スイッチなどの環境データの監視をはじめ，アラート，リモート・コンソールとコントロール，資産追跡などの機能があります。詳細については，次の Intel の Web サイトを参照してください。

<http://www.intel.com/design/servers/ipmi/>

## IVT

interrupt vector table の略。

## ジャケット・ルーチン

引数を，ある呼び出し規則から別の呼び出し規則に変換するインタフェース・ルーチン。

## MAS

Multiple Address Space の略。MAS オペレーティング・システムでは，システム上のすべてのプロセスが少なくとも部分的にプライベートな仮想アドレス空間を保有します。OpenVMS と Tru64 UNIX はどちらも MAS オペレーティング・システムです。SASも参照してください。

## MBR

Master Boot Record の略。最初のディスク・ブロックの内容であり，FAT32 でパーティショニング可能なディスクのコア・データ構造。

## NaT

Not a Thing の略。さまざまなレジスタに割り当てられる追加ビットであり、レジスタの内容が有効であるかどうかを示します。投機的実行で使用されます。たとえば、存在しない仮想アドレスからの読み込みを行うと、結果は NaT になります。NaT ビットは操作の実行とともに伝達されるので、NaT を含む操作の結果は NaT になります。

## NMI

Non-Maskable Interrupt の略。

## NUE

Linux Native User Environment の略。Ski Itanium エミュレータと組み合わせて使用される開発ソフトウェアの集合。

## PA

Hewlett-Packard (HP) Precision Architecture の略。PA-RISC と呼ばれるコンピュータ・アーキテクチャです。HP は、自社の UNIX を PA から Itanium アーキテクチャへ移植しています。

## PAL (Alpha)

Privileged Architecture Library の略。PAL は Alpha アーキテクチャの一部です。割り込みの処理や TLB 管理、不可分な操作など、VAX ではマイクロコードとしてインプリメントされていた下位レベルの操作をソフトウェアで実行するためのメカニズムです。Itanium システムでは、同じ機能がオペレーティング・システムの一部として提供されます。可能な場合は、OpenVMS I64 用の BLISS、C、および Macro コンパイラは下位互換性を維持するために、CALL\_PAL マクロを対応するオペレーティング・システム呼び出しに変換します。すべての Alpha PAL 操作が I64 でインプリメントされるわけではありません。場合によっては、PALcode を直接呼び出すプログラムを変更しなければならないこともあります。

## PAL (Itanium)

Processor Abstraction Layer の略。Itanium アーキテクチャの中で、ファームウェアでインプリメントされている部分。ハードウェア・エラーやシステムの初期化など、プロセッサ固有の機能に対して、一貫性のあるインタフェースを提供します。一般的に言うと、SAL はオペレーティング・システムをプラットフォーム固有のインプリメンテーションの違いから分離するのに対し、PAL はオペレーティング・システム (および SAL) をプロセッサ固有の違いから分離します。

## PIC

Position-independent code の略。相対アドレス参照だけを使用するマシン・コード。この方式を使用すると、コードは、変更せずにどのメモリ・ロケーションにもロードできます。

## POSSE

Pre-Operating System Startup Environment の略。POSSE は、Itanium® 2 サーバ上の標準的なファームウェア・ユーザ・インタフェースになります。HP 独自の付加価値コンポーネントであり、HP システムでのみ使用できます。

## PPN

Physical Page Number の略。

## PTE

Page Table Entry の略。

## プレディケーション

プレディケートと呼ばれる特殊な 1 ビット・レジスタ・セットの内容をもとに、条件に応じてインストラクションを実行する機能。大部分の I64 インストラクションには、プレディケートの番号が含まれています。対応するプレディケート・レジスタが真 (1) の場合は、そのインストラクションが実行されます。偽 (0) の場合は、インストラクションは“no-op”として取り扱われます。

## RAW

Read-after-write 違反。1 つのインストラクション・グループ内の 2 つのインストラクションの間のデータ依存性のタイプ。最初のインストラクションが書き込むレジスタと同じレジスタから、2 番目のインストラクションがデータを読み込むと、この状況が発生します。

## RID

Region Identifier の略。

## RISC

Reduced Instruction Set Computing の略。固定サイズのマシン・インストラクションであり、CISC より単純なマシン・インストラクションを目指して開発されました。CISC および VLIW も参照してください。

## RSE

Register Stack Engine の略。プロシージャ呼び出しの間で、レジスタの名前変更とスピル・アンド・フィルを取り扱うオンチップ機能。

## rx2600

最大 2 つの Itanium 2 CPU を搭載した HP サーバ。rx2600 は、OpenVMS I64 が稼動する最初の Itanium プラットフォームです。

