Open VMS Alpha
オペレーティング・システム

64 ビット・アドレッシングおよび VLM 機能説明書

AA-QTQ0C-TE

1999 年 4 月

本書は， OpenVMS Alpha がサポートする 64 ビット仮想アドレッシング，および VLM(Very Large Memory) について説明します。

改訂/更新情報： 本書は， OpenVMS Alpha V7.1 の『OpenVMS Alpha 64 ビット・アドレッシング・ガイド』に代わる新しいマニュアルです。

オペレーティング・システム： OpenVMS Alpha バージョン 7.2

コンパックコンピュータ株式会社

1999 年 4 月

本書の著作権はコンパックコンピュータ株式会社が保有しており，本書中の解説および図，表はコンパック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，コンパック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。

© Compaq Computer Corporation 1999.

All Rights Reserved.
Printed in Singapore.

UNIX は X/Open カンパニーリミテッドが独占的にランセンスしている米国ならびに他国における登録商標です。

その他の商標および登録商標はすべて該当する商標所有社の所有物です。

原典　OpenVMS Alpha Guide to 64-Bit Addressing and VLM Features
Copyright ©1999 Compaq Computer Corporation

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 vii

1 はじめに

1.1 64 ビット・アドレッシングの使用 1–1
1.2 VLM 機能の使用 1–2

2 仮想アドレス空間の概要

2.1 従来の OpenVMS 32 ビット仮想アドレス空間レイアウト 2–2
2.2 OpenVMS Alpha 64 ビット仮想アドレス空間レイアウト 2–3
2.2.1 プロセス・プライベート空間 2–4
2.2.2 システム空間 2–4
2.2.3 ページ・テーブル空間 2–5
2.2.4 仮想アドレス空間のサイズ 2–5
2.3 仮想リージョン 2–6
2.3.1 P0 空間および P1 空間内のリージョン 2–8
2.3.2 64 ビット・プログラム・リージョン 2–8
2.3.3 ユーザ定義仮想リージョン 2–9

3 システム・サービスの 64 ビット・アドレッシングのサポート

3.1 システム・サービスに関する用語 3–1
3.2 64 ビット・アドレスをサポートするシステム・サービス 3–2
3.3 符号拡張チェック 3–6
3.4 言語の 64 ビット・システム・サービスのサポート 3–6
3.5 C の NEW STARLET 定義 3–6

4 メモリ管理 VLM 機能

4.1 VLM 機能の概要 4–1
4.2 メモリ常駐グローバル・セクション 4–2
4.3 グローバル・セクションのための Fast I/O とバッファ・オブジェクト 4–4
4.3.1 $QIO と Fast I/O の比較 4–5
4.3.2 バッファのロックの概要 4–5
4.3.3 バッファ・オブジェクトの概要 4–6
4.3.4 バッファ・オブジェクトの作成と使用 4–7
4.4 共用ページ・テーブル 4–8
4.4.1 プライベート・ページ・テーブルのメモリ必要量 4–8
4.4.2 共用ページ・テーブルおよびプライベート・データ 4–9

4.5 **拡張可能なグローバル・ページ・テーブル** 4–10
4.6 **予約メモリ・レジストリ** 4–11
4.6.1 予約メモリ・レジストリの使用 4–12
4.6.1.1 予約メモリ・レジストリ・データ・ファイル 4–13
4.6.1.2 AUTOGEN 4–13
4.6.1.3 予約メモリ・レジストリへのエントリの追加 4–13
4.6.2 予約メモリ・レジストリからのエントリの削除 4–14
4.6.2.1 予約メモリの割り当て 4–15
4.6.2.2 予約メモリの解放 4–16
4.6.2.3 予約メモリの表示 4–16
4.6.2.4 予約メモリの使用 4–17
4.6.2.5 予約メモリの復帰 4–18
4.6.3 アプリケーション構成 4–18

5 64 **ビット・アドレッシングを対象とする** RMS **インタフェースの強化**

5.1 RAB64 **データ構造** 5–2
5.2 64 **ビット** RAB **拡張の使用** 5–3
5.3 **ユーザ** RAB **構造をサポートするマクロ** 5–4

6 **ファイル・システムの** 64 **ビット・アドレッシングのサポート**

7 OpenVMS Alpha **デバイスの** 64 **ビット・アドレッシングのサポート**

7.1 $QIO **の** 64 **ビット・アドレスのサポート** 7–1
7.2 64 **ビット・アドレスをサポートする** OpenVMS **ドライバ** 7–3
7.3 64 **ビット・アドレスをサポートする機能コード** 7–6
7.4 SCSI **クラス・ドライバ用の** 64 **ビット** IO$_DIAGNOSE **関数** 7–7
7.4.1 64 ビット S2DGB 例 7–13

8 OpenVMS Alpha 64 **ビット** API **ガイドライン**

8.1 **クォドワード**/**ロングワード引数ポインタのガイドライン** 8–1
8.2 Alpha/VAX **ガイドライン** 8–10
8.3 32 **ビット** API **から** 64 **ビット** API **への拡張** 8–11
8.4 32 **ビット・ルーチンおよび** 64 **ビット・ルーチンの例** 8–12

9 64 **ビット・アドレッシングをサポートする** OpenVMS Alpha **ツールおよびユーティリティ**

9.1 OpenVMS **デバッガ** 9–1
9.2 OpenVMS Alpha **システム・コード・デバッガ** 9–2
9.3 Delta/XDelta 9–2
9.4 OpenVMS **ランタイム・ライブラリの** LIB$**および** CVT$**機能** 9–2
9.5 **ウォッチポイント・ユーティリティ** 9–2
9.6 SDA 9–3

10 言語およびポインタの 64 ビット・アドレッシング・サポート

11 DEC C RTL の 64 ビット・アドレッシング・サポート

11.1 DEC C ランタイム・ライブラリの使用 11–1
11.2 メモリを指す 64 ビット・ポインタの取得 11–2
11.3 DEC C ヘッダ・ファイル 11–3
11.4 影響のある関数 11–4
11.4.1 ポインタ・サイズの影響がない関数 11–4
11.4.2 両方のポインタ・サイズを受け取る関数 11–5
11.4.3 2 種類の実装を持つ関数 11–5
11.4.4 32 ビット・ポインタに制限されている関数 11–7
11.5 ヘッダ・ファイルの読み込み 11–8

12 MACRO–32 プログラミングの 64 ビット・アドレッシング・サポート

12.1 64 ビット・アドレッシングのガイドライン 12–1
12.2 64 ビット・アドレッシングのための新規コンポーネントおよび変更されたコンポーネント 12–1
12.3 64 ビット値の引き渡し 12–2
12.3.1 固定長サイズ引数リストでの呼び出し 12–2
12.3.1.1 $SETUP_CALL64，$PUSH_ARG64，および$CALL64 の使用上の注意 12–3
12.3.2 可変サイズ引数リストでの呼び出し 12–4
12.4 64 ビット引数の宣言 12–5
12.4.1 QUAD_ARGS の使用上の注意 12–5
12.5 64 ビット・アドレス計算の指定 12–6
12.5.1 ロングワード操作の折り返し動作への影響 12–7
12.6 符合拡張およびチェック 12–8
12.7 Alpha 命令ビルトイン 12–8
12.8 ページ・サイズ依存値の計算 12–8
12.9 64 ビット・アドレス空間でのバッガの作成および使用 12–9
12.10 64K バイトを越える転送のコーディング 12–9
12.11 MACRO–32 コンパイラの使用 12–10

A 64 ビット・アドレッシングのための C マクロ

DESCRIPTOR64 . . . . . A–2
$is_desc64 . . . . . A–3
$is_32bits A–4

B 64 ビット・アドレッシングのための MACRO-32 マクロ

B.1 64 ビット・アドレスを操作するマクロ B–1
$SETUP_CALL64 . . . . . B–2
$PUSH_ARG64 . . . . . B–4
$CALL64 . . . . . B–6
B.2 符合拡張とディスクリプタ形式をチェックするマクロ B–6
$IS_32BITS B–7
$IS_DESC64 . . . . . B–9

C 64 ビット・プログラム例

D VLM プログラム例

索引

図

2–1 32 ビット仮想アドレス空間レイアウト 2–2
2–2 64 ビット仮想アドレス空間レイアウト 2–3
5–1 RAB64 データ構造 5–2
7–1 32 ビットの OpenVMS SCSI-2 Diagnose Buffer (S2DGB) のレイアウト 7–8
7–2 64 ビットの OpenVMS SCSI-2 Diagnose Buffer (S2DGB) のレイアウト 7–9
7–3 戻り IOSB の形式 7–14
8–1 32 ビット・ディスクリプタ 8–3
8–2 item_list_64a 8–5
8–3 item_list_64b 8–5

表

3–1 64 ビット・システム・サービス 3–3
3–2 _NEW_STARLET プロトタイプが使用する構造体 3–8
4–1 ページ・テーブル・サイズ必要量 4–9
7–1 $QIO[W]引数の変更 7–2
7–2 64 ビット・アドレスをサポートするドライバ 7–4
7–3 32 ビット・アドレスに制限されるドライバ 7–5
7–4 64 ビット・アドレスが使用できる機能コード 7–6
11–1 2 種類の実装を持つ関数 11–6
11–2 32 ビット・ポインタに制限されている関数 11–7
11–3 32 ビット・ポインタだけを渡すコールバック 11–7
12–1 64 ビット・アドレッシングのための新規コンポーネントおよび変更されたコンポーネント 12–1
12–2 固定サイズ引数リストでの 64 ビット値の引き渡し 12–2

まえがき

本書は， OpenVMS ALpha V7.2 による 64 ビット・アドレッシング，および VLM(Very Large Memory) について説明します。

本書での説明は， OpenVMS Alpha システム上のアプリケーションにのみ適用します。 OpenVMS VAX システム上のアプリケーションには影響ありません。

対象読者

本書は，システム・プログラマ，およびアプリケーション・プログラマを対象にしています。また， OpenVMS Alpha プログラミング環境および概念を理解していることを前提としています。

本書の構成

第 1 章では， OpenVMS Alpha の 64 ビット・アドレッシングのサポート，および VLM 機能について簡単に説明します。

第 2 章では， OpenVMS Alpha の 64 ビット仮想メモリ・アドレス空間の概要を説明します。

第 3 章から第 12 章まででは， 64 ビット・アドレッシングおよび VLM をサポートする， OpenVMS ALpha プログラミング・ツールや言語について説明します。

付録には，マクロの説明とプログラミング例があります。

関連資料

本書には，いくつかの項目についてハイレベルな記述が含まれています。詳細は以下に示すドキュメントを参照してください。

- 『OpenVMS Programming Concepts Manual』
- 『OpenVMS Calling Standard』
- 『OpenVMS System Services Reference Manual: A–GETMSG』および『OpenVMS System Services Reference Manual: GETQUI–Z』
- 『OpenVMS Record Management Services Reference Manual』

- 『OpenVMS RTL Library (LIB$) Manual』
- 『OpenVMS デバッガ説明書』
- 『OpenVMS Alpha System Dump Analyzer Utility Manual』
- 『OpenVMS Alpha Guide to Upgrading Privileged-Code Applications』

本書で使用する表記法

本書では次の表記法を使用しています。

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

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

1
はじめに

OpenVMS Alpha オペレーティング・システム V7.0 以降は， 64 ビット仮想メモリ・アドレッシングを提供します。これにより OpenVMS Alpha オペレーティング・システムおよびアプリケーション・プログラマは， Alpha アーキテクチャによって定義される 64 ビット仮想アドレス空間を使用することができます。 OpenVMS Alpha V7.2 は V7.1 の Very Large Memory (VLM) を発展させ，拡張的な追加メモリ管理 VLM 機能を提供します。

本章では， OpenVMS Alpha の 64 ビット機能および VLM 機能の特徴と長所について説明します。本書のこれ以降の章では，これらの機能を使用することによって，アプリケーション・プログラムを拡張して 64 ビット・アドレスをサポートし，大量の物理メモリを効果的に利用する方法について説明します。

1.1 64 ビット・アドレッシングの使用

(デバッガ，ランタイム・ライブラリ・ルーチン， DEC C を含め) 多くの OpenVMS Alpha ツールおよび言語は， 64 ビット仮想アドレッシングをサポートします。入出力操作は， RMS サービス，$QIO システム・サービス，および OpenVMS Alpha システムが提供する多くのデバイス・ドライバを使用して， 64 ビットアドレス指定が可能な空間との間で直接行われます。

この基になっているのが新しいシステム・サービスで，これによりアプリケーションは，プロセスがプライベートに使用できる 64 ビット仮想アドレス空間を割り当て，管理することができます。

64 ビット・アドレッシングをサポートする OpenVMS Alpha ツールおよび言語を使用することによって，プログラマは， 32 ビット仮想アドレスの制限を越えてデータをマップおよびアクセスするイメージを作成することができます。 64 ビット仮想アドレス空間の設計では， 64 ビット・アドレスをさまざまな方法で使用できる柔軟性に富む構成が提供され，新たに生じる問題に対処できる一方で， OpenVMS Alpha V7.0 以前のバージョンで稼働するプログラムの上位互換性が確保されます。

非特権プログラムであっても，変更を加えることによって 64 ビット・アドレッシング機能を利用できるようになります。なお， OpenVMS Alpha 64 ビット仮想アドレッシングは， 64 ビット・サポートを適用するように明示的に変更されていない非特権プログラムには，影響を与えません。既存の 32 ビット非特権プログラムのバイナリおよびソース・レベルでの互換性は保障されています。

64 ビット・アドレッシングの機能を使用することによって，アプリケーション・プログラムは大量のデータをメモリにマップし，高い性能を実現した上で， Very Large Memory (VLM) システムを利用することができます。さらに，より大きなユーザ・プロセスに加えて，実質的に無制限のスケーラビリティでより多数のユーザおよびクライアント/サーバ・プロセスが使用できるため，システムのリソースをより効率的に使用できます。

1.2 VLM 機能の使用

OpenVMS Alpha のメモリ管理 VLM 機能は，データベース，データ・ウェアハウス，およびその他の大規模データベース (VLDB) 製品に対して拡張的なサポートを提供します。データベース製品およびデータ・ウェアハウス・アプリケーションは，新しい VLM 機能を使用することにより，容量および性能が向上します。

拡張的な VLM 機能を使用すると，アプリケーション・プログラムは，プロセス制限値を増加せずに，大きなメモリ内グローバル・データ・キャッシュを作成することができます。この大きなメモリ常駐グローバル・セクションは共用グローバル・ページでマップできるため，大量のメモリをマップするために必要なシステム・オーバヘッドを大幅に削減できます。

この VLM 機能の利用についての詳細は，第 4 章を参照してください。

2

仮想アドレス空間の概要

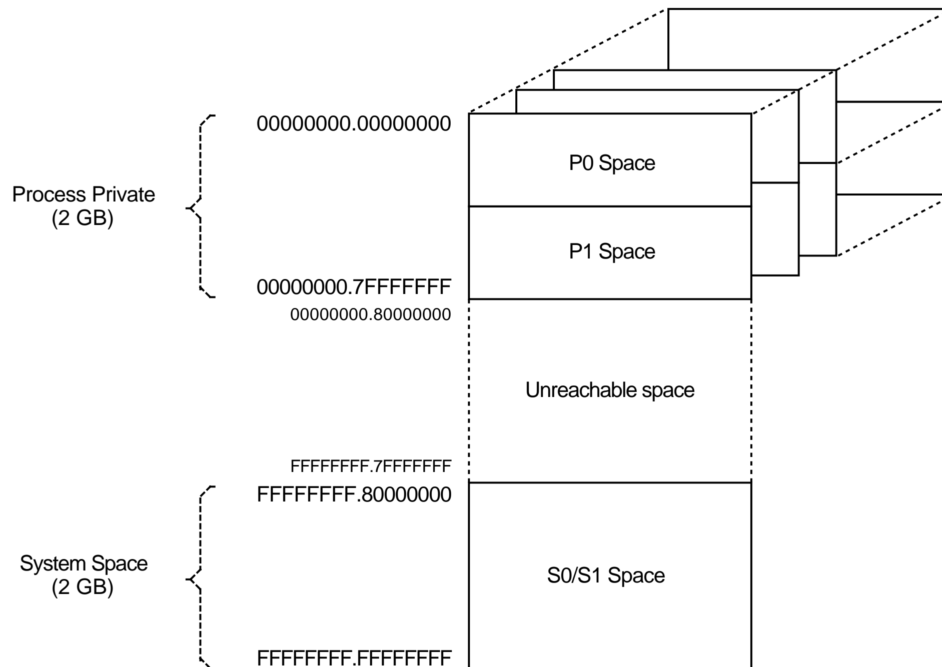
本章では， OpenVMS Alpha 64 ビット仮想メモリ・アドレス空間のレイアウトと構成要素について説明します。

64 ビット・アドレッシングをサポートする OpenVMS ALpha プログラミング・ツールや言語についての詳細，および 64 ビット・アドレッシングや VLM をサポートするようアプリケーションを拡張する場合の注意点については，本書のこれ以降の章を参照してください。

2.1 従来の OpenVMS 32 ビット仮想アドレス空間レイアウト

以前のバージョンの OpenVMS Alpha オペレーティング・システムの場合，仮想アドレス空間のレイアウトは基本的に， VAX アーキテクチャによって定義される 32 ビット仮想アドレス空間に基づいていました。 OpenVMS VAX レイアウトに基づく OpenVMS Alpha レイアウトを図 2–1 に示します。

図 2–1 32 ビット仮想アドレス空間レイアウト



OpenVMS VAX 仮想アドレス空間の下半分 (アドレス 0 〜 $7FFFFFFF_{16}$) は，**プロセス・プライベート空間**と呼ばれています。**プロセス・プライベート空間** この空間はさらに， P0 空間および P1 空間という 2 つの等しい空間に分けられます。各空間とも 1 GB 長です。 P0 空間の範囲は 0 から $3FFFFFFF_{16}$です。 P0 空間は 0 の位置で始まり，アドレスが増加する方向に向かって拡張します。一方， P1 空間の範囲は 40000000_{16}から $7FFFFFFF_{16}$です。 P1 は $7FFFFFFF_{16}$の位置で始まり，アドレスが減少する方向に向かって拡張します。

VAX 仮想アドレス空間の上半分は，**システム空間**と呼ばれています。システム空間の下半分 (アドレス 80000000_{16}〜 $BFFFFFFF_{16}$) は， S0 空間と呼ばれています。 S0 空間は 80000000_{16}で始まり，アドレスが増加する方向に向かって拡張します。

VAX アーキテクチャでは，ページ・テーブルと仮想アドレス空間の各領域を関連付けています。プロセッサは，システム・ページ・テーブルを使用して，システム空間アドレスを変換します。また，各プロセスは，プロセス自身の P0 ページ・テーブルお

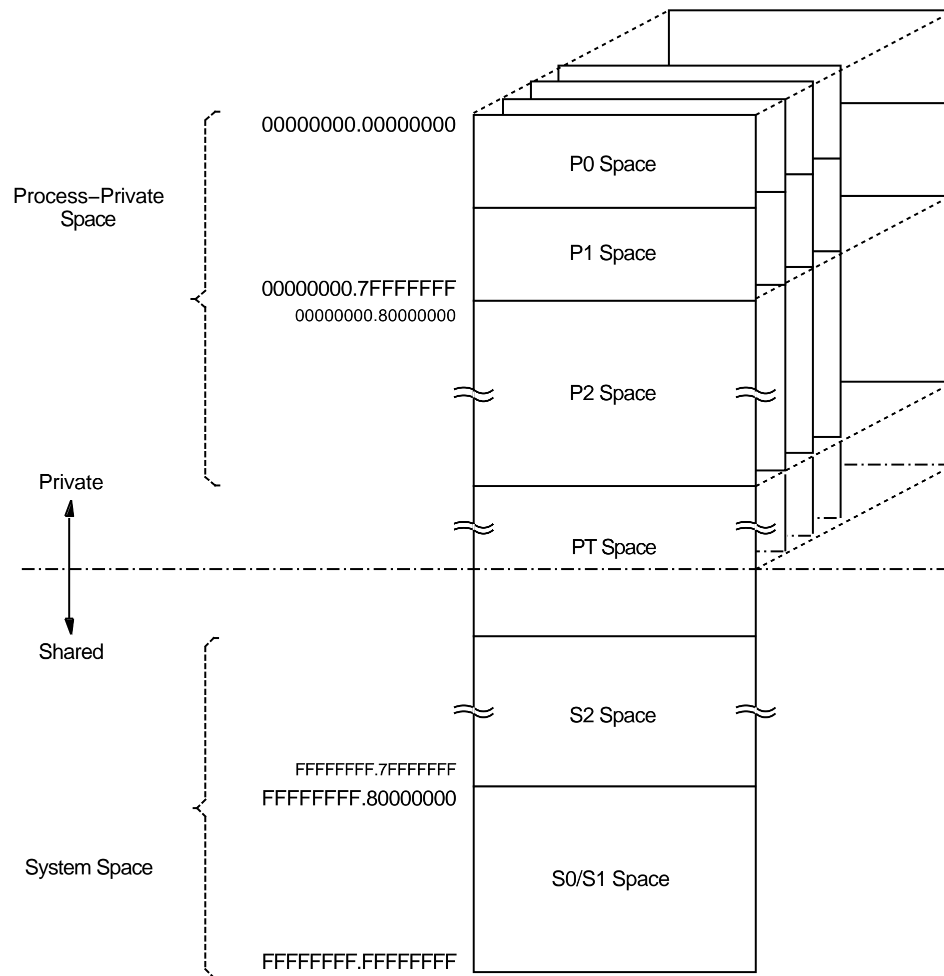
および P1 ページ・テーブルを持ちます。なお， VAX ページ・テーブルは，可能な仮想アドレス空間をすべてマップするわけではありません。作成されたそのリージョンの一部だけをマップします。

2.2 OpenVMS Alpha 64 ビット仮想アドレス空間レイアウト

OpenVMS Alpha 64 ビット・アドレス空間レイアウトは，従来の OpenVMS 32 ビット・アドレス空間レイアウトを拡張したものです。

64 ビット仮想アドレス空間レイアウトの設計を図 2–2 に示します。

図 2–2 64 ビット仮想アドレス空間レイアウト



64 ビット仮想アドレス空間レイアウトは， OpenVMS Alpha オペレーティング・システムおよびそのユーザの，現在の要望，および将来の要望を満たすように設計されています。新しいアドレス空間は，次の基本領域で構成されています。

- プロセス・プライベート空間
- システム空間
- ページ・テーブル空間

2.2.1 プロセス・プライベート空間

プロセス・プライベート・アドレス空間は， OpenVMS オペレーティング・システムのメモリ管理設計の中心です。

プロセス・プライベート空間，つまりプロセス空間は， PT 空間以下のすべての仮想アドレスを含みます。 図 2–2 に示すように，プロセス空間のレイアウトは，さらに P0， P1，および P2 空間に分かれています。 P0 空間はプログラム領域， P1 空間は制御領域， P2 空間は 64 ビット・プログラム領域です。

P0およびP1 **空間**は， VAX アーキテクチャで定義している P0 および P1 領域と同等に定義されています。両者で， 0.00000000_{16}から $0.7FFFFFFF_{16}$の範囲に相当する従来の 32 ビット・プロセス・プライベート領域を含みます。 P2 **空間**は， P1 空間のすぐ上の 0.80000000_{16}で始まり， PT 空間の最下位アドレスのすぐ下で終了する残りのプロセス空間をすべて含みます。

単一 64 ビット整数として解釈される場合， P2 空間アドレスは正の値でも負の値でも構いません。

2.2.2 システム空間

64 **ビット・システム空間**は， PT 空間を含む範囲よりも高位の 64 ビット仮想アドレス範囲全体の部分に相当します。 図 2–2 に示すように，システム空間はさらに S0, S1，および S2 空間に分かれています。

S0およびS1 **空間**は， VAX アーキテクチャで定義している S0 および S1 領域と同等に定義されています。両者で， $FFFFFFFF.80000000_{16}$から $FFFFFFFF.FFFFFFFF_{16}$ の範囲に相当する従来の 32 ビット・プロセス・プライベート領域を含みます。 S2 **空間**は， PT 空間の最高位アドレスと，結合された S0/S1 空間の最下位アドレスとの間に存在する，残りのシステム空間をすべて含みます。

S0， S1，および S2 空間は，すべてのプロセスによって完全に共用されます。 S0, および S1 は，上位方向の仮想アドレスに向かって拡張します。 S2 空間は，一般に下位方向の仮想アドレスに向かって拡張します。

システム空間内のアドレスを作成および削除できるのは，カーネル・モードで実行されるコードだけです。ただしシステム空間ページのページ保護を設定することによって，特権の少ないアクセス・モードでも読み込みおよび書き込みアクセスを認めることができます。

システム空間ベースは，S2_SIZE システム・パラメータにより制御されます。S2_SIZE は，S2 空間を予約するためのメガバイト数です。省略時の設定の値は，64 ビット (S2) システム空間を使用することが予想される要素が，それぞれ必要とするサイズに基づきます。ブート時に OpenVMS によって設定される利用者は，**ページ・フレーム数** (PFN) データベースおよびグローバル・ページ・テーブルです (SYSGEN によるシステム・パラメータの設定についての詳細は，『OpenVMS システム管理ユーティリティ・リファレンス・マニュアル (下巻)』を参照してください)。

グローバル・ページ・テーブル (GPT)，および PFN データベースは，S2 空間の最下位アドレス部分に存在します。GPT および PFN データベースを S2 空間に移動することによって，これらの領域のサイズが，小さな S0/S1 空間の制限をもはや受けることはありません。これにより OpenVMS は，より大容量の物理メモリ，およびより大きなグローバル・セクションをサポートできるようになりました。

2.2.3 ページ・テーブル空間

OpenVMS Alpha V7.0 以前のバージョンでは，**ページ・テーブル空間**(または，PT **空間**ともいう) を，複数の方法でアドレス指定することができました。PALcode TB ミス・ハンドラは，2.00000000_{16}で始まるアドレスを使用して PTE を読み込んでいました。一方，メモリ管理コードは，まず従来の 32 ビット・システム空間内にページ・テーブルのアドレスを指定します。プロセス・ページ・テーブルはプロセス・ヘッダ (PHD) 内にあり，システム空間ページ・テーブルは最高位の仮想アドレスに位置していました。これらはすべて，従来の 32 ビット・システム空間内にありました。

OpenVMS Alpha Version 7.0 の場合，ページ・テーブルのアドレスは基本的に 64 ビットの PT 空間内で指定されます。ページ・テーブル参照はこの仮想アドレス範囲に対して行われます。これらは，32 ビット共用システム・アドレス空間にはもはやありません。

図 2–2 内の点線は，プロセス・プライベート空間と共用空間の境界を表します。この境界は PT 空間にあり，プロセス・プライベート・ページ・テーブル・エントリと共用ページ・テーブル・エントリ間の境界として機能します。これらのエントリ・セットを合わせたものが，指定されたプロセスで使用できるアドレス空間全体をマップします。PT 空間は，プロセスごとに同じ仮想アドレスにマップされ，これは，$\text{FFFFFFFC.00000000}_{16}$など，通常はきわめて高位なアドレスです。

2.2.4 仮想アドレス空間のサイズ

Alpha アーキテクチャは 64 ビット・アドレスをサポートします。OpenVMS Alpha Version 7.0 は，現在の Alpha アーキテクチャ実装でサポートされる仮想アドレス空間の合計サイズを，4 GB (ギガバイト) から 8 TB (テラバイト) へと，大きく増加させました。

Alpha アーキテクチャでは，仮想アドレスを物理メモリ・アドレスに変換するとき，どの実装であっても，仮想アドレスの 64 ビットすべてを使用，またはチェックすることを要求しています。しかし， Alpha アーキテクチャの実装は，仮想アドレス空間のサブセットを実現することができます。現在の Alpha ハードウェア実装は 64 ビット仮想アドレスの中で 43 有効ビットをサポートします。この結果， 8 TB のアドレス空間が実現します。

現在の Alpha アーキテクチャ実装では，仮想アドレス内のビット 42 は，ビット 63 まで符号拡張されているか，またはその内容が引き継がれていなければなりません (最下位有効ビットは 0 から始まります)。ビット 42 からビット 63 までがすべて 0 でない，またはすべて 1 でない仮想アドレスが参照されると，アクセス違反を引き起こします。従って，有効な 8 TB のアドレス空間は，中央が“非アクセス”範囲で区切られた，別々の 2 つの 4 TB 範囲に分割されています。

OpenVMS Alpha アドレス空間のレイアウトでは，この非アクセス範囲を P2 空間に透過的に配置しています (OpenVMS Alpha メモリ管理システム・サービスは，実質的に連続したアドレス範囲を常に返します)。 OpenVMS Alpha アドレス空間レイアウトの設計の結果， P2 空間内の有効アドレスは，符号付き 64 ビット整数として解釈されるとき，正の値にも負の値にもなることができます。

32 ビット非特権コードの互換性を確保するには，有効な 32 ビット仮想アドレス内のビット 31 を引き続き使用して， P0/P1 空間内のアドレスと， S0/S1 空間内のアドレスを区別します。

2.3 仮想リージョン

仮想リージョンは，プロセス・プライベート仮想アドレスの予約範囲です。これは，実行時にユーザ・プログラムによって予約される **ユーザ定義**仮想リージョン，またはプロセス作成の中でプロセスに代わりシステムによって予約される**プロセス永久**仮想リージョンのいずれかに分けられます。

プロセスが作成されるとき， OpenVMS によって 3 つのプロセス永久仮想リージョンが定義されます。

- プログラム・リージョン (P0 空間内)
- 制御リージョン (P1 空間内)
- 64 ビット・プログラム・リージョン (P2 空間内)

この 3 つのプロセス永久仮想リージョンが存在するため，プログラマが作成するアプリケーションがアドレス空間を余分に確保する必要がない場合，仮想リージョンを作成する必要はありません。

仮想リージョンでは，アプリケーションの異なるコンポーネントが，異なる仮想リージョン内のデータを操作することができるため，アプリケーション内でのモジュール性が促進されます。仮想リージョンが作成されるとき，サービスの呼び出し者には，その仮想リージョンを識別するリージョン ID が返されます。リージョン ID は，そのリージョンの中で仮想アドレスを作成，操作，および削除するときに使用されます。アプリケーション内の異なるコンポーネントは別々の仮想リージョンを作成できるため，各自の仮想メモリの使用が競合することはありません。

仮想リージョンには次の特徴があります。

- 仮想リージョンは**軽量**オブジェクトである。つまり，指定されている仮想アドレスのページファイル制限値やワーキング・セット制限値を消費しない。新しい OpenVMS システム・サービスを呼び出すことによってユーザ定義仮想リージョンを作成すると，個別のアドレス・オブジェクトとして仮想アドレス範囲が単純に定義され，その中でアドレス空間を作成，操作，および削除することができる。
- 仮想リージョンは重なり合わない。仮想リージョン内でアドレス空間を作成するとき，プログラマは， OpenVMS システム・サービスに対してリージョン ID を指定しなければならない。プログラマは，アドレス空間を作成するリージョンを明示しなければならない。
- プログラマは，定義された仮想リージョンの範囲内にその全体が位置していないアドレス空間を作成，操作，または削除することはできない。
- 各ユーザ定義仮想リージョンのサイズは，作成された時点で決まる。 P2 空間に大規模な範囲の仮想アドレスが実現し，仮想リージョンは軽量特性を持つため，アプリケーション・コンポーネントがその仮想リージョンの中で直接必要とするよりも多くのアドレス空間を確保しても問題ない。

  プロセス永久リージョンは例外で，固定サイズを持たないことに注意すること。

  64 ビット・プログラム仮想リージョンは，作成されるときに，サイズが決定されない唯一の仮想リージョンです。プロセスの作成時， 64 ビット・プログラム・リージョンは， P2 空間をすべて含みます。ユーザ定義仮想リージョンが P2 空間に作成されると， 2 つのリージョンが重なり合うことがないように， OpenVMS メモリ管理は 64 ビット・プログラム・リージョンを縮小させます。ユーザ定義仮想リージョンが削除されると，下位の仮想アドレスにほかのユーザ定義仮想リージョンが存在しない場合， 64 ビット・プログラム・リージョンが拡張し，削除された仮想リージョン内の仮想アドレスを含みます。
- 各仮想リージョンは，それに付随するオーナ・モードおよび作成モードを持つ。アクセス・モードの特権が仮想リージョンのオーナよりも低い場合，その仮想リージョンを削除することはできない。アクセス・モードの特権が，仮想リージョンに設定されている作成モードよりも低い場合，仮想リージョン内で仮想アドレスを作成することはできない。オーナ・モードおよび作成モードは，仮想リージョンが作成されるときに設定され，変更することはできない。仮想リージョンの作成モードは，オーナ・モードより高い特権を持つことはできない。

- 仮想アドレス空間が仮想リージョン内に作成されるとき，仮想リージョン内での割り当ては一般に，プログラム・リージョン (P0 空間) および制御リージョン (P1 空間) で行われるのと同じように，稠密拡張方式で行われる。作成される時点で，各仮想リージョンは仮想アドレスに対して， P0 空間のように仮想アドレスが増加する方向に向かって，または P1 空間のように仮想アドレスが減少する方向に向かってセットアップされる。ユーザは，先頭アドレスを明示的に指定すると，この割り当てアルゴリズムを変更することができる。
- イメージのランダウン時に，ユーザ定義仮想リージョンはすべて，各リージョン内で作成されたページと共に削除される。

2.3.1 P0 空間および P1 空間内のリージョン

すべての P0 空間に対して，仮想アドレス 0 で始まり，仮想アドレス $0.3FFFFFFF_{16}$で終了するプロセス永久仮想リージョンが 1 つ存在します。これが**プログラム・リージョン**です。 P1 空間に対しても，仮想アドレス 0.40000000_{16}で始まり，仮想アドレス $0.7FFFFFFF_{16}$で終了するプロセス永久リージョンが 1 つ存在します。これが**制御リージョン**です。

プログラム・リージョンおよび制御リージョンは，カーネル・モードで所有され，ユーザの作成モードを持つものと考えられます。これは，ユーザ・モードの呼び出し者が，これらのリージョン内で仮想アドレス空間を作成することができるためです。これにより V7.0 より前のリリースの OpenVMS との上位互換性が保持されます。

このようなプログラム・リージョンおよび制御リージョンを削除することはできません。これらは，**プロセス永久**であると考えられます。

2.3.2 64 ビット・プログラム・リージョン

P2 空間は， P2 空間の最下位仮想アドレス 0.80000000_{16}から始まる，稠密拡張可能な仮想リージョンを持ちます。このリージョンを64 **ビット・プログラム・リージョン**と呼びます。 P2 空間内に 64 ビット・プログラム・リージョンを持つことによって，明示的な仮想リージョンを使用する必要がないアプリケーションは， P2 空間に仮想リージョンを作成するオーバヘッドを回避することができます。この仮想リージョンは常に存在するため，アドレスを P2 空間の中で直接作成することができます。

第 2.3.3 項で説明するように，ユーザは，占有されていない P2 空間に仮想リージョンを作成することができます。ユーザ定義仮想リージョンが， 64 ビット・プログラム・リージョンの最下位アドレスで開始するように定義されている場合，リージョン内で仮想メモリを割り当てようとすると失敗します。

リージョンは，これに付随してユーザ作成モードを持ちます。つまり，どのようなアクセス・モードでも，このリージョン内に仮想アドレス空間を作成することができます。

64 ビット・プログラム・リージョンを削除することはできません。これはプロセス永久とみなされ，イメージがランダウンしても残ります。なお，イメージのラウンダウンによって， 64 ビット・プログラム・リージョン内に作成されたすべてのアドレス空間は削除され，リージョンがリセットされて P2 空間をすべて含むことに注意してください。

2.3.3 ユーザ定義仮想リージョン

ユーザ定義仮想リージョンは，新しい OpenVMS SYS$CREATE_REGION_64 システム・サービスを呼び出すことによって作成される仮想リージョンです。ユーザ定義仮想リージョンが作成される位置は，一般的には予測できません。 64 ビット・プログラム・リージョンのための拡張領域を最大化するために， OpenVMS メモリ管理は，既存のユーザ定義リージョンより下位の， P2 空間内の最高位仮想アドレスを先頭に，仮想リージョンを割り当てます。

プロセス・プライベート・アドレス空間を最大限に制御するために，アプリケーション・プログラマは，仮想リージョンを作成するときに，先頭の仮想アドレスを指定することができます。これは，ユーザが正確な仮想メモリ・レイアウトを指定することが求められるような状況で効果的です。

仮想リージョンは，仮想アドレスが増加する方向に向かって，または減少する方向に向かって割り当てが行われるように作成することができます。これによって，スタック形式の構造を伴うアプリケーションは，仮想アドレス空間を作成し，自然に拡張することができます。

仮想リージョンを作成することによって， OpenVMS サブシステムおよびアプリケーション・プログラマは，拡張用の仮想アドレス空間を確保することができます。たとえば，アプリケーションは，大きな仮想リージョンを作成し，その仮想リージョンの中でいくつかの仮想アドレスを作成することができます。この後，アプリケーションがさらに仮想アドレス空間を必要とする場合，仮想リージョン内での拡張が可能です。その仮想リージョン内ですでに割り当てられているアドレスに実質的に連続した状態で，さらにアドレス空間を作成することができます。

SYS$CREATE_REGION_64 サービスへのフラグ引数に VA$M_P0_SPACE または VA$M_P1_SPACE を指定すると， P0 空間および P1 空間に仮想リージョンを作成することができます。

SYS$DELETE_REGION_64 システム・サービスで仮想リージョンを明示的に削除しないと，イメージが終了したときに，ユーザ定義仮想リージョンと共に，作成されたすべてのアドレス空間が削除されます。

3 システム・サービスの 64 ビット・アドレッシングのサポート

新規に，OpenVMS システム・サービスが追加されました。また，64 ビット・アドレス空間を管理することを目的として，多くの既存のサービスに変更が加えられました。本章では，64 ビット・アドレッシング，および VLM をサポートするシステム・サービスについて説明します。具体的には，64 ビット・アドレスをサポートするために既存の 32 ビット・サービスに加えられた変更について説明し，新しく追加された 64 ビット・システム・サービスの一覧を示します。

アプリケーション・プログラム内で 64 ビット・アドレッシングをサポートするシステム・サービスの例については，付録 C を参照してください。本章に示されている OpenVMS システム・サービスについての詳細は，『OpenVMS System Services Reference Manual: A–GETMSG』および『OpenVMS System Services Reference Manual: GETQUI–Z』を参照してください。

3.1 システム・サービスに関する用語

本書では，次に示すシステム・サービス定義を使用します。

32 ビット・システム・サービス

32 ビット・システム・サービスは，すべてのアドレス引数を 32 ビット・アドレスとして受け取ることが定義されているシステム・サービスです。値渡しの場合，32 ビット仮想アドレスは実際には 32 ビット符号拡張アドレスの可能性があり，この場合は 64 ビットがサービスに渡されます。

64 ビット・フレンドリ・インタフェース

64 ビット・フレンドリ・インタフェースは，すべての 64 ビット・アドレスで呼び出すことができるインタフェースです。インタフェースにまったく変更がなく，64 ビット・アドレスを処理するのに変更も必要としない場合，その 32 ビット・システム・サービス・インタフェースは，64 ビット・システム・サービス・フレンドリです。システム・サービスを実現する内部コードは，変更を必要とする場合があります。ただし，システム・サービス・インタフェースは変更の必要はありません。

バージョン 7.0 より前のバージョンの OpenVMS Alpha の OpenVMS Alpha システム・サービスの大部分は，次の理由から 64 ビット・フレンドリ・インタフェースを持っています。

- 『OpenVMS Calling Standard』では，標準ルーチンへの引数が 64 ビットになるように定義している。ルーチンの呼び出し側は 32 ビット引数を符合拡張して 64 ビットにする。
- 64 ビットの文字列ディスクリプタは，実行時に 32 ビットの文字列ディスクリプタと区別できる (64 ビットの文字列ディスクリプタについての詳細は『OpenVMS Calling Standard』参照)。
- ユーザ可視 RMS データ構造は， 32 ビット形式でない構造が使用されているかどうかを RMS ルーチンが識別できるなど，型情報を埋め込むことができる (RMS 64 ビット・アドレッシング・サポートについての詳細は， 第 5 章を参照)。

64 ビット・フレンドリ・システム・サービスの例には$QIO，$SYNCH，$ENQ，$FAO などがあります。

$CRETVA，$DELTVA，および$CRMPSC など，メモリ管理システム・サービスの大部分は， 64 ビット・フレンドリでないインタフェースを伴うルーチンです。INADR 引数および RETADR 引数配列を変更して 64 ビット・アドレスを保持することは簡単ではありません。

64 ビット・システム・サービス

64 ビット・システム・サービスは，すべての引数を 64 ビット・アドレスとして受け付けるよう定義されているシステム・サービスです。また 64 ビット・システム・サービスは，すべての仮想アドレスの 64 ビット全体を使用して， 64 ビットの仮想アドレスを渡します。

64 ビット・システム・サービスのうち， 64 ビット・アドレスを参照渡しで受け取るサービスには，_64という接尾辞がつきます。 32 ビットと 64 ビットの両方で使用できるシステム・サービスでは，これによって 64 ビット機能のバージョンと，対応する 32 ビット機能のバージョンが区別されます。一方，新規サービスの場合は，この接尾辞によって， 64 ビット長アドレス・セルが読み込み/書き込みされることが明確に示されます。埋め込み 64 ビット・アドレスを含む構造が渡されるとき，この構造が 64 ビット構造として自己識別しない場合にも，この接尾辞が使用されます。ルーチンが受け取るのは 64 ビット・ディスクリプタのため，ルーチン名に“_64”を含める必要はありません。なお，任意の値を参照渡しで渡すときには，接尾辞が必要ないことに注意してください。接尾辞が必要なのは， 64 ビット・アドレスを参照渡しで渡すときです。

3.2 64 ビット・アドレスをサポートするシステム・サービス

64 ビット・アドレスをサポートする OpenVMS Alpha システム・サービスを表 3–1 に示します。

RMS システム・サービスもいくつかの 64 ビット・アドレッシング機能を提供しますが，完全な 64 ビット・システム・サービスではないため，次の表には示されていません。詳細は，第 5 章を参照してください。