## SAL

System Abstraction Layer の略。Itanium アーキテクチャの中で、ファームウェアでインプリメントされている部分。ハードウェア・エラーやシステムの初期化など、プラットフォーム固有の機能に対して一貫性のあるインタフェースを提供します。一般的に言うと、SAL はオペレーティング・システムをプラットフォーム固有のインプリメンテーションの違いから分離するのに対し、PAL は SAL およびオペレーティング・システムをプロセッサ固有の違いから分離します。

## SAS

Single Address Space の略。SAS オペレーティング・システムでは、すべてのプロセスが同じアドレス空間を共有します。MAS も参照してください。

## SCI

System Control Interrupt の略。ACPI イベントについてオペレーティング・システムに通知するために、ハードウェアで使用するシステム割り込み。

## Ski

Linux 用に無料で提供される Itanium エミュレータ。Ski は Intel Itanium プロセッサ・ファミリ・アーキテクチャをエミュレートするものであり、特定のインプリメンテーションではありません。

## スペキュレーション

メモリへのアクセス待ち時間をアプリケーションから見えなくするために、メモリ・アクセスをコードの最初の方に移動する処理。メモリ・アクセスが無効または不要であることがわかった場合は、簡単に破棄することができます。Itanium プロセッサのインストラクション ld.a, ld.s, chk.a, および chk.s はスペキュレーションのために使用されます。

## スピル・アンド・フィル

Itanium レジスタ・ファイルの中で、スタックとして使用される回転する部分。オーバーフローすると、レジスタはレジスタ・バッキング・ストアというメモリ領域に保存されます。スピルは、1 つ以上のレジスタをバッキング・ストアに保存します。フィルは、レジスタをバッキング・ストアから復元します。スピル・アンド・フィルは、CPU のRSE (Register Stack Engine) によって自動的に処理されるため、プログラムの介入の必要はありません。

## SRM コンソール

Alpha システムのファームウェアであり、ブートのサポート、PALcode、プラットフォーム固有の機能が含まれており、OpenVMS、UNIX、および Linux で使用されます。Alpha システムに電源を投入したときに表示される山括弧プロンプト (>>>) は、SRM コンソールの一部です。コンソールの動作は、『Alpha System Reference Manual (SRM)』で定義されています。

Itanium アーキテクチャには SRM コンソールに相当するものはありません。Itanium では、SRM の機能は、PAL/SALファームウェア、EFIブート・ソフトウェア、およびオペレーティング・システムの間で分割されています。

## ストップ

インストラクション・グループの末尾であり、アセンブリ・リストでは2つのセミコロンの (;;) で示されます。

## Superdome

ハイエンドの HP サーバ製品ライン。Superdome サーバは PA-RISC または Itanium の約 64 の CPU をサポートします。最終的には、すべての新しい Superdome システムが Itanium ベースになります。

## SWIS

SoftWare Interrupt Support の略。OpenVMS で使用される割り込みモデルをインプリメントするサービスの集合。プロセッサ IPL, AST, モード変更, およびソフトウェア割り込みなどが含まれています。Alpha では, このサポートは PALcode でインプリメントされていました。

## テンプレート

Itanium プロセッサでは, テンプレートは, 1 つのバンドルに配置できるインストラクションの組み合わせを定義します。たとえば, MFI テンプレートで定義されるバンドルには, メモリ・インストラクション (M), 浮動小数点インストラクション (F), および整数インストラクション (I) がこの順序で配置されている必要があります。テンプレートは, ストップの位置も定義します。Itanium アーキテクチャでは 24 のテンプレートが定義されていますが, 英字の組み合わせは 12 組しかありません。各組み合わせに対して, 2 つのテンプレート・バージョンがあります。それは, 末尾にストップのあるバージョンと, ストップのないバージョンです。一部のテンプレートでは, バンドルの内部にもストップが含まれています。

## TLB

Translation Lookaside Buffer の略。実メモリと仮想メモリの間の, 最近使用されたマッピングを格納するオンチップ・キャッシュ。Itanium アーキテクチャには, インストラクション用の TLB とデータ用の TLB があります。