表 3–1 64 ビット・システム・サービス

サービス	引数
アライメント・システム・サービス	
$GET_ALIGN_FAULT_DATA	buffer_64, buffer_size, return_size_64
$GET_SYS_ALIGN_FAULT_DATA	buffer_64, buffer_size, return_size_64
$INIT_SYS_ALIGN_FAULT_REPORT	match_table_64, buffer_size, flags
AST システム・サービス	
$DCLAST	astadr_64, astprm_64, acmode
条件ハンドラ・システム・サービス	
$FAO	ctrstr_64, outlen_64, outbuf_64, p1_64...pn_64
$FAOL	ctrstr_64, outlen_64, outbuf_64, long_prmlst_64
$FAOL_64	ctrstr_64, outlen_64, outbuf_64, quad_prmlst_64
$GETMSG	msgid, msglen_64, bufadr_64, flags, outadr_64
$PUTMSG	msgvec_64, actrtn_64, facnam_64, actprm_64
$SIGNAL_ARRAY_64	mcharg, sigarg_64
CPU スケジューリング・システム・サービス	
$CPU_CAPABILITIES	cpu_id, select_mask, modify_mask, prev_mask, flags
$FREE_USER_CAPABILITY	cap_num, prev_mask, flags
$GET_USER_CAPABILITY	cap_num, select_num, select_mask, prev_mask, flags
$PROCESS_AFFINITY	pidadr, prcnam, select_mask, modify_mask, prev_mask, flags
$PROCESS_CAPABILITIES	pidadr, prcnam, select_mask, modify_mask, prev_mask, flags
$SET_IMPLICIT_AFFINITY	pidadr, prcnam, state, cpu_id, prev_mask
イベント・フラグ・システム・サービス	
$READEF	efn, state_64
高速 I/O システム・サービス	
$IO_CLEANUP	fandle
$IO_PERFORM	fandle, chan, iosadr, bufadr, buflen, porint
$IO_PERFORMW	fandle, chan, iosadr, bufadr, buflen, porint
$IO_SETUP	func, bufobj, iosobj, astadr, flags, return_fandle

(次ページに続く)

表 3–1 (続き) 64 ビット・システム・サービス

サービス	引数
I/O システム・サービス	
$QIO(W)[1]	efn, chan, func, iosb_64, astadr_64, astprm_64, p1_64, p2_64, p3_64, p4_64, p5_64, p6_64
$SYNCH	efn, iosb_64
ロッキング・システム・サービス	
$DEQ	lkid, vablk_64, acmode, flags
$ENQ(W)	efn, lkmode, lksb_64, flags, resnam_64, parid, astadr_64, astprm_64, blkast_64, acmode
論理名システム・サービス	
$CRELNM	attr, tabnam, lognam, acmode, itmlst
$CRELNT	ttr, resnam, reslen, quota, promsk, tabnam, partab, acmode
$DELLNM	tabnam, lognam, acmode
$TRNLNM	attr, tabnam, lognam, acmode, itmlst
メモリ管理システム・サービス	
$ADJWSL	pagcnt, wsetlm_64
$CREATE_BUFOBJ_64	start_va_64, length_64, acmode, flags, return_va_64, return_length_64, return_buffer_handle_64
$CREATE_GDZRO	gsdnam_64, ident_64, prot, length_64, acmode, flags, ...
$CRMPSC_GDZRO_64	gsdnam_64, ident_64, prot, length_64, region_id_6 4, section_offset_64, acmode, flags, return_va_64, return_length_64, ...
$CREATE_GFILE	gsdnam_64, ident_64, file_offset_64, length_64, chan, acmode, flags, return_length_64, ...
$CREATE_GPFILE	gsdnam_64, ident_64, prot, length_64, acmode, flags
$CREATE_GPFN	gsdnam_64, ident_64, prot, start_pfn, page_count, acmode, flags
$CREATE_REGION_64	length_64, region_prot, flags, return_region_id_64, return_va_64, return_length_64, ...
$CRETVA_64	region_id_64, start_va_64, length_64, acmode, flags, return_va_64, return_length_64
$CRMPSC_FILE_64	region_id_64, file_offset_64, length_64, chan, acmode, flags, return_va_64, return_length_64, ...
$CRMPSC_GFILE_64	gsdnam_64, ident_64, file_offset_64, length_64, chan, region_id_64, section_offset, acmode, flags, return_va_64, return_length_64, ...
$CRMPSC_GPFILE_64	gsdnam_64, ident_64, prot, length_64, region_id_64, section_offset_64, acmode, flags, return_va_64, return_length_64, ...

[1]64 ビット・アドレッシングをサポートする$QIO(W) 引数についての詳細は第 7 章参照。

(次ページに続く)

表 3–1 (続き) 64 ビット・システム・サービス

サービス	引数
メモリ管理システム・サービス	
$CRMPSC_GPFN_64	gsdnam_64, ident_64, prot, start_pfn, page_count, region_id_64, relative_page, acmode, flags, return_va_64, return_length_64, ...
$CRMPSC_PFN_64	region_id_64, start_pfn, page_count, acmode, flags, return_va_64, return_length_64, ...
$DELETE_BUFOBJ	buffer_handle_64
$DELETE_REGION_64	region_id_64, acmode, return_va_64, return_length_64
$DELTVA_64	region_id_64, start_va_64, length_64, acmode, return_va_64, return_length_64
$DGBLSC	flags, gsdnam_64, ident_64
$EXPREG_64	region_id_64, length_64, acmode, flags, return_va_64, return_length_64
$GET_REGION_INFO	function_code, region_id_64, start_va_64, ,buffer_length, buffer_address_64, return_length_64
$LCKPAG_64	start_va_64, length_64, acmode, return_va_64, return_length_64
$LKWSET_64	start_va_64, length_64, acmode, return_va_64, return_length_64
$MGBLSC_64	gsdnam_64, ident_64, region_id_64, section_offset_64, length_64, acmode, flags, return_va_64, return_length_64, ...
$MGBLSC_GPFN_64	gsdnam_64, ident_64, region_id_64, relative_page, page_count, acmode, flags, return_va_64, return_length_64, ...
$PURGE_WS	start_va_64, length_64
$SETPRT_64	start_va_64, length_64, acmode, prot, return_va_64, return_length_64, return_prot_64
$ULKPAG_64	start_va_64, length_64, acmode, return_va_64, return_length_64
$ULWSET_64	start_va_64, length_64, acmode, return_va_64, return_length_64
$UPDSEC_64(W)	start_va_64, length_64, acmode, updflg, efn, iosa_64, return_va_64, return_length_64, ...
プロセス制御システム・サービス	
$GETJPI(W)	efn, pidadr, prcnam, itmlst, iosb, astadr, astprm
$PROCESS_SCAN	pidctx, itmlst
時刻システム・サービス	
$ASCTIM	timlen, timbuf, timadr, cvtflg
$ASCUTC	timlen, timbuf, utcadr, cvtflg
$BINTIM	timbuf, timadr
$BINUTC	timbuf, utcadr

(次ページに続く)

表 3–1 (続き) 64 ビット・システム・サービス

サービス	引数
時刻システム・サービス	
$CANTIM	reqidt_64, acmode
$GETTIM	timadr_64
$GETUTC	utcadr
$NUMTIM	timbuf, timadr
$NUMUTC	timbuf, utcadr
$SETIME	timadr
$SETIMR	efn, daytim_64, astadr_64, reqidt_64, flags
$TIMCON	timadr, utcadr, cvtflg
他のシステム・サービス	
$CMEXEC_64	routine_64, quad_arglst_64
$CMKRNL_64	routine_64, quad_arglst_64
$GETSYI(W)	efn, csidadr, nodename, itmlst, iosb, astadr, astprm
$IDTOASC	id, namlen, nambuf, resid, attrib, contxt

3.3 符号拡張チェック

表 3–1 に示されていない OpenVMS システム・サービス，および 64 ビット・アドレスを受け取ることが明示的に指定されていないユーザ作成システム・サービスはすべて，符号拡張チェックを受けます。これらのサービスに渡される引数が正しく符号拡張されていないと， SS$_ARG_GTR_32_BITS というエラー状態が返ります。

3.4 言語の 64 ビット・システム・サービスのサポート

システム・サービスの C 関数プロトタイプは SYS$LIBRARY:SYS$STARLET_C.TLB (または STARLET) にあります。

システム・サービスに 64 ビット MACRO-32 マクロはありません。 MACRO-32 呼び出し者は， AMACRO ビルトイン EVAX_CALLG_64 マクロ，または$CALL64 マクロを使用しなければなりません。 MACRO-32 プログラミングでの 64 ビット・アドレッシングのサポートについての詳細は，第 12 章を参照してください。

3.5 C の NEW STARLET 定義

OpenVMS Alpha V7.0 以降， SYS$LIBRARY:SYS$STARLET_C.TLB (または STARLET) は，新規および拡張データ構造体定義，およびシステム・サービスの C 関数プロトタイプを提供します。新しい定義は， OpenVMS C 言語のコーディング規

則および SYS$LIBRARY:SYS$LIB_C.TLB で使用されている定義 (typedefs) との一貫性が向上しています。

既存ユーザの STARLET.H とのソース・レベルでの互換性を保持するために，**古い形式の**関数宣言および定義が，省略時の設定としてそのまま提供されています。新しいシステム・サービス関数プロトタイプおよびタイプ定義を利用するには，これらを明示的に有効にしなければなりません。

これには，__NEW_STARLET シンボルを DEC C コマンド行修飾子で定義する方法と，定義をソース・プログラムに直接指定する方法があります。次の例を参照してください。

- _NEW_STARLET シンボルを DEC C コマンド行修飾子で定義する。

  ```
  /DEFINE=(__NEW_STARLET=1)
  ```

  または

- SYS$STARLET_C.TLB ヘッダ・ファイルをインクルードする前に， C ソース・プログラムの中で__NEW_STARLET シンボルを定義する。

  ```
  #define __NEW_STARLET 1

  #include  <starlet.h>
  #include  <vadef.h>
  ```

Librarian ユーティリティを使用すると， STARLET.H の中で現在有効なシステム・サービス関数プロトタイプを確認できます。次の例を参照してください。

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

次の例は， STARLET.H に定義されている新しいシステム・サービス関数プロトタイプを示します。

```
#pragma __required_pointer_size __long

 int sys$expreg_64(
           struct _generic_64 *region_id_64,
           unsigned __int64 length_64,
           unsigned int acmode,
           unsigned int flags,
           void *(*(return_va_64)),
           unsigned __int64 *return_length_64);

#pragma __required_pointer_size __short
```

DEC C ポインタ・サイズ・プログマについての詳細は，『DEC C User's Guide for OpenVMS Systems』を参照してください。

次のソース・コード例は，プログラムの中で参照されている sys$expreg_64 関数プロトタイプを示します。

```
#define __NEW_STARLET 1              /* Enable "New Starlet" features */

#include <starlet.h>                 /* Declare prototypes for system service
s */
#include <gen64def.h>                /* Define GENERIC_64 type */
#include <vadef.h>                   /* Define VA$ constants */

#include <ints.h>                    /* Define 64-bit integer types */
#include <far_pointers.h>            /* Define 64-bit pointer types */

{
    int status;                      /* Ubiquitous VMS status value */
    GENERIC_64 region = { VA$C_P2 }; /* Expand in "default" P2 region */
    VOID_PQ p2_va;                   /* Returned VA in P2 space */
    uint64 length;                   /* Allocated size in bytes */
    extern uint64 page_size;         /* Page size in bytes */

    status = sys$expreg_64( &region, request_size, 0, 0, &p2_va, &length );
    ...

}
```

新しい関数プロトタイプで使用されるデータ構造体を表 3–2 に示します。

表 3–2 _NEW_STARLET プロトタイプが使用する構造体

プロトタイプが使用する構造体	定義されているヘッダ・ファイル	構造体メンバ名の共通接頭辞	説明
struct _cluevthndl	cluevtdef.h	cluevthndl$	クラスタ・イベント・ハンドル
struct _fabdef	fabdef.h	fab$	ファイル・アクセス・ブロック
struct _generic_64	gen64def.h	gen64$	汎用クォドワード構造体
struct _ieee	ieeedef.h	ieee$	IEEE 浮動小数点制御構造体
struct _ile2 [1]	iledef.h	ile2$	項目リスト・エントリ 2
struct _ile3 [1]	iledef.h	ile3$	項目リスト・エントリ 3
struct _ilea_64 [1]	ilea_64$	iledef.h	64 ビット項目リスト・エントリ A 構造体
struct _ileb_64 [1]	ileb_64$	iledef.h	64 ビット項目リスト・エントリ B 構造体
struct _iosa	iosadef.h	iosa$	入出力状態領域
struct _iosb	iosbdef.h	iosb$	入出力状態ブロック
struct _lksb	lksbdef.h	lksb$	ロック状態ブロック
struct _rabdef	rabdef.h	rab$	RMS レコード・アクセス・ブロック
struct _secid	seciddef.h	secid$	グローバル・セクション識別子
struct _va_range	va_rangedef.h	va_range$	32 ビット仮想アドレス範囲

[1]starlet.h 内の関数プロトタイプで，この構造体型の使用を要求するものはない。この構造体型は便宜上提供されており，適切な場所で使用できる。

4
メモリ管理 VLM 機能

本章では，次の OpenVMS Alpha メモリ管理 VLM 機能について説明します。

- メモリ常駐グローバル・セクション
- 共用ページ・テーブル
- 拡張可能なグローバル・ページ・テーブル
- 予約メモリ・レジストリ

これら多数の VLM 機能を紹介するサンプル・プログラムについては，付録 D を参照してください。

4.1 VLM 機能の概要

メモリ常駐グローバル・セクションを使用すると，データベース・サーバは，大量の“ホット”なデータを物理メモリにキャッシュしておくことができます。このため，データベース・サーバは，ディスク上のデータベース・ファイルから読み込み操作を実行することなく，物理メモリから直接データにアクセスできます。物理メモリ内のデータには，より高速にアクセスできるため，実行時の性能は大幅に向上します。

Fast I/Oは，入出力要求当たりの CPU コストを削減し，データベース操作の性能を向上します。 Fast I/O を使用するには，**バッファ・オブジェクト**を介してデータをメモリ内にロックしておく必要があります。 OpenVMS Alpha の以前のバージョンでは，バッファ・オブジェクトはプロセス・プライベート仮想アドレス空間に対してしか作成できませんでした。 OpenVMS Alpha 7.2 では，バッファ・オブジェクトは，メモリ常駐セクション内のページも含めて，グローバル・ページに対して作成できるようになりました。

共用ページ・テーブルを使用すると，同じデータベース・サーバがシステムの中で消費する物理メモリ量を削減することができます。複数のサーバ・プロセスが，大きなデータベース・キャッシュをマップする同じ物理ページ・テーブルを共用するため，OpenVMS Alpha システムはより多くのサーバ・プロセスをサポートできます。これによってシステム全般の容量が増加し，クライアント要求に対する応答時間を短縮することができます。

共用ページ・テーブルを使用すると，サーバ・プロセスは従来のグローバル・セクションの数百倍の速度でメモリ常駐グローバル・セクションをマップできるため，データベース・サーバのスタートアップ時間を大幅に短縮できます。数ギガバイトのグロ

ーバル・データベース・キャッシュを使用すると，サーバのスタートアップ時間を大幅に短縮できます。

システム・パラメータGBLPAGESとGBLPAGFILは動的パラメータです。 CMKRNL 特権を持つユーザは，実行中のシステムでこれらのパラメータの値を変更できるようになりました。 GBLPAGES パラメータの値を大きくすると，必要に応じて新しい最大サイズまで，グローバル・ページ・テーブルを拡張できます。

予約メモリ・レジストリは，メモリ常駐グローバル・セクションおよび共用ページ・テーブルをサポートします。予約メモリ・レジストリは， SYSMAN ユーティリティの中のそのインタフェースを介して，メモリ常駐セクションまたは他の特権コードで使用するためのメモリを大量に確保して， OpenVMS システムを構成することができます。また，予約メモリ・レジストリによって，前もって確保した予約メモリを考慮しながら， AUTOGEN を介して OpenVMS システムを適切にチューニングできます。

4.2 メモリ常駐グローバル・セクション

メモリ常駐グローバル・セクションは，ファイルにバックアップされないグローバル・セクションです。つまり，メモリ常駐グローバル・セクション内のページは，ページファイルや，ディスク上の他のファイルにバックアップされません。このため，プロセスやシステムがページファイル制限値を制限されることはありません。プロセスがメモリ常駐グローバル・セクションにマップしページを参照するとき，ページのワーキング・セット・リスト・エントリは作成されません。したがって，プロセスがワーキング・セット制限値には制限されることはありません。

メモリ常駐グローバル・デマンド・ゼロ (DZRO) セクション内のページには，最初は 0 が含まれています。

メモリ常駐グロバール DZRO セクションの作成は， SYS$CREATE_GDZRO システム・サービス，または SYS$CRMPSC_GDZRO_64 システム・サービスを呼び出すことによって行われます。

メモリ常駐グローバル DZRO セクションへのマップは， SYS$CRMPSC_GDZRO_64 システム・サービス，または SYS$MGBLSC_64 システム・サービスを呼び出すことによって行われます。

メモリ常駐グローバル・セクションを作成するには，プロセスに VMS$MEM_RESIDENT_USER ライト識別子が与えられていなければなりません。なお，メモリ常駐グローバル・セクションへのマップは，このライト識別子を必要としません。

メモリ常駐グローバル DZRO セクションを作成する場合，次の 2 つのオプションを使用できます。

- Fault オプション: 仮想アドレスが参照されたときだけページを割り当てる。

- Allocate オプション: セクションが作成されたときにすべてのページを割り当てる。

Fault オプション

fault オプションを使用するには，予約メモリ・レジストリを介して，システムの流動ページ・カウントからメモリ常駐グローバル・セクション内のページを差し引くことをおすすめします。なお，この操作が要求されるわけではありません。

予約メモリ・レジストリを使用すると，システムの流動ページ・カウントの計算にメモリ常駐セクション・ページを含めずに， AUTOGEN でシステムを適切にチューニングできます。 AUTOGEN は，システムの流動ページ・カウントに基づいて，システム・ページファイル，プロセス数，およびワーキング・セット最大サイズを判別します。

予約メモリ・レジストリを介してメモリ常駐グローバル・セクションが登録されていない場合，メモリ常駐グローバル・セクションを含むことができるだけの十分な流動ページがシステムにない時は，システム・サービス呼び出しは失敗します。

予約メモリ・レジストリを介してメモリ常駐グローバル・セクションが登録されている場合，グローバル・セクションのサイズが予約メモリのサイズを超えたときに，追加ページを含むことができるだけの十分な流動ページがシステムにないと，システム・サービス呼び出しは失敗します。

予約メモリ・レジストリを介してメモリが予約されている場合，そのメモリは，SYSMAN コマンドで指定されているグローバル・セクションで使用されなければなりません。メモリをシステムに戻すには， SYSMAN を実行して予約メモリを“解放”します。ページが戻され，システムの流動ページ・カウントの対象となります。

起動時にメモリ常駐グローバル・セクションの名前がわからない場合，または，システムがプールしている流動メモリを超えて大量のメモリを構成する場合，予約メモリ・レジストリ内のエントリを追加し， AUTOGEN でシステムを再チューニングすることができます。システムを再起動した後で，予約メモリを“解放”して，VMS$MEM_RESIDENT_USER ライト識別子を持つシステム内の任意のアプリケーションにこれを使用させることができます。この方法は，予約メモリを受け取るアプリケーションや名前付きグローバル・セクションを限定することなく，メモリ常駐グローバル・セクションに使用する流動メモリを増加することができます。RESERVED_MEMORY FREE コマンドについての詳細は，第 4.6.2.2 項を参照してください。

Allocate **オプション**

allocate オプションを使用するには，システムを初期化する時にあらかじめメモリを割り当てて，連続的にアラインした物理ページが使用できる状態でなければなりません。マッピングの仮想アライメントが 8 ページ， 64 ページ，または 512 ページ境界にある場合は，メモリ常駐グローバル・セクションにマップするときに粒度ヒントが使用されます (システム・ページ・サイズが 8 K バイトの場合，粒度ヒント仮想アライメントは 64 K バイト， 512 K バイト，および 4 M バイト境界にあります)。 OpenVMS は， SYS$MGBLSC などのマッピング・システム・サービスへの呼び出しにフラグ SEC$M_EXPREG が設定されている場合，粒度ヒントを使用した最適な仮想アライメントを選択します。

連続的でアラインされた PFN は，予約メモリ・レジストリを使用して予約されます。連続的でアラインされたページは，予約メモリの記述に基づいて，システムの初期化の際に割り当てられます。メモリ常駐グローバル・セクションのサイズは，予約メモリのサイズ以下でなければなりません。そうでない場合は，システム・サービス呼び出しからエラーが返ります。

メモリが予約メモリ・レジストリを介して予約されている場合，そのメモリは SYSMAN コマンドで指定されているグローバル・セクションで使用されなければなりません。メモリをシステムに戻すには， SYSMAN を実行して予約済みのメモリを解放します。予約済みのメモリが解放されると， allocate オプションを使用してメモリ常駐グローバル・セクションを作成することはできません。

4.3 グローバル・セクションのための Fast I/O とバッファ・オブジェクト

OpenVMS Alpha 7.2 では， VLM アプリケーションは，グローバル・セクションを介してプロセスで共用されるメモリに対して， Fast I/O を使用できます。 OpenVMS Alpha の以前のバージョンでは，バッファ・オブジェクトはプロセス・プライベート仮想アドレス空間に対してしか作成できませんでした。 Fast I/O を使用するには，バッファ・オブジェクトを介してデータをメモリにロックしておかなければなりません。複数のプロセスが大きなキャッシュを共用するデータベース・アプリケーションでは，次の種類のグローバル・セクションに対して，バッファ・オブジェクトを作成できるようになりました。

- ページファイルにバックアップされるグローバル・セクション
- ディスク・ファイルにバックアップされるグローバル・セクション
- メモリ常駐グローバル・セクション

バッファ・オブジェクトを使用すると， Fast I/O システム・サービスが有効になります。これらのサービスを使用すると， I/O 装置との間で高速に非常に大量の共用データを読み書きすることができます。 I/O 要求当たりの CPU コストを削減することで， Fast I/O は I/O 操作の性能を向上します。

Fast I/O は，データベース・サーバなどの VLM アプリケーションの能力を向上するので，従来より大きな容量を取り扱うことができるようになり，高いデータ・スループットを実現できます。

4.3.1 $QIO と Fast I/O の比較

$QIO システム・サービスでは，指定された範囲のメモリが存在することと，それぞれのダイレクト I/O 要求が実行される間，そのメモリにアクセスできることが必要です。バッファが存在することと，アクセス可能であることの確認は，**プローブ**と呼ぶ操作で実行されます。 I/O がアクティブな間，バッファを削除できないようにし，アクセス保護が変更されないようにするために， I/O 要求が実行されている間はメモリ・ページをロックし， I/O が終了した時点でアンロックするようにしています。

I/O で実行されるプローブとロック/アンロック操作は，コストのかかる操作です。この操作を各 I/O に対して実行するには， CPU の能力の多くを使用しなければならない可能性があります。 Fast I/O の利点は， 1 つの I/O の間だけメモリがロックされ，それ以外はページングできるという点です。

Fast I/O の場合も，バッファが使用可能であるかどうかを確認する必要がありますが，多くの I/O 要求は同じメモリ・キャッシュから実行されるため，各 I/O に対してではなく，キャッシュが 1 回だけプローブされ，ロックされる場合は，性能を向上できます。 OpenVMS では，複数の I/O の間にメモリ・アクセスが変更されないことだけを確認する必要があります。 Fast I/O では，この目標を達成するためにバッファ・オブジェクトを使用します。また， Fast I/O では，一部のシステム・リソースを前もって割り当て， I/O の流れを全般的に単純化することで，さらに性能を向上しています。

4.3.2 バッファのロックの概要

I/O サブシステムは，バッファード I/O でシステム空間からデータを移動するか，ダイレクト I/O 操作を認めることで，データをユーザ・バッファに移動できます。しかし，その前にユーザ・バッファが実際に存在することと，アクセス可能であることを確認しなければなりません。

バッファード I/O の場合，この処理は通常， I/O を要求しているプロセスのコンテキストを想定し，ターゲット・バッファをプローブすることで行われます。ほとんどの QIO 要求では，この処理は IPL 2 (IPL$_ASTDEL) で行われるため，バッファのプローブとデータの移動の間で AST が実行されることはありません。操作全体が完了するまで，バッファは削除されません。また， IPL 2 で実行することにより，データがコピーされている間，通常のページング・メカニズムが動作できます。

ダイレクト I/O の場合は通常， I/O に対してターゲット・ページをロックすることで行われます。この結果，バッファを構成するページは，ページングやスワッピングの

対象から除外されます。 I/O サブシステムは，ページ・フレーム番号，最初のページ内のバイト・オフセット， I/O 要求の長さで，バッファを識別できます。

この方法では，プロセスがページングを続行でき， I/O がまだ実行されていなかったり，アクティブな間も，バランス・セットからスワップすることができるので，柔軟性を最大限に向上できます。バッファード I/O の場合は，ページをロックする必要はありません。ダイレクト I/O の場合は，ほとんどのプロセス・ページはページングまたはスワッピングが可能です。しかし，この柔軟性を実現するために，コストが必要になります。 I/O に関係するすべてのページを，おのおのの I/O に対してプローブまたはロックし，アンロックしなければなりません。 I/O の実行回数の多いアプリケーションの場合は，オペレーティング・システムがこれらの操作に大量の時間を費やす可能性があります。

バッファ・オブジェクトを使用すると，このオーバーヘッドの大部分を回避するのに役立ちます。

4.3.3 バッファ・オブジェクトの概要

バッファ・オブジェクトとは，プロセス内の仮想アドレスに割り当てられたプロセス・エンティティです。バッファ・オブジェクトが作成されると，この範囲のアドレス内のすべてのページがメモリ内でロックされます。バッファ・オブジェクトが削除されるまで，これらのページを解放することはできません。 Fast I/O 環境では，$IO_SETUP の間，バッファ・オブジェクト自体をロックすることで，この機能を利用します。このようにすると，バッファ・オブジェクトと，そのオブジェクトに割り当てられたページが削除されるのを防止できます。バッファ・オブジェクトは$IO_CLEANUP でアンロックされます。これにより，コストのかかるプローブ，ロック，アンロック操作は， I/O バッファがバッファ・オブジェクトを越えないことを確認する単純なチェックに置き換えられます。ただし，バッファ・オブジェクトに割り当てられたページは，メモリ内で永久的にロックされるという欠点があります。アプリケーションは従来より多くの物理メモリを必要とするようになりますが，実行速度は向上します。

システム・メモリへのこの種のアクセスを制御するには，ユーザは VMS$BUFFER_OBJECT_USER 識別子を保有しなければなりません。システムはバッファ・オブジェクトで使用するために特定のページ数だけをロックすることを認めます。この数は動的 SYSGEN パラメータ MAXBOBMEM で制御されます。

第 2 のバッファ・オブジェクト・プロパティにより， Fast I/O は高い IPL でシステム・コンテキストから完全に複数の I/O 関連タスクを実行することができ，プロセス・コンテキストを想定する必要がありません。バッファ・オブジェクトが作成されると，システムはデフォルトでシステム空間のセクション (S2) を，バッファ・オブジェクトに割り当てられたプロセス・ページにマップします。このシステム空間ウィンドウは，カーネル・モードだけから読み込みアクセスと書き込みアクセスを許可するように保護されます。すべてのシステム空間はどのコンテキストの内部からも等しく

アクセスできるので，元のユーザのプロセス・コンテキストを想定するために，さらにコストのかかるコンテキスト切り換えを回避することができます。

バッファ・オブジェクト・ページへのシステム空間アクセスを可能にするには，コストが必要です。たとえば， S2 空間は通常，数ギガバイトですが， Fast I/O のために数ギガバイトのデータベース・キャッシュを多くのプロセスで共用しなければならない場合は，このサイズでも不十分な可能性があります。このような環境では，キャッシュ・バッファとの間のすべてまたは大部分の I/O はダイレクト I/O であり，システム空間マップは必要ありません。

OpenVMS バージョン 7.2 では，バッファ・オブジェクトは，システム空間ウィンドウを割り当てた状態で作成でき，割り当てずに作成することもできます。バッファ・オブジェクトが使用するリソースは次のようにチャージされます。

- ページがメモリ常駐セクションに属しているか，ページがすでに別のバッファ・オブジェクトに割り当てられている場合を除き，物理ページは MAXBOBMEM に対してチャージされます。
- デフォルトでは，システム空間ウィンドウ・ページは MAXBOBS2 に対してチャージされます。 CBO$_SVA_32 が指定されている場合は， MAXBOBS0S1 に対してチャージされます。
- CBO$_NOSVA がセットされている場合は，システム空間ウィンドウは作成されず，必要に応じて MAXBOBMEM だけがチャージされます。

Fast I/O 機能の使用の詳細については，『OpenVMS I/O User's Reference Manual』を参照してください。

4.3.4 バッファ・オブジェクトの作成と使用

バッファ・オブジェクトを作成して使用する場合，次のことに注意する必要があります。

- バッファ・オブジェクトはプロセス空間 (P0， P1， P2 のいずれか) ページにだけ関連付けることができます。
- PFN でマップされたページをバッファ・オブジェクトに関連付けることはできません。
- システム空間が関連付けられていない特殊なバッファ・オブジェクトは， Fast I/O データ・バッファを記述するためだけに使用できます。 IOSA は常にシステム空間を含む完全なバッファ・オブジェクトに関連付けなければなりません。
- データ・バッファがシステム空間を含まないバッファ・オブジェクトに関連付けられている場合は，一部の Fast I/O 操作は完全に最適化されません。 VIOC キャッシュを介したディスク読み込み I/O やバッファード I/O の完了時に，データのコピーは，完全なバッファ・オブジェクトに対してシステム・コンテキストで IPL 8 で実行される可能性があります。しかし，システム空間が割り当てられて

いないバッファ・オブジェクトの場合は，プロセス・コンテキストで実行しなければなりません。アプリケーションで独自のキャッシュを実装している場合は，IO$M_NOVCACHE ファンクション・コード修飾子を設定することで，ディスク I/O に対して VIOC を使用しないようにしてください。 Fast I/O はこの条件を認識し，バッファ・オブジェクトの種類とは無関係に，最大レベルの最適化を使用します。

4.4 共用ページ・テーブル

共用ページ・テーブルを使用すると， 2 つ以上のプロセスが同じ物理ページにマップすることができます。このとき，各プロセスがページ・テーブル作成，ページ・ファイル・アカウンティング，およびワーキング・セット制限値アカウンティングのオーバヘッドを受けることはありません。共用ページ・テーブルは，内部的には特別な種類のグローバル・セクションとして扱われており，メモリ常駐グローバル・セクションの一部であるページをマップするときに使用されます。ページ・テーブルの共用を実現する特別グローバル・セクションを，**共用ページ・テーブル・セクション**と呼びます。共用ページ・テーブル・セクション自体は，メモリ常駐です。

共用ページ・テーブルは，いくつかのシステム・サービスによって作成され，複数のプロセスに送信されます。プロセスやアプリケーションが共用ページ・テーブルを使用するのに，特別な特権やライト識別子は必要ありません。メモリ常駐グローバル・セクションを作成するときだけ， VMS$MEM_RESIDENT_USER ライト識別子が必要です。この識別子を持たないプロセスであっても， (特定のマッピング基準を満たしている限り) 共用ページ・テーブルを使用できます。

メモリ常駐グローバル・セクションで予約されるメモリと同様に，共用ページ・テーブルのメモリも，システムの流動ページ集合から除外しなければなりません。メモリ常駐グローバル・セクションが登録される時，予約メモリ・レジストリによってこの除外が行われます。

4.4.1 プライベート・ページ・テーブルのメモリ必要量

表 4–1 は，各種サイズのグローバル・セクションにマップするプライベート・ページ・テーブルおよび共用ページ・テーブルの物理メモリ必要量を，プロセス数に応じて示しています。この表から，共用ページ・テーブルを使用することで，システム全体で節約される物理メモリ量がわかります。たとえば， 100 プロセスが 1 G バイトのグローバル・セクションにマップするとき，共用ページ・テーブルでグローバル・セクションにマップすることによって， 99 M バイトの物理メモリが節約されます。

物理メモリが節約されると，物理メモリ・システム・リソースの競合が抑制されるため，システム全体の性能が向上します。各プロセスについては，ページ・テーブル・ページが使用するワーキング・セットが少なくてすむため，プロセスがより多くのプライベート・コードおよびデータを物理メモリに保持できるという利点があります。

表 4–1 ページ・テーブル・サイズ必要量

マッピングプロセスの個数	8MB		1GB		8GB		1TB	
	PPT	SHTP	PPT	SHTP	PPT	SHTP	PPT	SHTP
1	8KB	8KB	1MB	1MB	8MB	8MB	1GB	1GB
10	80KB	8KB	10MB	1MB	80MB	8MB	10GB	1GB
100	800KB	8KB	100MB	1MB	800MB	8MB	100GB	1GB
1000	8MB	8KB	1GB	1MB	8GB	8MB	1TB	1GB

PPT— プライベート・ページ・テーブル
SHPT— 共用ページ・テーブル

4.4.2 共用ページ・テーブルおよびプライベート・データ

プロセスが共用ページ・テーブルを使用するのに，特別な特権やライト識別子は必要ありません。メモリ常駐グローバル・セクションを作成するときだけ，ライト識別子 VMS$MEM_RESIDENT_USER が必要です。メモリ常駐グローバル・セクションを作成すると，予約メモリ・レジストリによって共用ページ・テーブルが必要ないと指定されていない限り，そのグローバル・セクションをマップする共用ページ・テーブルが作成されます。一見すると，このように広くデータを共用することは，本質的なセキュリティ上のリスクを伴う可能性があります。しかし，この節で説明する理由によって，セキュリティ上のリスクはありません。

共用ページ・テーブルでメモリ常駐グローバル・セクションにマップするアプリケーションまたはプロセスは，次の手順を実行しなければなりません。

1. システム・サービス SYS$CREATE_REGION_64 を呼び出すことによって，共用ページ・テーブル・リージョンを作成します。

   リージョンがページ・テーブル・ページ境界で開始および終了するように，リージョンの開始仮想アドレスは切り下げ，長さは切り上げます。

2. SYS$CRMPSC_GDZRO_64 システム・サービスまたは SYS$MGBLSC_64 システム・サービスを使用して，メモリ常駐グローバル・セクションにマップします。これらのサービスを使用すると，呼び出し者は，次の条件を満たす場合に，グローバル・セクションに関連付けられた共用ページ・テーブルを使用することができます。

   - 呼び出し者がマッピング要求で指定する読み込み/書き込みアクセス・モードが，マップするグローバル・セクションに関連するアクセス・モードに完全に一致している。
   - 呼び出し者がマッピング要求で正しい仮想アドレッシング・アライメントを指定している。

共用ページ・テーブル・リージョンは，メモリ常駐グローバル・セクションのマップのみ行えます。アプリケーションは，複数のメモリ常駐グローバル・セクションを，共用ページ・テーブル・リージョンにマップできます。共用ページ・テーブル・リージョンにマップされるグローバル・セクションの先頭仮想アドレスは，常にページ・テーブル・ページ境界に丸められます。これによって， 2 つの異なるグローバル・セクションが同じページ・テーブル・ページを共用することが避けられます。手順 2 に示した以外のシステム・サービスで，共用ページ・テーブル・リージョンに仮想アドレス空間を作成しようとしても失敗します。

注意

プロセスは非共用ページ・テーブル・リージョンを指定して，共用ページ・テーブルを伴うメモリ常駐グローバル・セクションにマップすることができます。この場合，グローバル・セクションにマップするために，プロセス・プライベート・ページ・テーブルが使用されます。

4.5 拡張可能なグローバル・ページ・テーブル

GBLPAGES システム・パラメータがグローバル・ページ・テーブルのサイズを定義します。パラメータ・ファイルに格納されている値は起動時に使用され，グローバル・ページ・テーブルの初期サイズを設定します。

OpenVMS Alpha V7.1 以降，システム・パラメータの GBLPAGES および GBLPAGFIL が変更され，ダイナミック・パラメータになりました。 CMKRNL 特権を持つユーザであれば，実行中のシステム上でこれらの値を直ちに変更できます。実行時に GBLPAGES パラメータの値を増加すると，グローバル・ページ・テーブルを必要に応じて新しい最大値まで拡張できます。なお，グローバル・ページ・テーブルを拡張または増加させるためには，次の条件をすべて満たす必要があります。

- グローバル・ページ・テーブルの連続空き領域が，要求されたグローバル・セクションを作成するのに十分でない。
- GBLPAGES パラメータの現在の設定値が，グローバル・ページ・テーブルの拡張できる値になっている。
- グローバル・ページ・テーブルの上限の位置に，グローバル・ページ・テーブルを拡張できるだけの十分な未使用の仮想メモリがある。
- グローバル・ページ・テーブルを拡張できるだけの流動メモリ (メモリにロックされていないページ) がシステムに十分にある。

グローバル・ページ・テーブルは，最低でも 6 G バイトの 64 ビット S2 空間にマップされるため，これらの条件はほとんどすべてのシステムで満たされます。極端にメモリを消費するシステムや， S2 仮想アドレス空間を大量に使用するアプリケーションを実行しているシステムに限って，要求に応じてグローバル・ページ・テーブルを拡張できないことがあります。

グローバル・ページは，他のチューニング・パラメータにも影響するシステム・リソースであるため， GBLPAGES を増加するには， AUTOGEN を使用してシステムを再起動することをおすすめします。操作上の理由で再起動できない場合は，実行中のシステム上で次のコマンドを使用することによって，パラメータを変更できます。

```
$ RUN SYS$SYSTEM:SYSGEN
SYSGEN> USE ACTIVE
SYSGEN> SET GBLPAGES  new_value
SYSGEN> WRITE ACTIVE
```

WRITE ACTIVE コマンドを実行するには， CMKRNL 特権が必要です。

同じコマンドで，グローバル・ページ・サイズの有効サイズを小さくすることもできます。次の条件が満たされる場合，グローバル・ページ・テーブルは実際に縮小され，フル・ページが流動ページとしてシステムに解放されます。

- グローバル・セクションが削除され，グローバル・ページ・テーブル・エントリが解放される。
- GBLPAGES の値が示すグローバル・ページ・テーブルのサイズが，現在のサイズよりも小さい。
- グローバル・ページ・テーブルの高位アドレスの終端に未使用のエントリが存在し，構造を縮小できる。

GBLPAGES のアクティブ値を，現在使用されているグローバル・ページ・カウントより低い値に設定しても，現在使用されているグローバル・ページに影響はありません。新たなグローバル・ページの作成が抑制されるだけです。

GBLPAGFIL パラメータのアクティブ値は，正の最大整数値まで，いつでも増加できます。 GBLPAGES と同様に， GBLPAGFIL の値を，システムのページファイルに対してページングされているグローバル・ページ数より低い値に設定しても，そのページに影響はありません。新たなグローバル・ページファイル・セクションの作成が抑制されるだけです。

なお， GBLPAGFIL を増加すると，追加のページファイル空間が必要となり，追加のページファイルをインストールしなければならないことがあるので注意してください。

4.6 予約メモリ・レジストリ

予約メモリ・レジストリは， SYSMAN ユーティリティの中のそのインタフェースを介して， OpenVMS Alpha システムに，メモリ常駐セクションおよび他の特権アプリケーションで使用するためのメモリを大量に設定できます。また，予約メモリ・レジストリを使用すると，割り当て済みの予約メモリを考慮しながら， AUTOGEN ユーティリティを介して OpenVMS システムを適切にチューニングできます。

予約メモリ・レジストリによって，次の操作を実行できます。

- システムの非流動メモリをメモリ常駐グローバル・セクションの fault オプション用に予約する。
- システムの非流動メモリに加えて割り当て済みの連続的にアラインされた物理ページを，メモリ常駐グローバル・セクションの allocate オプション用に予約する。

予約メモリ・レジストリは，システムの起動時に割り当て済みのページをゼロ化することを指定できる機能を備えています。このオプションは，メモリ常駐クローバル・デマンド・ゼロ・セクションを作成するのに必要な時間を短縮します。

予約メモリ・レジストリは，予約メモリの内，メモリ常駐グローバル・セクションにマップするのに必要なページ・テーブルのサイズを指定するオプションも備えています。このオプションが指定され，予約メモリがメモリ常駐グローバル・セクションに使用される場合，共用ページ・テーブルと共にメモリ常駐グローバル・セクションが作成されます。

4.6.1 予約メモリ・レジストリの使用

OpenVMS は，メモリ常駐グローバル・デマンド・ゼロ・セクションでの使用を目的として，非流動メモリを予約するメカニズムを備えています。予約されたメモリは，システムの非流動メモリ・サイズから単純に差し引かれたものであるか，または連続的にアラインされた物理ページとしてあらかじめ割り当てられています。

予約メモリ・レジストリを使用すると，システムの流動ページ・カウントの計算にメモリ常駐セクション・ページを含めずに， AUTOGEN でシステムを適切にチューニングできます。 AUTOGEN は，システムの流動ページ・カウントに基づいて，システム・ページファイル，プロセス数，およびワーキング・セット最大サイズを算出します。他の目的のために永久的に予約されている物理メモリを考慮していない流動ページ・カウントに基づいて AUTOGEN がパラメータを調整すると，性能に関する重大な問題がシステムで発生します。

また，予約メモリ・レジストリで allocate オプションを指定することにより，連続的にアラインされたメモリがメモリ常駐セクションで使用されます。

注意

ここでは，予約メモリ・レジストリをグローバル・セクションに対して使用する方法について説明していますが，この機能は他の特権アプリケーションに対して使用することもできます。

4.6.1.1 予約メモリ・レジストリ・データ・ファイル

予約されている非流動メモリを使用するものは，メモリの特徴を，システムの初期化 (起動) の際に読み込まれるデータ・ファイルに入力します。データ・ファイルを操作するメカニズムは， SYS$LOADABLE_IMAGES:VMS$SYSTEM_IMAGES.DATA と同様です (インストール固有のエグゼクティブ・ロード・イメージを指定します)。

このファイルの名前は次のとおりです。

```
SYS$SYSTEM:VMS$RESERVED_MEMORY.DATA
```

このファイルは， (エグゼクティブ・ロード・イメージ・データ・ファイルと同様に) SYSMAN ユーティリティで管理します。

4.6.1.2 AUTOGEN

予約メモリ・レジストリ・ファイル VMS$RESERVED_MEMORY.DATA は, AUTOGEN フィードバック・メカニズムによって読み込まれ，システムの流動ページ・カウントの設定に反映されます。 AUTOGEN は，システムの流動ページ・カウントに基づいて，システム・ページファイル，プロセス数，およびワーキング・セット最大サイズを算出します。