## USB

Universal Serial Bus の略。キーボード, マウス, プリンタ, カメラ, 外部ディスクなどの I/O デバイス用のコネクタ。

## VEST

VAX の実行イメージを Alpha の実行イメージに変換するツール。DECmigrate とも呼ばれます。

## VHPT

Virtual Hash Page Table の略。

## VIAL

OpenVMS to Itanium Abstraction Layer の略。VIAL は, OpenVMS の SRM コンソール PALcode レイヤに相当します。

## VLIW

Very Long Instruction Word コンピューティング・アーキテクチャ。多くの VAX プロセッサの奥深くでインプリメントされていたマイクロコード・エンジンを連想させる非常に複雑なインストラクション。CISC, RISC, および EPIC も参照してください。

## VPN

Virtual Page Number の略。

## VRN

Virtual Region Number の略。

#### WAW

Write-after-write 違反。1つのインストラクション・グループ内の2つのインストラクション間のデータ依存性のタイプ。2つのインストラクションがどちらも同じレジスタに書き込むことによって発生する違反です。

#### WWID

Worldwide ID の略。世界で固有の Fibre Channel デバイス ID であり、イーサネットのステーション・アドレスに類似しています。

## A

Ada ..... 6-1  
 ADA ..... 6-2  
 Ada 83 ..... A-4  
 ADA コンパイラ  
   代替製品 ..... B-1  
 Alpha 命令  
   解釈実行 ..... 4-5  
 Analyze/Image ユーティリティ (ANALYZE  
   /IMAGE) ..... 5-2  
 Analyze/Object ユーティリティ (ANALYZE  
   /OBJECT) ..... 5-3  
 asm ステートメント ..... A-9  
 AST (非同期システム・トラップ) ..... A-7

## B

BASEstar Classic  
   代替製品 ..... B-1  
 BASIC ..... 2-5, 6-1, 6-2  
 BLISS コンパイラ  
   Alpha と I64 の相違点 ..... 6-2  
 BLISS ビルトイン ..... A-10

## C

C ..... 2-5  
 C++ ..... 2-5, 5-12  
 C asm ステートメント ..... A-9  
 CLISDCL\_PARSE  
   外部定義 ..... 4-19  
 CLUE (Crash Log Utility Extractor)  
   Crash Log Utility Extractor を参照  
 CMA\_DELAY API ライブラリ・ルーチン ... 4-21  
 CMA\_TIME\_GET\_EXPIRATION API ライブラリ・  
   ルーチン ..... 4-21  
 CMS ..... 4-3  
 COBOL ..... 2-5, 5-12, 6-1, 6-19  
 Code Management System  
   CMS を参照  
 Crash Log Utility Extractor (CLUE) ... 4-10, 5-3  
 SCRMPSC ..... A-7

## D

DECset ..... 5-3  
 Digital Test Manager ..... A-6  
 DWARF イメージ・ファイル形式 ..... 4-16

## E

ELF オブジェクト・ファイル形式 ..... 4-16

## F

Fortran ..... 6-1, 6-20  
 Fortran 77 ..... A-4

## H

HP C ..... 6-23  
 HP C++ ..... 6-25  
 HP OpenVMS Migration Software for Alpha to  
   Integrity Servers ..... 4-3, 5-3  
   Alpha システムおよび I64 システムで動  
   作 ..... 4-4  
   機能 ..... 4-4  
   必要なリソース ..... 4-4  
   分析機能 ..... 4-5  
 HP rx2600 システム ..... 4-3  
 HP Services のサポート ..... 3-9

## J

Java ..... 6-26  
 JSB (jump to subroutine) ..... 3-5

## K

KPB ..... C-4  
 KP サービス ..... C-1  
 KP 制御ルーチン ..... C-11

## L

LIB\$UNLOCK\_IMAGE ルーチン ..... 4-25  
 LIB\$WAIT  
   I64 と Alpha に共通のコード ..... 4-18  
 LIB\$LOCK\_IMAGE ルーチン ..... 4-25  
 Librarian ユーティリティ (LIBRARIAN)  
   ネイティブ Alpha ..... 5-2

## M

Macro-32 ..... 5-1  
Macro-32 コンパイラ ..... 5-2  
MACRO-32 コンパイラ ..... 6-26  
Macro-64 ..... A-4  
MACRO コード  
置換 ..... A-4  
Message ユーティリティ (MESSAGE)  
ネイティブ Alpha ..... 5-2  
MMS ..... 4-4  
Module Management System  
MMS を参照

## O

OTSSCALL\_PROC ルーチン ..... 4-6

## P

Pascal ..... 2-5, 6-1, 6-27  
Pathworks 32  
代替製品 ..... B-1  
PGFIPLHI バグ・チェック ..... 4-24  
PL/I ..... A-4