4.6.1.3 予約メモリ・レジストリへのエントリの追加

データ・ファイルにエントリを追加するには， SYSMAN ユーティリティを使用します。 SYSMAN コマンドは次の形式で指定します。

```
SYSMAN RESERVED_MEMORY ADD gs_name -
                    /GROUP = n -
                    /SIZE = {size of reserved memory, unit: MB} -
                    /[NO]ALLOCATE -
                    /[NO]ZERO -
                    /[NO]PAGE_TABLES
```

- gs_name フィールドには，この予約メモリに関連付けられているメモリ常駐グローバル・セクションの名前を指定する。この名前は必ず指定しなければならない。
- /GROUP 修飾子が指定されていない場合，予約メモリはシステム・グローバル・セクション (SYSGBL) で使用される。
- /GROUP 修飾子が指定されている場合，予約メモリはグループ・グローバル・セクションで使用される。値 n には，グループ・グローバル・セクションを作成するプロセスの UIC グループ番号 (8 進数) を指定する。作成者の UIC グループ番号と等しいプロセスだけがグローバル・セクションにアクセスできる。たとえば， UIC が[6,100]のプロセスがグループ・グローバル・セクションの作成者の場合， /GROUP 修飾子には 6 というグループ番号を指定する。
- /ALLOCATE 修飾子が指定されていない場合，または/NOALLOCATE 修飾子が指定されている場合，システムを次に再起動するときに予約メモリが割り当てられない。予約メモリはシステムの流動ページ・カウントから除外されるだけで，メモリ常駐グローバル・セクションの作成で fault オプションが使用される。

- /ALLOCATE 修飾子が指定されている場合，システムを次に起動するときに連続的にアラインされたページが割り当てられる。割り当てられたメモリはシステムの流動ページ・カウントから差し引かれ，メモリ常駐グローバル・セクションの作成で **allocate** オプションが使用される。ページの物理アライメントは，予約メモリのサイズのページをマップできる最大粒度ヒント係数に基づいている。粒度ヒント係数は **512** ページ (または **4 M** バイト)，あるいは **64** ページ (または **512 K** バイト) のいずれかである。このため， **8 K** バイトのシステム・ページ・サイズを仮定すると，予約メモリは次のように物理的にアラインされる。

  1. size >= 4 M バイト: 4 M バイト境界上に物理的にアラインされる。
  2. size < 4 M バイト: 512 K バイト境界上に物理的にアラインされる。

- /ZERO 修飾子が指定されていない場合，または/NOZERO 修飾子が指定されている場合，システムの初期化のときに割り当て済みのページはゼロ化されない。グローバル・セクションが作成されるときにページがゼロ化される。
- /ZERO 修飾子は，/ALLOCATE 修飾子が指定されているときだけ指定できる。/ZERO 修飾子が指定されている場合，システムの初期化のときに割り当て済みのページがゼロ化される。メモリ常駐グローバル・セクションにとってゼロ化されたページが必要であるが，必ずしもシステムの初期化のときにページをゼロ化する必要はない。
- /PAGE_TABLES 修飾子が指定されていない場合，または/NOPAGE_TABLES 修飾子が指定されている場合，共用ページ・テーブル用に追加メモリが予約されない。メモリ常駐グローバル・セクションが作成されるとき，グローバル・セクションに対して共用ページ・テーブルが作成されない。
- /PAGE_TABLES 修飾子が指定されている場合，共用ページ・テーブル用に追加メモリが予約される。メモリ常駐グローバル・セクションが作成されるとき，グローバル・セクションに対して共用ページ・テーブルが作成される。/ALLOCATE 修飾子が指定されていない場合，または/NOALLOCATE 修飾子が指定されている場合，追加予約メモリがシステムの流動ページ・カウントから除外だけされる。/ALLOCATE 修飾子が指定されている場合，システムを次に再起動するときに，連続的にアラインされた追加のページが共用ページ・テーブル用に割り当てられ，追加予約メモリがシステムの流動ページ・カウントから除外される。

4.6.2 予約メモリ・レジストリからのエントリの削除

次の SYSMAN コマンドを実行することによって，予約メモリ・エントリを削除できます。

```
SYSMAN RESERVED_MEMORY REMOVE gs_name /GROUP = n
```

gs_nameには，予約メモリ・レジストリから削除するエントリに関連付けられているメモリ常駐セクションの名前を指定します。名前は必ず指定しなければなりません。

/GROUP 修飾子に指定する値nは，削除するメモリ常駐セクションに関連付けられている UIC グループ番号 (8 進数) です。メモリ常駐グローバル・セクションがグループ・グローバル・セクションの場合は，/GROUP 修飾子を指定しなければなりません。メモリ常駐グローバル・セクションがシステム・グローバル・セクションの場合は，/GROUP 修飾子を指定しないでください。

ページ・テーブルが名前付きメモリ常駐グローバル・セクション用に予約されている場合，そのための追加予約メモリも削除されます。

REMOVE コマンドは予約メモリ・レジストリ・データ・ファイルからエントリを削除するだけで，実行中のシステム内のメモリには影響しません。

4.6.2.1 予約メモリの割り当て

システムを初期化する時に， VMS$RESERVED_MEMORY.DATA データ・ファイルが読み込まれます。

データ・ファイル内の各エントリについて， RESERVED_MEMORY ADD コマンドの/SIZE 修飾子で指定したメガバイト数が，このメモリ常駐グローバル・セクションに対するシステムの流動ページ・カウントから差し引かれます。 /PAGE_TABLES が指定されている場合，メモリ常駐グローバル・セクションをマッピングする共用ページ・テーブルが必要とするメモリ量も，システムの流動ページ・カウントから差し引かれます。

RESERVED_MEMORY ADD コマンドに/ALLOCATE が指定された場合，物理ページの連続的なまとまりが割り当てられ，メモリ常駐グローバル・セクション用に確保されます。 /PAGE_TABLES が指定された場合，物理ページの連続的なまとまりがさらに割り当てられ，共用ページ・テーブル用に確保されます。ページは，与えられたサイズのまとまりに対して，最大の粒度ヒント係数を使用するのに適した物理アライメントを持ちます。 /ZERO が指定された場合，システムの初期化の際に，またはシステムがアイドル状態のとき，ページがゼロ化されます。 /ZERO が指定されていない場合，または/NOZERO が指定された場合は，メモリ常駐グローバル・セクションが作成されるときにページがゼロ化されます。

システム・パラメータ STARTUP_P1 が MIN に設定されると，予約メモリ・レジストリ・エントリ内のエントリは無視され，メモリは予約されません。

システムの初期化の際に行われる予約メモリ・レジストリ・データ・ファイルの処理で，システム流動ページの予約や，連続的にアラインされた物理ページの割り当てに関してエラーが発生すると，コンソールにエラー・メッセージが出力され，システムは起動を続けます。

4.6.2.2 予約メモリの解放

実行中のシステムの中で次の SYSMAN コマンドを実行することによって，予約メモリを解放できます。

```
SYSMAN RESERVED_MEMORY FREE gs_name /GROUP = n
```

gs_nameには，予約メモリ・レジストリから解放するエントリに関連付けられているメモリ常駐セクションの名前を指定します。名前は必ず指定しなければなりません。

/GROUP 修飾子に指定する値nは，解放するメモリ常駐セクションに関連付けられている UIC グループ番号 (8 進数) です。メモリ常駐グローバル・セクションがグループ・グローバル・セクションの場合は，/GROUP 修飾子を指定しなければなりません。メモリ常駐グローバル・セクションがシステム・グローバル・セクションの場合は，/GROUP 修飾子を指定しないでください。

システムの初期化の際に，このグローバル・セクションに対して連続的にアラインされた物理ページがあらかじめ割り当てられなかった場合，システムの流動ページ・カウントに予約メモリが単純に追加されます。そうでなければ，システムの未使用またはゼロ化ページ・リスト上で，物理ページの割り当てが解除されます。システムの流動ページ・カウントは，割り当てを解除されたページを含むように調整されます。

名前付きのメモリ常駐グローバル・セクションに対してページ・テーブルも予約されている場合，共用ページ・テーブル用の予約メモリも解放されます。

名前付きのメモリ常駐グローバル・セクションによって予約メモリが使用されている場合，現在使用されていない予約メモリが解放されます。

RESERVED_MEMORY FREE コマンドは，予約メモリ・レジストリ・データ・ファイルの内容には影響しません。実行中のシステム内のメモリにだけ影響します。

4.6.2.3 予約メモリの表示

予約メモリ情報は，予約メモリ・レジストリ・データ・ファイルと，データ・ファイル内のエントリに基づいてシステムの初期化の際に作成される，実行中のシステム内の予約メモリ・レジストリの 2 ヶ所に保存されています。

予約メモリについての情報がどこから発生するかによって，表示メカニズムがそれぞれ異なります。

実行中のシステムの中で予約メモリ・レジストリを表示するには， SYSMAN， DCL SHOW MEMORY コマンド，および SDA という 3 種類のメカニズムがあります。

- SYSMAN

  次の SYSMAN コマンドを実行することによって，実行中のシステムの中で予約メモリ・レジストリを表示できる。

```
SYSMAN RESERVED_MEMORY SHOW gs_name /GROUP = n
```

  gs_nameには，実行中のシステムの中で表示するエントリに関連付けられているメモリ常駐グローバル・セクションの名前を指定する。 gs_name の指定を省略すると，登録されているすべてのグローバル・セクションに対する予約メモリが表示される。

  /GROUP 修飾子に指定する値nは，表示するメモリ常駐セクションに関連付けられている UIC グループ番号 (8 進数) である。メモリ常駐グローバル・セクションがグループ・グローバル・セクションの場合は，/GROUP 修飾子を指定しなければならない。メモリ常駐グローバル・セクションがシステム・グローバル・セクションの場合は，/GROUP 修飾子を指定してはならない。/GROUP 修飾子は，gs_name が指定されているときだけ指定できる。

- DCL SHOW MEMORY コマンド

  DCL SHOW MEMORY コマンドを実行することによって，実行中のシステム内の予約メモリ・レジストリを表示できる。このコマンドは，予約メモリ・レジストリをはじめ，実行中のシステムについてのメモリ関連情報をすべて表示する。

  SHOW MEMORY /RESERVED コマンドは，実行中のシステム内の予約メモリ・レジストについての情報だけを表示する。

  SHOW MEMORY コマンドによって表示される情報には，名前付きグローバル・セクションによって現在使用されているメモリ量も含まれる。また，ページ・テーブル用に予約されているメモリ量と，使用されているメモリ量 (使用されている場合) も表示される。

- SDA

  SDA もさまざまな拡張機能を備えており，実行中のシステム内の予約メモリ・レジストリに加えて，クラッシュ・ダンプ・ファイルを表示する。

4.6.2.4 予約メモリの使用

システム・サービス SYS$CREATE_GDZRO および SYS$CRMPSC_GDZRO_64 は，内部カーネル・モードの OpenVMS Alpha ルーチンを呼び出し，予約メモリ・レジストリに登録されている予約メモリを使用します。

グローバル・セクションは，予約メモリ・レジストリに登録されている必要はありません。グローバル・セクション名が予約メモリ・レジストリに登録されている場合，グローバル・セクションのサイズが，予約メモリのサイズに正確に一致している必要はありません。グローバル・セクションが登録されていない場合，またはグローバル・セクションが予約メモリ・レジストリに登録されるときに/NOALLOCATE が指定された場合，メモリ常駐グローバル DZRO セクションに対して fault オプションが使用されます。サイズが予約メモリのサイズよりも大きい場合，メモリ常駐グローバ

ル DZRO セクションを作成するシステム・サービス呼び出しは，システム内に十分な追加流動ページがない限り失敗します。

グローバル・セクションが予約メモリ・レジストリに登録されるときに/ALLOCATE が指定された場合，メモリ常駐グローバル DZRO セクションに対して allocate オプションが使用されます。グローバル・セクションのサイズは，予約されている割り当て済みのメモリのサイズ以下でなければなりません。そうでないと，システム・サービス呼び出しからエラー SS$_MRES_PFNSMALL が返されます。

4.6.2.5 予約メモリの復帰

メモリ常駐グローバル・セクションが削除された時，このグローバル・セクションに対して，連続的でアラインされた物理ページがあらかじめ割り当てられていなかった場合，このグローバル・セクションに使用されていた物理ページの割り当てが解除され，空きページ・リストに返されます。システムの流動ページ・カウントは，このグローバル・セクションについて，予約メモリ・レジストリに予約されていないページ分だけ調整されます。

メモリ常駐グローバル・セクションが削除された時，このグローバル・セクションに対して，連続的でアラインされた物理ページがあらかじめ割り当てられていた場合，このグローバル・セクション用に使用されていた物理ページが予約メモリ・レジストリに返されます。物理ページの割り当てが解除されて空きページ・リストに返されることはなく，引き続き予約されます。システムの流動ページ・カウントも調整されません。

予約メモリは， SYSMAN ユーティリティの RESERVED_MEMORY FREE コマンドでのみ実行中のシステムに解放されます。

注意

パーマネント・グローバル・セクションは， SYS$DGBLSC **の呼び出しと，グローバル・セクションへの最終リファレンスで削除されます。非パーマネント・グローバル・セクションは，グローバル・セクションへの最終リファレンスで単純に削除されます。**

4.6.3 アプリケーション構成

メモリ常駐グローバル・セクションを使用する OpenVMS Alpha アプリケーションを構成するには，次の手順を実行します。

1. SYSMAN RESERVED_MEMORY ADD コマンドを実行し，必要な予約メモリを指定します。
2. フィードバックを伴う AUTOGEN を実行し，システムの流動ページ・カウントを適切に設定し，システムのページファイル，プロセス数，およびワーキング・セット最大サイズを適切に算出します。

3. システムを再起動することによって，予約メモリをシステムの流動ページ・カウントから除外し，連続的でアラインされたページを割り当て，必要に応じてゼロ化します。

5

64 ビット・アドレッシングを対象とする RMS インタフェースの強化

本章では，64 ビット・アドレッシングをサポートし， RMS を使用して P2 または S2 空間への入出力操作を実現するために， RMS インタフェースに加えられた変更について示します。既存の RMS コードにわずかな変更を加えるだけで，この RMS 強化機能をすべて使用することができます。

RMS の 64 ビット・アドレッシングのサポートについての詳細は，『OpenVMS Record Management Services Reference Manual』を参照してください。

RMS ユーザ・インタフェースは，多くの制御データ構造 (FAB， RAB， NAM， XAB) で構成されています。これらは 32 ビット・ポインタで一緒にリンクされ，ファイル名文字列と項目リストに加えて入出力バッファを含む I/O バッファおよびさまざまなユーザ・データ・バッファに対する埋め込みポインタを含みます。 64 ビットのアドレス指定可能リージョンをサポートする RMS を使用すると，次に示すユーザ I/O バッファで 64 ビット・アドレスを使用できます。

- UBF (ユーザ・レコード・バッファ)
- RBF (レコード・バッファ)
- RHB (固定長レコード・ヘッダ・バッファ; VFC レコード形式の固定部分)
- KBF (ランダム・アクセス用のキー値を含むキー・バッファ)

ただし，RAB$L_PBFで示されるプロンプト・バッファは例外です。これは，ターミナル・ドライバが64 ビット・アドレスを使用しないためです。

64 ビット・アドレッシングを実現するため， RMS インタフェースに対して次の強化が行われました。

- 次のユーザ I/O サービスのために，データ・バッファが P2 または S2 空間に存在する。
  - レコード I/O サービス: $GET, $FIND, $PUT, $UPDATE
  - ブロック I/O サービス: $READ, $WRITE
- RAB 構造が，これらのサービスで使用するレコードおよびデータ・バッファを指す。
- 既存の RAB 構造を拡張して， 64 ビット・バッファ・ポインタとサイズ用に使用する。

- RMS ブロック I/O サービス ($READ および$WRITE) のバッファ・サイズの最大値を， 64 K バイトから 2 G バイトに増やす。ただし次の場合は例外。
  - RMS ジャーナリングの場合，ジャーナルされた$WRITE サービスは現在の最大値 (65535 からジャーナル・オーバヘッドの 99 バイトを引いた値) に制限される。最大値を超えると， RSZ エラーが返される (RAB$L_STS)。
  - 磁気テープは，デバイス・ドライバ・レベルで引き続き 65535 バイトに制限される。

  RMS レコード I/O サービス ($GET，$PUT，$UPDATE) のバッファ・サイズの最大値に変更はない。以前の RMS オン・ディスク・レコード・サイズの制限がそのまま適用される。

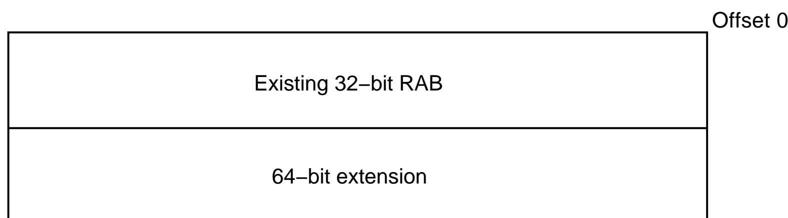
残りの RMS インタフェースは現時点では 32 ビットに制限される。

- FAB， RAB， NAM，および XAB は 32 ビット空間に引き続き割り当てなければならない。
- ファイル名へのディスクリプタ，または埋め込みポインタ，項目リストなどは，引き続き 32 ビット・ポインタでなければならない。
- RMS システム・サービスへ渡される引数は， 32 ビット引数のままである。 64 ビット引数を渡そうとすると，エラーSS$_ARG_GTR_32_BITSが返される。

5.1 RAB64 データ構造

RMS ユーザ・インタフェース構造である RAB64 は， RAB を拡張したもので， 64 ビット・バッファ・アドレスに対処することができます。 RAB64 データ構造は，図 5–1 に示すように， 32 ビットの RAB 構造と，これに続く 64 ビット拡張で構成されています。

図 5–1 RAB64 データ構造



RAB64 に含まれるフィールドは RAB フィールドとすべて同一ですが，フィールド名には， RAB 接頭辞ではなく RAB64 接頭辞が含まれます。 RAB64 はまた，拡張部に次の新フィールドを含みます。

フィールド	このフィールドの拡張	説明
RAB64$Q_CTX	RAB64$L_CTX	ユーザ・コンテキスト。 RMS はこのフィールドを使用しないが，ユーザが使用する。 CTX フィールドは，ポインタを格納しておくために使用されることがある。非同期 I/O の場合，ユーザに AST パラメータに相当する機能を提供する。
RAB64$PQ_KBF	RAB64$L_KBF	ランダム・アクセスするためのキー値を含むキーバッファ・アドレス ($GET および$FIND)。
RAB64$PQ_RBF	RAB64$L_RBF	レコード・バッファ・アドレス ($PUT，$UPDATE，および$WRITE)。
RAB64$PQ_RHB	RAB64$L_RHB	レコード・ヘッダ・バッファ・アドレス (VFC レコード形式の固定部分)。
RAB64$Q_RSZ	RAB64$W_RSZ	レコード・バッファ・サイズ。
RAB64$PQ_UBF	RAB64$L_UBF	ユーザ・バッファ・アドレス ($GET および$READ)。
RAB64$Q_USZ	RAB64$W_USZ	ユーザ・バッファ・サイズ。

名前に PQ タグを含むフィールドは， 64 ビット・アドレス，または 32 ビット・アドレスの符合拡張による 64 ビット・アドレスのいずれも持つことができます。そのため 64 ビット・アドレスの使用に関わらず，すべてのアプリケーションでこれらのフィールドを使用することができます。

大部分のレコード I/O サービス要求では，デバイスとユーザのデータ・バッファとの間に RMS 内部バッファがあります。 RMS サービスの$PUT は唯一の例外です。デバイスがユニット・レコード・デバイスでネットワーク越しにアクセスされない場合， RMS はユーザ・レコード・バッファ (RBF) のアドレスを$QIO システム・サービスに渡します。 64 ビット・アドレス空間をサポートしないレコード単位取り扱い装置を対象として， 64 ビット・アドレス空間に割り当てられているレコード・バッファ (RBF) を不適切に$PUT に指定すると， SS$_NOT64DEVFUNC が返されます ($QIO についての詳細は第 7 章を参照してください)。 RMS は RAB64$L_STV の 2 番目の状態値として SS$NOT64DEVFUNC でエラー状態 RMS$_SYS を返します。

RMS システム・サービスは， RAB64 構造と RAB 構造をサポートします。

5.2 64 ビット RAB 拡張の使用

アプリケーションが64 ビット RMS サポートを使用するには，最小限のソース・コードの変更だけが必要です。

RMS では， RAB を使用できる場所で， RAB64 を使用することを認めています。たとえば，ある RMS レコードやブロック I/O サービスに渡される先頭引数として，RAB の代わりに RAB64 を使用することができます。

RAB64 は，既存の RAB の拡張で上位互換性があります。そのため大部分のソース・モジュールは， RAB64 内のフィールドへの参照を，まるで RAB への参照であるかのように取り扱うことができます。一方， 64 ビット・バッファ・アドレスの場合は，次の 2 つの条件が満たされるときに限って使用されます。

- RAB64$B_BLN フィールドが RAB64$C_BLN64 に初期化され，拡張が存在することを示している。
- RAB の 32 ビット部分内の 32 ビット・アドレス・フィールドが –1 を含んでいる。

クォドワード・サイズ・フィールド内の値は， 32 ビット・アドレス・フィールドの内容がその使用を指定しているときに限って使用されます。この例を次に示します。

このアドレス・フィールドが -1 を含む場合	このフィールド内のアドレスが使用される	このフィールド内のサイズが使用される
RAB64$L_UBF	RAB64$PQ_UBF[1]	RAB64$Q_USZ
RAB64$L_RBF	RAB64$PQ_RBF[1]	RAB64$Q_RSZ
RAB64$L_KBF	RAB64$PQ_KBF	RAB64$B_KSZ
RAB64$L_RHB	RAB64$PQ_RHB	FAB$B_FSZ

[1] このフィールドは， 64 ビット・アドレス，または 64 ビットに符号拡張された 32 ビット・アドレスのいずれかを含むことができる。

RMS は， RAB を使用できる場所での RAB64 の使用を認めていますが，ソース言語によっては，ほかの制限が課せられる場合があります。詳細は，使用しているソース言語のマニュアルを参照してください。

5.3 ユーザ RAB 構造をサポートするマクロ

次の新しい MACRO-32 マクロおよび BLISS マクロが実装され，ユーザ RAB 構造に対する 64 ビット拡張をサポートします。

- MACRO-32 マクロ
  - $RAB64 ($RAB のカウンタパート)
  - $RAB64_STORE ($RAB_STORE のカウンタパート)

  これらのマクロを使用することによって，次の状態が発生する。
  - RAB$B_BLN にRAB$C_BLN64の定数が代入される。
  - 元のロングワード I/O バッファが–1 に初期化され， USZ および RSZ ワード・サイズが 0 に初期化される。
  - UBF， USZ， RBF， RSZ， RHB，または KBF キーワードを使用して指定した値が，これらのキーワードのクォドワード・フィールドに移動する (これに対して，$RAB および$RAB_STOREマクロは，これらの値を，これらのキーワードのロングワード[またはワード]フィールドに移動する)。

• BLISS マクロ

次の BLISS マクロは， STARLET.R64 ライブラリでだけ使用できる。これは，マクロが使用する QUAD キーワードが， BLISS-64 でだけ使用できるためである。このため，このマクロを参照する BLISS ルーチンはすべて， BLISS-64 コンパイラを使用してコンパイルされなければならない。

– $RAB64 ($RAB のカウンタパート)
– $RAB64_INIT ($RAB_INIT のカウンタパート)
– $RAB64_DECL ($RAB_DECL のカウンタパート)

先頭の 2 つのマクロ ($RAB64 および$RAB64_INIT) を使用することによって，次の状態が発生する。

– RAB$B_BLN にRAB$C_BLN64の定数が代入される。
– 元のロングワード I/O バッファが–1 に初期化され， USZ および RSZ ワード・サイズが 0 に初期化される。
– キーワード UBF， USZ， RBF， RSZ， RHB， KBF に代入された値は，これらのキーワードのクォドワード・フィールドに移動される (これに対して，$RAB および$RAB_INITマクロは，これらの値を，これらのキーワードのロングワード[またはワード]フィールドに移動する)。

3 番目のマクロ ($RAB64_DECL) は，RAB$C_BLN64の長さのバイトのブロック構造を割り当てる。

6 ファイル・システムの 64 ビット・アドレッシングのサポート

Files–11 On-Disk Structure Level 2 (ODS-2) を実現する Extended QIO Processor (XQP) ファイル・システム，および Magnetic Tape Ancillary Control Process (MTAAACP) は共に，仮想読み込み関数と仮想書き込み関数における， 64 ビット・バッファ・アドレスの使用をサポートします。

XQP および ACP は，ファイルへの仮想 I/O 要求を，デバイスへの 1 つ以上の論理 I/O 要求に変換します。 XQP または ACP 要求で指定されるバッファはデバイス・ドライバに渡されるため， P2 または S2 空間内のバッファのサポートも， XQP および ACP で使用されるデバイス・ドライバに依存します。

OpenVMS が提供するディスク・ドライバおよびテープ・ドライバはすべて，仮想，論理，および物理的な読み込み関数と書き込み関数において，ディスク・デバイスとテープ・デバイスを対象とするデータ転送で 64 ビット・アドレスをサポートします。したがって， XQP および Magnetic Tape ACP は，仮想読み込み関数と仮想書き込み関数において， P2 または S2 空間内のバッファをサポートします。

XQP および ACP は，制御関数 (IO$_ACCESS， IO$_DELETE， IO$_MODIFY など) において， P2 または S2 空間内のバッファをサポートすることはしません。

64 ビット・バッファ・アドレスをサポートするデバイス・ドライバについての詳細は，第 7 章を参照してください。

7

OpenVMS Alpha デバイスの 64 ビット・アドレッシングのサポート

RMS サービス，$QIO システム・サービス，および OpenVMS Alpha システムが提供するほとんどのデバイス・ドライバによる入出力操作は， P2 または S2 空間と直接行われます。

本章では$QIO システム・サービスが 64 ビット・アドレスをサポートする方法について説明します。また 64 ビット・アドレスをサポートする/しない OpenVMS Alpha デバイス・ドライバ， 64 ビット・アドレスをサポートする OpenVMS Alpha ディスクおよびテープ・ドライバの一覧を示します。

ユーザが作成したデバイス・ドライバは， 64 ビット・アドレスをサポートするよう修正することができます。詳細は『**OpenVMS Alpha Guide to Upgrading Privileged-Code Applications**』を参照してください。すべての関数内で 64 ビット・バッファ・アドレスをサポートするよう修正したデバイス・ドライバの例については， SYS$EXAMPLES ディレクトリの LRDRIVER デバイス・ドライバを参照してください。

重要

OpenVMS Alpha バージョン 7.0 は， OpenMVS Alpha 特権インタフェースおよびデータ構造が大幅に変更されています。そのため V7.0 より前のバージョンの OpenVMS Alpha で使用していたユーザ作成の特権コード・アプリケーション，およびデバイス・ドライバを OpenVMS Alpha V7.0 以降で正しく実行するためには，再コンパイルおよび再リンクしなければなりません。

必要な変更を V7.0 で行った場合は，ユーザが作成した特権コード・アプリケーションを再コンパイルおよび再リンクする必要はありません。

再コンパイルおよび再リンクについての詳細は，『OpenVMS Alpha Guide to Upgrading Privileged-Code Applications』または『OpenVMS V 7.2 リリース・ノート[翻訳版]』を参照してください。

7.1 $QIO の 64 ビット・アドレスのサポート

$QIO および$QIOW システム・サービスは，次の引数を取ることができます。

```
$QIO[W] efn,chan,func,iosb,astadr,astprm,p1,p2,p3,p4,p5,p6
```

これらのサービスは，(第 3 章で説明されている) 64 ビット・フレンドリ・インタフェースを持ち，64 ビット・アドレスをサポートすることができます。

64 ビット・アドレスのサポートを目的として，$QIO および$QIOW システム・サービス引数のデータ型に加えられた変更を表 7–1 に示します。

表 7–1　$QIO[W]引数の変更

引数	以前の型	新しい型	説明
efn	符号なし ロングワード	-	イベント・フラグ番号。変更なし。
chan	符号なしワード	-	チャネル番号。変更なし。
func	符号なし ロングワード	-	I/O 関数コード。変更なし。
iosb	32 ビット・ポインタ[1]	64 ビット・ポインタ	クォドワード I/O 状態ブロック (IOSB) へのポインタ。IOSB 形式は変更なし。
astadr	32 ビット・ポインタ[1]	64 ビット・ポインタ	呼び出し者の AST ルーチンのプロシージャ値。Alpha システムでは，プロシージャ値はプロシージャ・ディスクリプタへのポインタである。
astprm	符号なし ロングワード[2]	クォドワード	AST ルーチンの引数値。
P1	ロングワード[2]	クォドワード	デバイスに依存する引数。P1 はしばしばバッファ・アドレスである。
P2	ロングワード[2]	クォドワード	デバイスに依存する引数。下位 32 ビットだけが，P2 をバッファ・サイズとして使用するシステム提供の FDT ルーチンによって使用される。
P3	ロングワード[2]	クォドワード	デバイスに依存する引数。
P4	ロングワード[2]	クォドワード	デバイスに依存する引数。
P5	ロングワード[2]	クォドワード	デバイスに依存する引数。
P6	ロングワード[2]	クォドワード	デバイスに依存する引数。場合によって P6 は，診断バッファのアドレスを含むために使用される。

[1]32 ビット・ポインタは，『OpenVMS Calling Standard』の要求に従い，64 ビットに符号拡張されていた。

[2]32 ビット・ロングワード値は，『OpenVMS Calling Standard』の要求に従い，64 ビットに符号拡張されていた。

通常，$QIO P1 引数はバッファ・アドレスを指定します。読み込み関数および書き込み関数をサポートする，システム提供の上位レベル FDT ルーチンはすべて，この規則を使用します。P1 引数によって，$QIO サービスの呼び出し者が，64 ビット・サポートを要求するかどうかが決定されます。$QIO システム・サービスが 64 ビットの I/O 要求を拒否すると，次に示す回復不可能なシステム・エラー状態が返ります。

```
SS$_NOT64DEVFUNC  64-bit address not supported by device for this function
```

この回復不可能な状態値は，次の環境のもとで返されます。

- 呼び出し者が，デバイスに依存する P1 引数に 64 ビット仮想アドレスを指定したが，デバイス・ドライバが，要求された I/O 関数で 64 ビット・アドレスをサポートしない。
- 呼び出し者が，診断バッファに対して 64 ビット・アドレスを指定したが，デバイス・ドライバが，診断バッファで 64 ビット・アドレスをサポートしない。
- 64 ビット・バッファ・アドレスが P2 〜 P6 引数を使用して渡され，ドライバが，要求された I/O 関数で 64 ビット・アドレスをサポートしない場合，デバイス・ドライバによってはこの状態値を返すことがある。

$QIO，$QIOW，および$SYNCH システム・サービスについての詳細は，『OpenVMS System Services Reference Manual: GETQUI–Z』を参照してください。

7.2 64 ビット・アドレスをサポートする OpenVMS ドライバ

デバイス・ドライバは， 64 ビット・アドレスのサポートを I/O 関数コードごとに宣言します。ディスク・デバイス・ドライバおよびテープ・デバイス・ドライバは，仮想，論理，および物理的な読み込み関数と書き込み関数において，ディスク・デバイスとテープ・デバイスを対象とするデータ転送で 64 ビット・アドレスをサポートします。たとえば， OpenVMS SCSI ディスク・クラス・ドライバの SYS$DKDRIVER は， IO$_READVBLK 関数および IO$_WRITEVBLK 関数において 64 ビット・アドレスをサポートしますが， IO$_AUDIO 関数ではサポートしません。

64 ビット・アドレスをサポートする OpenVMS Alpha デバイス・ドライバには，次のドライバが含まれます。

- すべてのディスクおよびテープ・ドライバ
- ディスクおよびテープ・ドライバの下のすべてのポート・ドライバ
- LAN ドライバ
- メールボックス・ドライバ
- ISA パラレル・ポート・ドライバ (LRDRIVER.C)

少なくとも 1 つの関数で 64 ビット・アドレスをサポートする OpenVMS Alpha デバイス・ドライバを表 7–2 に示します。

表 7–2　64 ビット・アドレスをサポートするドライバ

ドライバ	説明
SYS$DADDRIVER	ローカル・エリア・ディスク・クライアント・ディスク・ドライバ
SYS$DKDRIVER	SCSI ディスク・クラス・ドライバ
SYS$DUDRIVER	DSA ディスク・クラス・ドライバ
SYS$DVDRIVER	Intel 83077AA のフロッピー・ディスク
SYS$ECDRIVER	LAN driver for PMAI
SYS$ERDRIVER	LAN driver for DE422
SYS$ESDRIVER	LAN driver for DESUA
SYS$EWDRIVER	TULIP LAN, PCI
SYS$EXDRIVER	DEMNA LAN, XMI
SYS$EZDRIVER	SGEC/INEC/TGEC LAN
SYS$FADRIVER	FDDI for Futurebus
SYS$FCDRIVER	DEFZA, DEFTA LAN, TC
SYS$FRDRIVER	DEFEA LAN, EISA
SYS$FXDRIVER	DEMFA LAN, XMI
SYS$GKDRIVER	SCSI 汎用クラス・ドライバ
SYS$HCDRIVER	OTTO クラス ATM
SYS$ICDRIVER	TMS380 LAN, TC
SYS$IRDRIVER	TMS380 EISA トークン・リング
SYS$LADDRIVER	Local Area Disk
SYS$LASTDRIVER	Local Area System Transport
SYS$LRDRIVER	VL82C106 parallel printer driver
SYS$MADDRIVER	ローカル・エリア・クライアント・テープ
MBDRIVER	メールボックス・ドライバ
SYS$MKDRIVER	SCSI テープ・クラス・ドライバ
NLDRIVER	ヌル・デバイス・ドライバ
SYS$PADRIVER	SHAC CI および DSSI ポート・ドライバ
SYS$PEDRIVER	NI SCS ポート・ドライバ
SYS$PIDRIVER	NCR 53C710 DSSI ポート
SYS$PKCDRIVER	SCSI NCR 53C94 ポート
SYS$PKEDRIVER	NCR 53C810 SCSI ポート
SYS$PKJDRIVER	ADAPTEC 1742A SCSI ポート
SYS$PKSDRIVER	SIMport TC-SCSI ポート
SYS$PKTDRIVER	NCR 53C710 SCSI ポート
SYS$PKZDRIVER	XZA SCSI ポート
SYS$PNDRIVER	NPORT SCS ポート
SYS$PUDRIVER	CI UDA ポート・ドライバ
SYS$SHDRIVER	ボリューム・シャドウィング

(次ページに続く)

表 7–2 (続き) 64 ビット・アドレスをサポートするドライバ

ドライバ	説明
SYS$TUDRIVER	MSCP/DSA テープ・クラス
SYS$WPDRIVER	ウォッチポイント・ドライバ

OpenVMS Alpha バージョン 7.0 で 64 ビット・アドレスをサポートしない OpenVMS Alpha デバイス・ドライバを表 7–3 に示します。

表 7–3 32 ビット・アドレスに制限されるドライバ

ドライバ	説明
SYS$CTDRIVER	CTERM ドライバ
SYS$FBDRIVER	ターミナル・フォールバック・ドライバ
SYS$FTDRIVER	擬似ターミナル・ドライバ
SYS$FYDRIVER	DUP DSA プロトコル・クラス・ドライバ
SYS$GQADRIVER	QVISION ドライバ
SYS$GTADRIVER	DECwindows TX driver for Flamingo
SYS$GXADRIVER	Flamingo CXTurbo (aka SFB, aka HX) ドライバ
SYS$GYADRIVER	SFB+ aka HX+, aka FFB ドライバ
SYS$GYBDRIVER	PCI バス上の TGA グラフィクス用ドライバ
SYS$IEDRIVER	DECwindows 拡張
SYS$IKBDRIVER	DECwindows PCXAL キーボード
SYS$IKDRIVER	DECwindows LKxxx キーボード
SYS$IMBDRIVER	DECwindows PCXAS (PS2) マウス
SYS$IMDRIVER	DECwindows VSxxx マウス
SYS$INDRIVER	DECwindows 入力ドライバ
SYS$LTDRIVER	LAT ターミナル・ドライバ
NDDRIVER	DECnet Phase IV DLE (MOP サポート)
NETDRIVER	DECnet Phase IV
SYS$RTTDRIVER	リモート DECnet ターミナル・ドライバ
SYS$SODRIVER	AMD79C30A Audio/ISDN ドライバ
SYS$TTDRIVER	ターミナル・クラス・ドライバ
DECW$XTDRIVER	X ターミナル・クラス・ドライバ
SYS$YRDRIVER	Z85C30 SCC ターミナル・ポート・ドライバ
SYS$YSDRIVER	PC87312 ターミナル・ポート・ドライバ

32 ビット・バッファ・アドレスに制限されるドライバについて，特に次の点に注意してください。

- ターミナル・ドライバは 64 ビット・アドレスをサポートしない。
- DECwindows Motif ソフトウェアで使用されるドライバは 64 ビット・アドレスをサポートしない。

- DECnet Phase IV ドライバは 64 ビット・アドレスをサポートしない。

7.3 64 ビット・アドレスをサポートする機能コード

64 ビット・アドレスをサポートする OpenVMS Alpha I/O 機能コードを 表 7–4 に示します。

表 7–4 64 ビット・アドレスが使用できる機能コード

ドライバのタイプ	機能コード	64 ビット・アドレス
ディスク		
	IO$_READLBLK	P1
	IO$_READPBLK	P1
	IO$_READVBLK	P1
	IO$_WRITECHECK	P1
	IO$_WRITELBLK	P1
	IO$_WRITEPBLK	P1
	IO$_WRITEVBLK	P1
磁気テープ		
	IO$_READLBLK	P1
	IO$_READPBLK	P1
	IO$_READVBLK	P1
	IO$_WRITELBLK	P1
	IO$_WRITEOF	P1
	IO$_WRITEPBLK	P1
	IO$_WRITEVBLK	P1
メールボックス		
	IO$_READLBLK	P1
	IO$_READPBLK	P1
	IO$_READVBLK	P1
	IO$_WRITELBLK	P1
	IO$_WRITEPBLK	P1
	IO$_WRITEVBLK	P1
Local Area Network (LAN)		
	IO$_READLBLK	P1,P5
	IO$_READPBLK	P1,P5
	IO$_READVBLK	P1,P5
	IO$_WRITELBLK	P1,P4,P5
	IO$_WRITEPBLK	P1,P4,P5
	IO$_WRITEVBLK	P1,P4,P5

7.4 SCSI クラス・ドライバ用の 64 ビット IO$_DIAGNOSE 関数

$QIO IO$_DIAGNOSE 関数は強化され， SCSI クラス・ドライバの GKDRIVER, DKDRIVER，および MKDRIVER に対して 64 ビット・アドレッシングをサポートするようになりました。つまり S2DGB の中で指定される仮想アドレスは，ユーザ・アプリケーションの要求に応じて 64 ビット仮想アドレスになります。

$QIO IO$_DIAGNOSE 引数を次に示します。

引数	目的
P1	S2DGB 基底アドレス
P2	S2DGB の長さ
P3	予約。 0 を指定する。
P4	予約。 0 を指定する。
P5	予約。 0 を指定する。
P6	予約。 0 を指定する。

STARLET で定義される SCSI Diagnose Buffer (S2DGB) は， 32 ビット・アドレッシングと 64 ビット・アドレッシングの 2 つの形式を認めています。 32 ビット形式は OpenVMS Alpha バージョン 6.2 でサポートされている形式と同じです。 32 ビット S2DGB 形式を図 7–1 に， 64 ビット S2DGB 形式を図 7–2 に示します。

図 7–1　32 ビットの OpenVMS SCSI-2 Diagnose Buffer (S2DGB) のレイアウト

Field	Offset
S2DGB$L_OPCODE	:00
S2DGB$L_FLAGS	:04
S2DGB$L_32CDBADDR	:08
S2DGB$L_32CDBLEN	:0C
S2DGB$L_32DATADDR	:10
S2DGB$L_32DATLEN	:14
S2DGB$L_32PADCNT	:18
S2DGB$L_32PHSTMO	:1C
S2DGB$L_32DSCTMO	:20
S2DGB$L_32SENSEADDR	:24
S2DGB$L_32SENSELEN	:28
Reserved Should Be Zero	:2C
	:30
	:34
	:38

ZK−8486A−GE

図 7–2 64 ビットの OpenVMS SCSI-2 Diagnose Buffer (S2DGB) のレイアウト

フィールド	オフセット
S2DGB$L_OPCODE	:00
S2DGB$L_FLAGS	:04
S2DGB$PQ_64CDBADDR	:08
S2DGB$PQ_64DATADDR	:10
S2DGB$PQ_64SENSEADDR	:18
S2DGB$L_64CDBLEN	:20
S2DGB$L_64DATLEN	:24
S2DGB$L_64SENSELEN	:28
S2DGB$L_64PADCNT	:2C
S2DGB$L_64PHSTMO	:30
S2DGB$L_64DSCTMO	:34
Reserved. Should be Zero	:38

ZK−8487A−GE

ユーザ・アプリケーションは，S2DGB$L_OPCODE で形式値を渡すことによって，2 種類の S2DGB 形式のうちどちらの形式を使用するのかを指定しなければなりません。特に S2DGB$L_OPCODE には，32 ビット形式を要求する OP_XCDB32 (= 1)，または 64 ビット形式を要求する OP_XCDB64 (= 2) を割り当てる必要があります。OP_XCDB64 の値が指定されると，ユーザ・アプリケーションは 64 ビットの S2DGB 形式の使用と，特に，次に説明する S2DGB フィールドに対して 64 ビット名の使用が求められます。同様に，オペレーティング・コードの OP_XCDB32 は，ユーザ・アプリケーションがそのフィールドに対して 32 ビット名を使用することを求めます。

構造の正しい長さは，定数 S2DGB$K_XCDB64_LENGTH (値: 60 - 10 進) と同様に，定数 S2DGB$K_XCDB32_LENGTH (値: 60 - 10 進) によって定義されます。

S2DGB 内のフィールドを次に定義します。フィールドの名前が 32 ビットと 64 ビットとでそれぞれ異なる場合は，まず最初に 32 ビット名を示し，次に 64 ビット名を括