## R

RSE スタック ..... C-1

## S

SDA  
System Dump Analyzer ユーティリティを参照  
SS\$\_FLTDIV 例外 ..... 4-16  
SS\$\_FLTINV 例外 ..... 4-16  
SS\$\_HPARITH 例外 ..... 4-16  
SYSS\$BASE\_IMAGE  
に対してリンク ..... A-8  
SYSS\$GOTO\_UNWIND\_64 システム・サービ  
ス ..... 4-15  
SYSS\$GOTO\_UNWIND システム・サービ  
ス ..... 4-15, A-7  
SYSS\$LCKPAG\_64 システム・サービス ..... 4-25  
SYSS\$LCKPAG システム・サービス ..... 4-25  
SYSS\$LKWSET\_64 システム・サービス ..... 4-24  
SYSS\$LKWSET システム・サービス ..... 4-24, A-7  
System Dump Analyzer ユーティリティ  
(SDA) ..... 5-3  
OpenVMS Alpha ..... 4-9

## T

THREADCP コマンド ..... 4-21  
TIE (Translated Image Environment) .. 4-3, 4-4  
自動的に起動される ..... 4-5  
説明 ..... 4-4

## V

VAX MACRO  
OpenVMS I64 システムでの再コンパ  
イル ..... 5-2

## X

XDeltta デバッガ ..... 4-9, 5-2

## ア

アーキテクチャへの依存性 ..... A-6  
アプリケーション  
アーキテクチャへの依存性 ..... A-6  
オペレーティング環境 ..... 3-7  
開発環境 ..... 3-7  
基準値の設定 ..... 4-8  
グローバル・セクションの使用 ..... A-7  
コンパイラ ..... A-4  
コンポーネント ..... A-3  
サイズ ..... A-3  
使用されている言語 ..... A-4  
ソフトウェアへの依存性 ..... 3-6, A-5  
ドキュメント ..... A-4  
ハードウェアへの依存性 ..... A-5  
パフォーマンスへの感度 ..... A-8  
評価 ..... 3-2  
依存性 ..... 3-6, A-4  
オペレーション・タスク ..... 3-8  
ポーティングの内容 ..... 3-3  
ポーティング方法 ..... 3-3  
変更が必要 ..... 3-5  
マシン・インストラクション ..... A-8  
要素間の同期化 ..... A-7  
呼び出し規則 ..... 3-5, A-9  
アプリケーションの基準値  
設定 ..... 4-8  
アプリケーションのテスト  
Alpha システムでの基準値の設定 ..... 4-8  
I64 システム ..... 4-10  
コンパイル ..... A-10  
システム・テスト ..... A-10  
単体テスト ..... A-10  
リグレッション・テスト ..... A-6, A-10  
アプリケーションのポーティング ..... 6-1

## イ

移行環境  
用意 ..... 4-2  
移行ツール ..... 4-3  
インクルード・ファイル  
C プログラム ..... 4-3



## オ

オブジェクト・ファイル形式  
依存 ..... 4-16  
オブジェクト・ファイルの形式 ..... 3-5, A-9

## ク

クラッシュ  
分析 ..... 4-9

## ケ

言語 ..... A-4

## コ

コード  
アセンブラ ..... A-8  
変更が必要 ..... 3-5  
ユーザ作成 ..... A-8  
コマンド定義ファイル ..... 4-19  
コンパイラ  
VAX 浮動小数点データ・タイプ用の修飾  
子 ..... 5-1  
アプリケーションのポーティングの準備 ... 6-1  
コマンド ..... 4-8  
最適化 ..... 5-1  
ネイティブ Alpha ..... 5-1  
ネイティブ I64 ..... 5-1  
ポーティング・サポートの評価 ..... A-4  
コンパイル・コマンド  
必要な変更 ..... 4-8  
コンパイル・プロシージャ ..... 4-3

## サ

再コンパイル  
エラーの解決 ..... 4-8  
コンパイル・コマンドの変更点 ..... 4-8  
再リンク ..... 5-2  
リンク・コマンドの変更点 ..... 4-8

## シ

実行イメージ・ファイルの形式 ..... 3-5, A-9  
システムコード・デバッグ ..... 4-9  
デバッグも参照  
システム・サービス ..... 3-2, 3-5  
SYSSGOTO\_UNWIND ..... 4-15, A-7  
SYSSGOTO\_UNWIND\_64 ..... 4-15  
SYSSLCKPAG ..... 4-25  
SYSSLCKPAG\_64 ..... 4-25  
SYSSLKWSET ..... 4-24, A-7  
SYSSLKWSET\_64 ..... 4-24