弧で囲んで示します。アドレスを格納するフィールドを除いて，すべてのフィールドは符号なしのロングワードです。

S2DGB$L_OPCODE

このフィールドは，ユーザ・アプリケーションが S2DGB のほかのフィールドに 32 ビット仮想アドレスを指定するか，または 64 ビット仮想アドレスを指定するかに応じて， S2DGB$K_OP_XCDB32 または S2DGB$K_OP_XCDB64 を含みます。

S2DGB$L_FLAGS

このフィールドは，次の表に示すビット・フィールドを含みます。これらのビット定義はビット 0 で始まり，省略されるビットはありません。これは， Alpha OpenVMS V6.1 以前で使用可能な IO$_DIAGNOSE インタフェースとの互換性のために必要です。

S2DGB$V_READ	実行されている操作が読み込みの場合，このビットは 1 である。操作が書き込みの場合は 0 である。
S2DGB$V_DISCPRIV	このビットには，この操作で送られる IDENTIFY メッセージの中で使用される DiscPriv ビット値が含まれる。 S2DGB$V_TAGGED_REQ が 1 の場合，このビットは無視される。なお，ポートによってはこのビットが無視される場合があるので注意する。
S2DGB$V_SYNCHRONOUS	このビットは，その値を SISC-2 ドライバの中でユーザが制御できないために無視される。
S2DGB$V_OBSOLETE1	このビットは無視される。以前のリリースではこのビットは，コマンドの再送の無効化を表していたが，現在は， SISC-2 ドライバの中でユーザが制御できないために無視される。
S2DGB$V_AUTOSENSE	このビットが 1 の場合， S2DGB$L_32SENSEADDR および S2DGB$L_32SENSELEN は，有効なセンス・バッファ・アドレスと長さを含む。 CHECK CONDITION または COMMAND TERMINATED 状態が返される場合， REQUEST SENSE データは， S2DGB$L_32SENSEADDR および S2DGB$L_32SENSELEN で定義されるバッファに返される。 S2DGB$V_AUTOSENSE が 0 の場合， S2DGB$L_32SENSEADDR および S2DGB$L_32SENSELEN で定義されるバッファは無視される。このような場合，クラス・ドライバはオートセンス・データをプールに保存し，これを次の IO$_DIAGNOSE に返す。ただしこれは， IO$_DIAGNOSE が REQUEST SENSE CDB を持つ場合に限られる。 S2DGB$L_FLAGS 内のほかのすべてのビットは 0 です。

S2DGB$V_TAGGED_REQ

このビットが 1 の場合，タグ付きコマンド・キュー登録を使用しているものとして操作が処理され，S2DGB$V_TAG は使用するタグ値を定義しておく必要がある。このビットが 0 の場合，タグ付きコマンド・キュー登録を使用せずに操作が処理される。タグ付きコマンド・キュー登録をサポートしないポートは常に，このビットが 0 として動作する。なお，ポートによっては，適切なタグ付き操作を使用して，非タグ付き操作をシミュレーションするので注意する。S2DGB$V_TAGGED_REQ が 1 の場合，この 3 ビットのフィールドは次のコード化された定数値を含まなければならない。

S2DGB$K_SIMPLE は，コマンドが SIMPLE キュー・タグで送られることを指定する。

S2DGB$K_ORDERED は，コマンドが ORDERED キュー・タグで送られることを指定する。

S2DGB$K_EXPRESS は，コマンドが HEAD OF QUEUE キュー・タグで送られることを指定する。

S2DGB$V_TAGGED_REQ が 0 の場合，このフィールドは無視される。タグ付きコマンド・キュー登録をサポートしないポートは，常に S2DGB$V_TAG フィールドを無視し，すべてのコマンドを非タグ付き操作として送信する。

IO$_DIAGNOSE 関数からは，自動的条件付き忠実処理 (automatic contingent allegiance processing) はアクセスできない。また，これは 3 ビット・フィールドだが，現在は 2 ビットのみ使用されている。つまり，上の 3 つの定数は，ビット位置ではなく値を表す。

S2DGB$L_32CDBADDR (S2DGB$PQ_64CDBADDR)

このフィールドは，この IO$_DIAGNOSE 操作によってターゲットに送られる，SCSI コマンド・データ・ブロック (CDB) の 32 ビット (または 64 ビット) 仮想アドレスを含みます。

なお， S2DGB$PQ_64CDBADDR はクォドワードへのポインタであるのに対して，S2DGB$L_32CDBADDR はロングワードへのポインタであることに注意してください。

S2DGB$L_32CDBLEN (S2DGB$L_64CDBLEN)

このフィールドは，この IO$_DIAGNOSE 操作によってターゲットに送られる，SCSI コマンド・データ・ブロック (CDB) 内のバイト数を含みます。 (有効値: 2 ～ 248。ただし，ポートによってはより小さな長さに CDB を制限している場合もあります。推奨値: 2 ～ 16。)

S2DGB$L_32DATADDR (S2DGB$PQ_64DATADDR)
このフィールドは，この SCSI 操作で使用される DATAIN または DATAOUT バッファの 32 ビット (または 64 ビット) 仮想アドレスを含みます。ターゲットに送られる CDB が DATAIN または DATAOUT バッファを使用しない場合，このフィールドの値は 0 です。

なお， S2DGB$PQ_64DATADDR はクォドワードへのポインタであるのに対して，S2DGB$L_32DATADDR はロングワードへのポインタであることに注意してください。

S2DGB$L_32DATLEN (S2DGB$L_64DATLEN)
このフィールドは，この操作に関連する DATAIN または DATAOUT バッファ内のバイト数を含みます。ターゲットに送られる CDB が DATAIN または DATAOUT バッファを使用しない場合，このフィールドの値は 0 です。 (有効値: 0 〜 UCB$L_MAXBCNT。推奨値: 0 〜 65,536。ポートはすべて，少なくとも 65,536 バイトのデータ転送をサポートすることが求められます。)

S2DGB$L_32PADCNT (S2DGB$L_64PADCNT)
このフィールドは，この操作で必要なパディング DATAIN または DATAOUT バイト数を含みます。 (有効値: 0 〜このシステム上のディスク・ブロック内の最大バイト数から 1 を引いた値。現在の有効値: 0 〜 511。)

S2DGB$L_32PHSTMO (S2DGB$L_64PHSTMO)
このフィールドは，フェーズ遷移が発生するまで，または期待される割り込みの実行要求までの，ポート・ドライバの待ち時間 (秒数) を含みます。 S2DGB$V_TAGGED_REQ が 1 の場合，またはこのフィールドが 0 または 1 を含む場合，現在設定されているフェーズ遷移の時間切れの値は変更しません。 (有効値: 0 〜 300 {5 分})

S2DGB$L_32DSCTMO (S2DGB$L_64DSCTMO)
このフィールドは，切断していたトランザクションが再び接続するまでの，ポート・ドライバの待ち時間 (秒数) を含みます。 S2DGB$V_TAGGED_REQ が 1 の場合，またはこのフィールドが 0 または 1 を含む場合，現在設定されている切断の時間切れの値は変更しません。 (有効値: 0 〜 65,535 {約 18 時間})

S2DGB$L_32SENSEADDR (S2DGB$PQ_64SENSEADDR)
S2DGB$V_AUTOSENSE が 1 の場合，このフィールドは，この SCSI 操作で使用されるセンス・バッファの 32 ビット (または 64 ビット) 仮想アドレスを含みます。S2DGB$V_AUTOSENSE が 0 の場合，このフィールドは無視されます。

なお， S2DGB$PQ_64SENSEADDR はクォドワードへのポインタであるのに対して， S2DGB$L_32SENSEADDR はロングワードへのポインタであることに注意してください。

S2DGB$L_32SENSELEN (S2DGB$L_64SENSELEN)
S2DGB$V_AUTOSENSE が 1 の場合，このフィールドは，この操作に関連するセンス・バッファ内のバイト数を含みます。 (有効値: 0 ～ 255。注意: 値 0 はクラス・ドライバに対して，受信したセンス・データの廃棄を指定します。推奨値: 18。ポートによってはセンス・バイト数を 18 に制限しています。) S2DGB$V_AUTOSENSE が 0 の場合，このフィールドは無視されます。

7.4.1 64 ビット S2DGB 例

64 ビットの S2DGB の設定例を示します。

```
#include <s2dgbdef.h>                                  /* Define S2DGB   */
#include <far_pointers.h>                              /* Define VOID_PQ */

   S2DGB diag_desc;

   /* Set up some default S2DGB descriptor values */

   diag_desc.s2dgb$l_opcode = OP_XCDB64               /* Use 64-bits  */
   diag_desc.s2dgb$l_flags =  (S2DGB$M_READ |         /* Flags*/
                       S2DGB$M_TAGGED_REQ |
                       S2DGB$M_AUTOSENSE);
   diag_desc.s2dgb$v_tag = S2DGB$K_SIMPLE;           /* SIMPLE que tag */
   diag_desc.s2dgb$pq_64cdbaddr = (VOID_PQ)(&cdb[0]);/* Command addr  */
   diag_desc.s2dgb$l_64cdblen =  6;                  /* Command length */
   diag_desc.s2dgb$pq_64dataddr = (VOID_PQ)(&buf[0]);/* Data addr      */
   diag_desc.s2dgb$l_64datlen = 20;                  /* Data length    */
   diag_desc.s2dgb$l_64padcnt = 0;                   /* Pad length    */
   diag_desc.s2dgb$l_64phstmo = 20;                  /* Phase timeout */
   diag_desc.s2dgb$l_64dsctmo = 10;                  /* Disc timeout  */
   diag_desc.s2dgb$pq_64senseaddr = (VOID_PQ)(&asn[0]);/* Autosense addr */
   diag_desc.s2dgb$l_64senselen = 255;               /* Sense length  */
   diag_desc.s2dgb$l_reserved_1 = 0;                 /* Reserved     */
        .
        .
        .
   status = sys$qiow(0, target_chan, IO$_DIAGNOSE, &iosb, 0, 0,
           &diag_desc,  S2DGB$K_XCDB64_LENGTH,  0,  0,  0,  0);
```

パラメータがすべて有効な場合，クラス・ドライバは必要なポート関数を起動して，CDB の送信とデータの転送に加えて，入力 S2DGB によって定義されるセンス・データの戻し，保存，または廃棄を行います。終了時には，戻り IOSB の形式を 図 7–3 に示します。

図 7–3 **戻り** IOSB **の形式**

Byte count <15:0>		Port VMS status	
SCSI status	Zero	Byte count <31:16>	:04

(First row offset: :00)

ZK−8488A−GE

ほかの QIO 関数を実現する DKDRIVER， GKDRIVER，および MKDRIVER クラス・ドライバは， IO$_DIAGNOSE 要求と共にほかのタグ付き要求を混合します。要求が送られる順序は一般に，ドライバに対して要求が指定される順序に一致します。ただしこの順序には例外があり，ドライバが REQUEST SENSE を受け取り，これに対してオートセンス・データが以前に復元および保存されている場合がこれに相当します。この場合， IO$_DIAGNOSE が直ちに終了し，コマンドがターゲットに送られることはありません。

DKDRIVER， GKDRIVER，および MKDRIVER クラス・ドライバは，次に説明する場合を除き，任意の時点で， 1 つの IO$_DIAGNOSE 操作に限って (開始の入出力ルーチンの中で) アクティブであることを許可します。しかしアプリケーションは，センス・データの存在を正しく検出し，要求される REQUEST SENSE コマンドを送信するために， IO$_DIAGNOSE 要求を単一スレッド化する必要があります。これは， VAX IO$_DIAGNOSE の動作に一致するものです。たとえば， 3 つの読み込みが警告メッセージを伴わずに実行されると，最初の読み込みが CHECK CONDITION を受け取り， 2 番目の読み込みが到着するときにセンス・データがターゲットによって廃棄されます。

DKDRIVER， GKDRIVER，および MKDRIVER ドライバは，すべてのアクティブ操作の S2DGB$V_AUTOSENSE フラグが 1 に等しいときに限って，複数の IO$_DIAGNOSE 操作が (開始の入出力ルーチンの中で) アクティブであることを許可します。 S2DGB$V_AUTOSENSE が 0 に等しい IO$_DIAGNOSE が発生した時点で，クラス・ドライバは上で説明した制限を適用します。

8
OpenVMS Alpha 64 ビット API ガイドライン

本章では， OpenVMS Alpha 64 ビット仮想アドレッシングをサポートする， 64 ビット・インタフェースを開発する際のガイドラインについて説明します。独自の 64 ビット・アプリケーション・プログラミング・インタフェースを開発しているアプリケーション・プログラマにとって，このガイドラインは大変有用です。

本章で推奨するガイドラインは，難しく厳密な規則ではありません。適切なプログラミングの例を紹介しながら，ガイドラインを説明します。

C ポインタ・プラグマについての詳細は，『DEC C User's Guide for OpenVMS Systems』を参照してください。

8.1 クォドワード/ロングワード引数ポインタのガイドライン

OpenVMS Alpha 64 ビット・アドレッシングのサポートは，アプリケーション・プログラムが 64 ビット・アドレス空間のデータへアクセスすることを認めています。そのためアプリケーションの中では， 32 ビット符合拡張値でないポインタ (64 ビット・ポインタ) がより一般的になります。既存の 32 ビット API も引き続きサポートされるため，プログラマは 64 ビット・ポインタの存在を特に注意する必要があります。

たとえば 64 ビット・アドレスを，誤って 32 ビット・アドレスしか処理できないルーチンに渡す可能性が考えられます。また，新しい API が 64 ビット・ポインタをデータ構造に埋め込む場合も考えられます。このようなポインタは，新しいデータ構造の中では符号拡張の 32 ビット値として存在しますが，最初は 32 ビット・アドレス空間のポイントに制限されます。

どのルーチンも， 32 ビット・アドレスに代わって 64 ビット・アドレスが渡される場所では，プログラミング・エラーに注意する必要があります。このようなチェックは**符号拡張チェック**と呼ばれるもので，ビット 31 の値に一致した状態で，アドレスの上位 32 ビットがすべて 0，またはすべて 1 であることを確認します。このチェックは，この制限を適用しているルーチン・インタフェースで実行できます。

ルーチン・インタフェースを新しく定義するときは， 32 ビット・ソース・モジュールから，ルーチンの呼び出しを簡単にプログラムできるように考慮しなければなりません。また， 64 ビット・アドレッシングのサポートをもともと意図している言語だけでなく，すべての OpenVMS プログラミング言語で作成された呼び出しを考慮しなければなりません。新規ルーチンの 32 ビット呼び出し側に対して不慣れなプログラ

ミングを強いることを防ぐために， 64 ビット呼び出し側だけでなく 32 ビット呼び出し側にも配慮しなければなりません。

32 ビット・アドレス空間 (P0/P1/S0/S1) での常駐が禁止されている参照渡しの引数は，その参照アドレスを符号拡張チェックする必要がある

OpenVMS 呼び出し規則では，ルーチンに渡される 32 ビット値は，ルーチンが呼び出される前に， 64 ビットに符号拡張されることが要求されます。このため，呼び出されるルーチンは，常に 64 ビット値を受け取ります。 32 ビット・ルーチンは，引数に対する参照を符号拡張チェックしない限り，その呼び出し側が 32 ビット・アドレスで正しくルーチンを呼び出したかどうかを判断できません。

データがディスクリプタでルーチンに渡される場合には，ディスクリプタへの参照に対してもこの符号拡張チェックが適用されます。

符号拡張チェックが失敗した場合，呼び出されたルーチンは，エラー状態 SS$_ARG_GTR_32_BITS を返します。

あるいは，呼び出されるルーチンが，エラーがない状態で 64 ビットの位置に渡されるデータを受け取ることが求められる場合，符号拡張チェックが失敗したときは，データを 32 ビット・アドレス空間にコピーすることができます。ルーチンがデータをコピーする 32 ビット・アドレス空間は，ローカル・ルーチン記憶域 (つまり，現在のスタック) です。ローカル記憶域以外の 32 ビット位置にデータがコピーされる場合，メモリ・リークおよびリエントラントについて考慮する必要があります。

新規ルーチンを開発する場合，コードへのポインタおよび新規ルーチンへ渡されるすべてのデータ・ポインタは，できるだけ 64 ビット・アドレス空間を利用することが望まれます。これは，データがルーチンの場合，またはプログラマやコンパイラ，リンカが通常は 64 ビット・アドレス空間に配置しない**静的データ**と一般に考えられる場合であっても同様です。コードと静的データが 64 ビット・アドレス空間の中でサポートされる場合，このルーチンを改めて変更する必要はありません。

32 ビット・ディスクリプタ引数が 32 ビット・ディスクリプタであることを確認する必要がある

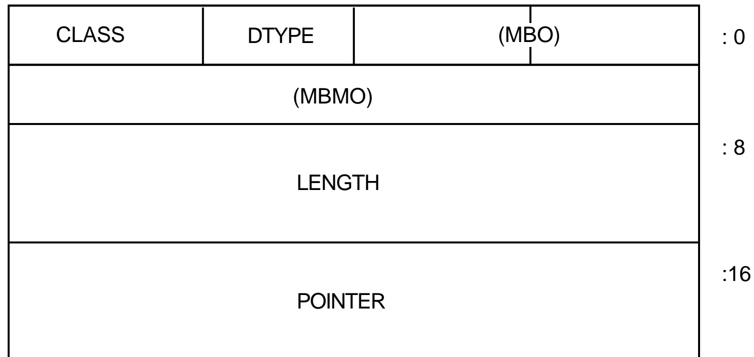
ディスクリプタを受け取るルーチンは， 32 ビットおよび 62 ビット・ディスクリプタ形式を区別するフィールドをチェックしなければなりません。 64 ビット・ディスクリプタを受け取ると，ルーチンはエラーを返します。

既存の大半の 32 ビット・ルーチンは， 64 ビット・ディスクリプタが間違って指定されると，次の理由でエラー状態 SS$_ACCVIO を返します。

- ディスクリプタの 64 ビット形式は， 32 ビット・ディスクリプタの LENGTH が位置しているオフセットで， MBO(0 でなければならない) ワードを含む。

- 図 8–1 に示すように， 32 ビット・ディスクリプタの POINTER が位置しているオフセット 4 で， MBMO(-1 でなければばならない) ロングワードを含む。

図 8–1　32 ビット・ディスクリプタ



64 ビット・ディスクリプタで渡される引数を受け取るルーチンは， 64 ビット・ディスクリプタと同様に 32 ビット・ディスクリプタに対処する必要がある

新規ルーチンは，同じルーチンの中で 32 ビット・ディスクリプタと 64 ビット・ディスクリプタに対処します。同じ引数で 32 ビット・ディスクリプタまたは 64 ビット・ディスクリプタを指すことができます。オフセット 0 での 64 ビット・ディスクリプタ MBO ワードは，値が 1 であることをテストし，オフセット 4 での 64 ビット・ディスクリプタ MBMO ロングワードは -1 であることをテストすることによって， 32 ビット・ディスクリプタと 64 ビット・ディスクリプタを区別します。

32 ビット・ディスクリプタに加えて 64 ビット・ディスクリプタの処理を目的として変換されている，既存の 32 ビット・ルーチンを考えてみます。入力ディスクリプタが 64 ビット・ディスクリプタであると判断されると， 64 ビット・ディスクリプタが指すデータを，まず 32 ビットのメモリ位置にコピーした上で， 32 ビット・メモリの中で 32 ビット・ディスクリプタが作成されます。この新しい 32 ビット・ディスクリプタを既存の 32 ビット・コードに渡すことで，ルーチン内部での変更はこれ以上必要ありません。

32 ビット項目リスト引数が 32 ビット項目リスト引数であることを確認する必要がある

項目リストは， item_list_2 および item_list_3 という 2 種類の形式で定義されます。item_list_2 形式の項目リストは 2 つのロングワードで構成されており，先頭のロングワードには長さと項目コードのフィールドが含まれ， 2 番目のロングワードには通常バッファ・アドレスが含まれています。

これに対して item_list_3 形式の項目リストは 3 つのロングワードで構成されています。先頭のロングワードには長さと項目コードのフィールドが含まれ，2 番目と 3 番目のロングワードには，通常バッファ・アドレスおよび戻り長アドレス・フィールドが含まれています。

32 ビット項目リストが 2 種類存在するため，64 ビット項目リストも 2 種類定義されています。新しく加えられた **item_list_64a** および **item_list_64b** は，32 ビットの項目リストにそれぞれ対応します。どちらの形式の項目リストも，オフセット 0 および 4 の位置に，MBO および MBMO フィールドをそれぞれ含んでいます。また，両方の項目リストとも，ワード・サイズの項目コード・フィールド，クォドワード・サイズの長さフィールド，およびクォドワード・サイズのバッファ・アドレス・フィールドをそれぞれ含んでいます。なお，**item_list_64b** 形式の項目リストは，これとは別にクォドワード・サイズの戻り値長アドレス・フィールドも含んでいます。戻り値長は 64 ビットです。

項目リストを受け取るルーチンは，フィールドをテストして，これが 32 ビット項目リストと 64 ビット項目リストのどちらであるかを区別しなければなりません。64 ビット項目リストを受け取った場合，ルーチンはエラーを返します。

既存の大半の 32 ビット・ルーチンは，64 ビット項目リストが間違って渡されたことを次の理由で判別した場合，エラー状態 SS$_ACCVIO を返します。またはこれをシグナル通知します。