## システム・ダンプ・ファイル

分析 ..... 4-9  
ジャケット・ルーチン ..... 4-4, 4-6  
条件付きコード ..... 3-5, 4-12, A-8  
条件ハンドラ ..... 4-16, A-7

## ス

スタック  
切り換え ..... C-1  
スレッド・インタフェース  
I64 でのサポート ..... 4-20  
既存の API ライブラリ・ルーチン ..... 4-21

## セ

接続/割り込み機能 ..... A-8

## ソ

相互運用性  
準拠 ..... 4-11

## タ

ターミナル・ドライバ ..... 3-5  
ダンプ・ファイル  
システム・ダンプ・ファイルを参照

## テ

テスト・ツール ..... 4-3  
I64 固有 ..... 4-11  
I64 にポーティングされている ..... 4-10  
データ・アラインメント ... 3-5, 4-22~4-23, A-6  
アラインメントされていないデータの検  
出 ..... 4-22  
コンパイラ・オプション ..... 4-22, 4-23  
パフォーマンス ..... 4-22  
変換されたソフトウェアとの互換性の問  
題 ..... 4-23  
データ構造  
アラインメント ..... 3-5, A-6  
内部 ..... 3-5, A-8  
浮動小数点 ..... 3-5, 3-8, A-6  
メカニズム・アレイ ..... 3-5, A-7  
データ・タイプ  
浮動小数点 ..... 3-5, 3-8, A-6  
デバッグ ..... 4-8~4-9  
XDelta ..... 5-2  
ネイティブ I64 ..... 5-2  
デバッグ ..... 5-2  
デバッグ・シンボル・テーブル ..... 4-16  
形式 ..... 3-5, A-9

## ト

同期化	
プロセス間通信	A-7
特権モード動作	A-8
ドライバ	
ターミナル	3-5
トランスレータ	
バイナリ・トランスレータを参照	

## ナ

内部データ構造	3-5, A-8
---------	----------

## ハ

バイナリ・トランスレータ	4-1, 5-3
バグ	
潜在的	4-11

## フ

浮動小数点データ・タイプ	3-5, 3-8, A-6
IEEE	4-17
VAX	4-17, 5-1
プログラミング言語	
Ada	6-1
BASIC	5-1, 6-1
BLISS	5-1, 6-1
C	5-1, 6-1
インクルード・ファイル	4-3
C++	5-1, 6-1
COBOL	5-1, 6-1
Fortran	5-1, 6-1
Java	5-1, 6-1
Macro-32	6-1
Macro-64	6-1
Pascal	5-1
VAX MACRO	5-1
プログラム・セクション	
オーバーレイ	4-16

## ヘ

変換	5-3, A-3, A-4
I64 コンパイラが提供されていない言語で作成されたプログラム	5-3
生成されるイメージの種類	4-5
用のツール	4-4
変換イメージ	
共存	4-5
変換されたイメージ	
内容	4-5

## ホ

ポーティング・チェックリスト	7-5
ポーティングの依存性	
外部	3-6, A-4
ポーティングのステップ	A-10
ポーティングの要件	
評価	3-1
ポーティング方法	3-3
ポーティング・リソース	3-9

## マ

マシン・インストラクション	
正しい実行	A-8
マルチプロセッシング	A-7

## メ

メカニズム・アレイ・データ構造	3-5, 4-16
-----------------	-----------

## ヨ

呼び出し規則	3-5, A-9
I64 での変更点	2-1
依存するコード	4-24

## ラ

ライブラリ・インタフェース	3-2, 3-5
---------------	----------

## リ

リンカ・ユーティリティ	
ネイティブ I64	5-2
リンク・コマンド	
必要な変更	4-8
リンク・プロシージャ	4-3
リンケージ宣言	A-9

## レ

レイヤード・プロダクト	
サポート対象外	B-1

## ロ

論理名	
ツールとファイル	4-3

## ワ

ワーキング・セット	
変更	A-7





OpenVMS Alpha から OpenVMS I64 へのアプリケーション・ポーティング・ガイド

---

2005 年 4 月 発行

日本ヒューレット・パカード株式会社

〒140-8641 東京都品川区東品川 2-2-24 天王洲セントラルタワー

電話 (03)5463-6600 (大代表)

---

AA-RX0JA-TE