- 64 ビット形式の項目リストはオフセット 0 の位置に MBO (must be one) ワードを含んでいるが，32 ビット項目リストの場合は，ここに LENGTH が位置している。
- 64 ビット項目リストはオフセット 4 の位置に MBMO (must be minus one) ロングワードを含むが，32 ビット項目リストの場合は，ここに BUFFER ADDRESS が位置している。図 8–2 および図 8–3 はこの状態を示している。

図 8–2 item_list_64a

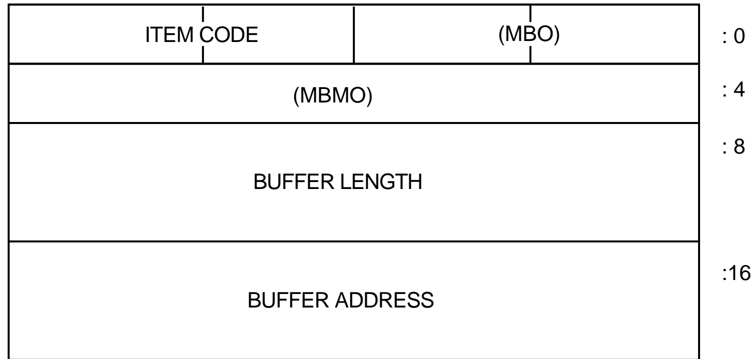
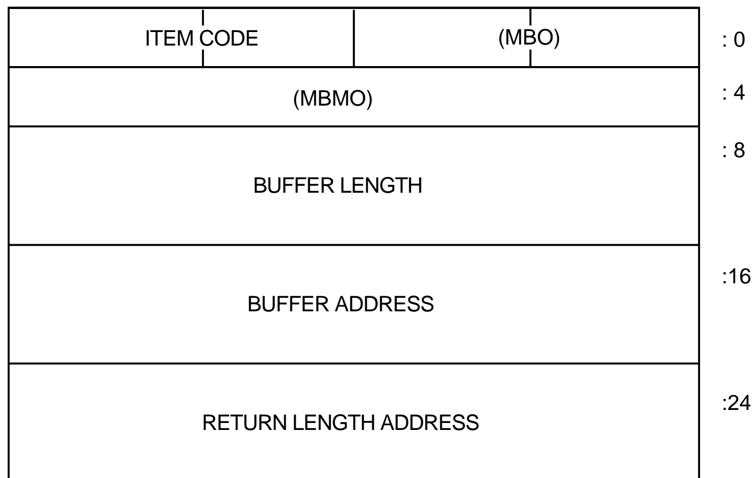


図 8–3 item_list_64b



64 ビット項目リスト引数で渡される引数を受け取るルーチンは，64 ビット項目リスト同様 32 ビット項目リストにも対処する必要がある

新規ルーチンは，同一ルーチン内で 32 ビット項目リストと 64 ビット項目リストに対処しなければなりません。同じ引数が， 32 ビット項目リストと 64 ビット項目リストのどちらも指すことができます。 64 ビット項目リストのオフセット 0 にある MBO ワードは 1 でなければならず， 64 ビット項目リストのオフセット 4 にある MBMO ロングワードは -1 でなければなりません。そのため，これをテストすることによって， 64 ビット項目リストと 32 ビット項目リストを区別します。

ポインタの参照渡しを避ける

特定のメモリ管理ルーチンなどで，ポインタを参照渡しする必要がある場合，ポインタは 64 ビットで定義します。

32 ビット・ポインタと 64 ビット・ポインタが混在すると， 64 ビット・ポインタが期待されている場所に，呼び出し側が間違って 32 ビット・ポインタを参照渡しするなどの，プログラミング・エラーを引き起こします。

プログラマによってロングワードしか割り当てられていないのに，呼び出されたルーチンが 64 ビット・ポインタを読み込むと，ルーチンの中で間違ったアドレスが使用されることになります。

また，呼び出されたルーチンが 64 ビット・ポインタを返すと，プログラマによって割り当てられているロングワードに対して 64 ビット・アドレスが書き込まれるため，データの破損が発生します。

ポインタを参照渡しされる既存のルーチンには， 64 ビット・サポート用の新規インタフェースが必要です。旧ルーチン・インタフェースは， 32 ビット・メモリ位置の中でポインタをこれまでのように渡し，これに対して新規ルーチン・インタフェースは， 64 ビット・メモリ位置でポインタを渡します。同じインタフェースを保持した上で， 64 ビット・ポインタを渡すと，既存のプログラムに異常が発生します。

例： SYS$CRETVA_64 サービスで使用される戻り仮想アドレスは，参照渡しでポインタを渡すことのできる例です。 P0 空間および P1 空間の中で作成される仮想アドレスは， 64 ビットすべてが返されますが，意味があるのは 32 ビットだけであることが保証されています。 SYS$CRETVA_64 は 64 ビット空間にアドレス空間を作成し，64 ビット・アドレスを返すこともできます。 64 ビット・アドレスが返るため，返される値は常に 64 ビットであることが必要です。

メモリ割り当てルーチンは，可能な場合，値 (つまり R0) で割り当てられるデータへのポインタを返します。 C 割り当てルーチン， **malloc**， **calloc**， **realloc** がこの例です。

メモリ管理ルーチンでないルーチンの新規インタフェースで，出力引数を定義してアドレスを受け取ることは避けてください。 64 ビット・サブシステムがメモリを割り当て， 32 ビット呼び出し側に対してポインタを出力引数で戻すと，問題が発生します。呼び出し側は， 64 ビット・ポインタをサポートまたは指定できない可能性があります。あるデータへのポインタを返す代わりに，呼び出し側はバッファへのポインタを用意して，呼び出されたルーチンがこのユーザ・バッファにデータをコピーすることが望まれます。

参照渡しされる 64 ビット・ポインタは，ルーチンへの呼び出しが 64 ビット言語または 32 ビット言語で作成できるような環境で定義する必要があります。 64 ビット・ポインタはすべての呼び出し側によって渡される必要があることを，明確に指示しなければなりません。

特に要求されない限りルーチンは 64 ビット・アドレスを返す必要はない

呼び出し側で 64 ビット・アドレスを処理できることが確実でない限り，メモリを割り当て，その呼び出し側に対してアドレスを返すルーチンは，常に 32 ビット・アドレス・メモリを割り当てることに，十分に注意してください。これは，関数の戻り値と出力パラメータの両方に当てはまります。この規則は， 64 ビット・アドレスを求めていないアプリケーションに， 64 ビット・アドレスが**侵入**することを防ぐものです。この結果，呼び出し可能なライブラリを開発しているプログラマは，特に注意してこの規則に従う必要があります。

既存のルーチンが，割り当てられているメモリのアドレスをルーチン値として返す場合を考えます。呼び出し側が 64 ビットを処理できることがわかっていて，ルーチンが入力パラメータを受け取った場合， 64 ビット・アドレスを返すことに何も問題はありません。これ以外の場合は，引き続き 32 ビット符号拡張アドレスを返さなければ**なりません。**後者の場合， 64 ビットを処理できる呼び出し側が 64 ビット・メモリの割り当てを優先するのであれば，既存のバージョンに代わって，ルーチンの新バージョンを提供できます。

例：文字列ディスクリプタを操作する LIBRTL 内のルーチンは，渡されたディスクリプタが新しい 64 ビット形式であれば，呼び出し側が 64 ビットを処理できると判断します。この場合は，文字列データに 64 ビット・メモリを割り当てても安全です。これ以外の場合は， 32 ビット・アドレス・メモリだけを引き続き使用します。

パブリック・インタフェースのデータ構造では埋め込みポインタを避ける

新規インタフェースの新しい構造で埋め込みポインタが必要な場合， (クォドワード・アラインされている) 64 ビット・ポインタ用の構造の中で記憶域を用意します。呼び出されたルーチンは，場合によって構造からポインタを読み込む必要がありますが，単純に 64 ビット全体を読み込みます。

ポインタが 32 ビット符号拡張アドレスの制約を受けている場合 (たとえば，ポインタが 32 ビット・ルーチンに渡される場合)，ルーチンの入り口で， 64 ビット・ポインタについて符号拡張チェックが実行されます。符号拡張チェックが失敗すると，エラー状態 SS$_ARG_GTR_32_BITS が呼び出し側に返されます。または 64 ビット・アドレス空間に常駐しているデータが検出されると，これが 32 ビット・アドレス空間にコピーされます。

新しい構造は， 64 ビットの呼び出し側も 32 ビットの呼び出し側も，余分なコードを使用しないで済むように定義する必要があります。構造では， 64 ビットの呼び出し側を対象とするクォドワード・フィールドと， 32 ビットの呼び出し側を対象とする 2 つのロングワード・フィールドが互いに重なり合って提供されます。先頭のロングワードは 32 ビット・ポインタ・フィールドで，次のロングワードは MBSE (must be sign-extension) フィールドです。 32 ビットの呼び出し側の大半では， MBSE フィ

ールドは 0 になります。これは，ポインタが 32 ビット・プロセス空間アドレスとなるためです。ここで重要なのは，ポインタを 64 ビット値として定義し， 32 ビットの呼び出し側に対してクォドワード全体を入力する必要があることを明確にすることです。

次の例では， 64 ビットと 32 ビットの両方の呼び出し側が，関数 routine を呼び出すときに，block構造にポインタを渡し，同じ関数プロトタイプを使用します (dataは別のモジュールで定義され，その構造は不定であるものとします)。

```
#pragma required_pointer_size save
#pragma required_pointer_size 32

typedef struct block {
    int blk_l_size;
    int blk_l_flags;
    union  {
#pragma required_pointer_size 64
        struct data *blk_pq_pointer;
#pragma required_pointer_size 32
        struct  {
            struct data *blk_ps_pointer;
            int blk_l_mbse;
            } blk_r_long_struct;
        } blk_r_pointer_union;
    } BLOCK;

#define blk_pq_pointer     blk_r_pointer_union.blk_pq_pointer
#define blk_r_long_struct  blk_r_pointer_union.blk_r_long_struct
#define blk_ps_pointer     blk_r_long_struct.blk_ps_pointer
#define blk_l_mbse         blk_r_long_struct.blk_l_mbse

/* Routine accepts 64-bit pointer to the "block" structure */
#pragma required_pointer_size 64
int routine(struct block*);

#pragma required_pointer_size restore
```

入力引数を指定する既存の 32 ビット・ルーチンの場合，これがポインタを埋め込む構造のときは，別の方法で既存の 32 ビット・インタフェースを保持できます。実行時に， 32 ビット形式のデータ構造と区別できる， 64 ビット形式のデータ構造を開発できます。 32 ビット形式の構造だけを受け取る既存のコードは， 64 ビット形式の構造が指定されると，自動的に異常終了します。

新しい 64 ビット構造についての構造定義には， 32 ビット形式の構造を含む必要があります。 32 ビット形式の構造を含むことによって，呼び出されたルーチンは入力引数を， 64 ビット形式の構造へのポインタとして宣言し，いずれの場合でも明確に処理することができます。

言語に対して，型のチェックを行う 2 種類の関数プロトタイプを用意しなければなりません。省略時の設定時の関数プロトタイプは，引数を 32 ビット形式の構造へのポインタとして指定します。 64 ビット形式の関数プロトタイプは，マニュアルに説明されているように，シンボルを定義することによって選択できます。

64 ビット対 32 ビット・ディスクリプタは，これがどのように行われるかを示す例です。

例:次の例では，シンボルFOODEF64の状態が，正しい関数プロトタイプと共に，64 ビット形式の構造を選択します。シンボルFOODEF64が未定義の場合，古い 32 ビット構造が定義され，古い 32 ビット関数プロトタイプが使用されます。

関数foo_printを実現するソース・モジュールはシンボルFOODEF64を定義し，32 ビットおよび 64 ビットの呼び出し側からの呼び出しを処理することができます。64 ビットの呼び出し側はフィールドfoo64$l_mbm1を -1 に設定します。foo_printはフィールド **foo64$l_mbm1** が -1 であるかどうかをテストすることによって，呼び出し側が 64 ビット形式の構造を使用しているか，または 32 ビット形式の構造を使用しているかを判断します。

```
#pragma required_pointer_size save
#pragma required_pointer_size 32

typedef struct foo {
    short int     foo$w_flags;
    short int     foo$w_type;
    struct data * foo$ps_pointer;
    } FOO;

#ifndef FOODEF64

/* Routine accepts 32-bit pointer to "foo" structure */
int foo_print(struct foo * foo_ptr);

#endif

#ifdef FOODEF64

typedef struct foo64 {
    union  {
        struct  {
            short int     foo64$w_flags;
            short int     foo64$w_type;
            int           foo64$l_mbmo;
#pragma required_pointer_size 64
            struct data * foo64$pq_pointer;
#pragma required_pointer_size 32
            } foo64$r_foo64_struct;
        FOO foo64$r_foo32;
        } foo64$r_foo_union;
    } FOO64;

#define foo64$w_flags    foo64$r_foo_union.foo64$r_foo64_struct.foo64$w_flags
#define foo64$w_type     foo64$r_foo_union.foo64$r_foo64_struct.foo64$w_type
#define foo64$l_mbmo     foo64$r_foo_union.foo64$r_foo64_struct.foo64$l_mbmo
#define foo64$pq_pointer foo64$r_foo_union.foo64$r_foo64_struct.foo64$pq_point
er#define foo64$r_foo32    foo64$r_foo_union.foo64$r_foo32

/* Routine accepts 64-bit pointer to "foo64" structure */
#pragma required_pointer_size 64
int foo_print(struct foo64 * foo64_ptr);
```

```
#endif

#pragma required_pointer_size restore
```

上の例で，構造fooおよびfoo64が同じソース・モジュールの中で交換して使用される場合，シンボル FOODEF64 を削除できます。この場合，ルーチン foo_print は次のように定義されます。

```
int foo_print (void * foo_ptr);
```

FOODEF64 シンボルを削除することによって， 32 ビットの呼び出し側と 64 ビットの呼び出し側が同じ関数プロトタイプを使用できます。ただし C ソースのコンパイルの際に，厳密な型のチェックが行われることはありません。

8.2 Alpha/VAX ガイドライン

アドレス，サイズ，および長さの引数だけが，クォドワードとして値で渡されなければならない

値で渡される引数は， VAX 上ではロングワードに制限されています。 VAX API との互換性を実現するには，クォドワード引数を値渡しではなく参照渡しする必要があります。しかし，アドレス，サイズ，および長さは，アーキテクチャが原因で，理論的には OpenVMS VAX 上でロングワード，かつ OpenVMS Alpha 上ではクォドワードになる引数の例です。

API が OpenVMS VAX 上で利用できない場合であっても，すべての API 上での一貫性を実現するため，このガイドラインに従ってください。

ページ・サイズに依存する単位の使用を避ける

長さやオフセットなどの引数は，バイトなど，ページ・サイズに依存しない単位で指定します。

ページレットは不便な単位です。これは VAX との互換性を目的として開発され，OpenVMS Alpha 上で， OpenVMS VAX 互換インタフェースの中で使用されています。ページレットはそのサイズが VAX ページに等しく，ページ・サイズに依存しない単位とは考えられません。これは Alpha 上の CPU 固有ページとしばしば混乱されるためです。

例: EXPREG_64 内の Length_64 引数は，クォドワード・バイト・カウントとして，値で渡されます。

参照渡しされるすべてのデータをすべて自然にアラインする

呼び出されるルーチンは，コンパイラに対して引数がアラインされていることを指定し，これによってコンパイラは，より効率的なロードおよび保存のシーケンスを実行できます。データがそのままアラインされていない場合，性能は低下します。

参照渡しのデータが自然にアラインされていないために呼び出されたルーチンが正しく実行できない場合，ルーチンは明示的にチェックを実行し，アラインされていない場合はエラーを返す必要があります。たとえば，ロック付きロード，条件付き保存がルーチンの内部でデータについて実行される場合，データがアラインされていないと，ロック付きロード，条件付き保存は正しく処理されません。

8.3 32 ビット API から 64 ビット API への拡張

API の拡張を， 32 ビット設計の改善や新規機能の追加とは区別して考えると，必要な作業を容易に行うことができます。新しい 64 ビット API の中でのルーチンの呼び出しは，簡単なプログラミング作業に過ぎません。

64 ビット・ルーチンは， 64 ビット形式の構造に加えて 32 ビット形式の構造を受け取る

API への呼び出しを簡単に修正できるように，インタフェースは， 64 ビット形式に加えて， 32 ビット形式の構造も受け取ることができなければなりません。

例: 32 ビット API が情報をディスクリプタで渡していた場合，新規インタフェースは同じ情報をディスクリプタで渡さなければなりません。

64 ビット・ルーチンは 32 ビット・ルーチンと同じ機能を提供する

新しい 64 ビット API が古い API の機能スーパセットではない場合でも，現在 32 ビット API を呼び出しているアプリケーションは，古い 32 ビット API への古い呼び出しの一部を保持することなく，完全にアップグレードして 64 ビット API を呼び出すことができなければなりません。

例: SYS$EXPREG_64 は P0， P1，および P2 プロセス空間で動作します。 SYS$EXPREG_64 は$EXPREG の機能スーパセットであるため，呼び出し側はすべての呼び出しを SYS$EXPREG に置換できます。

接尾辞“_64”を適切に使用する

システム・サービスの場合，この接尾辞は 64 ビット・アドレスを参照渡しで受け取るサービスで使用されます。拡張したサービスの場合，これによって 64 ビット機能のバージョンと，対応する 32 ビット機能のバージョンが区別されます。一方，新規ルーチンの場合は，この接尾辞によって， 64 ビット長アドレス・セルが読み込み/書き込みされることが明確に示されます。埋め込み 64 ビット・アドレスを含む構造が

渡されるとき，この構造が 64 ビット構造として自己識別しない場合にも，この接尾辞が使用されます。ルーチンが受け取るのは 64 ビット・ディスクリプタであるという理由で，ルーチン名に "_64"を含める必要はありません。なお，任意の値を参照渡しで渡すときには，接尾辞が必要ないことに注意してください。接尾辞が必要なのは， 64 ビット・アドレスを参照渡しで渡すときです。

この規則は，ほかのルーチンに対しても同じように推奨されます。

例:

SYS$EXPREG_64 (region_id_64, length_64, acmode, return_va_64, return_length_64)
SYS$CMKRNL_64 (routine_64, quad_arglst_64)

8.4 32 ビット・ルーチンおよび 64 ビット・ルーチンの例

64 ビット・アドレッシングのサポートを目的として拡張された 32 ビット・ルーチン・インタフェースの例を次に示します。この例では，ガイドラインで記述された各種の問題を処理しています。

古いシステム・サービス SYS$CRETVA の C 関数宣言は，次の形式で行われます。

```
#pragma required_pointer_size save
#pragma required_pointer_size 32
int sys$cretva (
        struct _va_range * inadr,
        struct _va_range * retadr,
        unsigned int       acmode);
#pragma required_pointer_size restore
```

新しいシステム・サービス SYS$CREATE_VA の C 関数宣言は，次の形式で行われます。

```
#pragma required_pointer_size save
#pragma required_pointer_size 64
int sys$cretva_64 (
        struct _generic_64 * region_id_64,
        void *               start_va_64,
        unsigned __int64     length_64,
        unsigned int         acmode,
        void **              return_va_64,
        unsigned __int64 *   return_length_64);
#pragma required_pointer_size restore
```

SYS$CRETVA_64 の新規ルーチン・インタフェースは，`_va_range`構造の中で埋め込みポインタを訂正し， 64 ビットの`region_id_64`引数を参照で渡し， 64 ビットの`length_64`引数を値で渡します。

9

64 ビット・アドレッシングをサポートする OpenVMS Alpha ツールおよびユーティリティ

本章では， 64 ビット仮想アドレッシングをサポートするように強化された次の OpenVMS Alpha ツールについて，簡単に説明します。

- OpenVMS デバッガ
- システム・コード・デバッガ
- XDELTA
- ウォッチポイント・ユーティリティ
- SDA
- OpenVMS ランタイム・ライブラリの LIB$および CVT$機能

9.1 OpenVMS デバッガ

OpenVMS Alpha システムのデバッガは， 64 ビット・アドレッシングのサポートによって実現される拡張メモリにアクセスすることができます。完全な 64 ビット・アドレス空間の中で，データを検査したり操作することができます。

新しい[Quad]オプションを使用することによって，変数をクォドワードとして扱うことができます。このオプションは，[Monitor]プルダウン・メニューおよび[Examine]ダイアログ・ボックスの，[Typecast]メニューにあります。

デバッガの省略時の設定の型は**ロングワード**です。これは， 32 ビット・アプリケーションをデバッグするのに適しています。 64 ビット・アドレス空間を使用するアプリケーションをデバッグするには，省略時の設定の型を**クォドワード**に変更することをおすすめします。この操作を行うには， SET TYPE QUADWORD コマンドを使用します。

16 進アドレスは， Alpha 上では 16 桁の数値であることに注意してください。この例を次に示します。

```
DBG> EVALUATE/ADDRESS/HEX %hex 000004A0
00000000000004A0
DBG>
```

デバッガは 32 ビット・ポインタおよび 64 ビット・ポインタをサポートします。

OpenVMS デバッガの使用についての詳細は，『OpenVMS デバッガ説明書』を参照してください。

9.2 OpenVMS Alpha システム・コード・デバッガ

OpenVMS Alpha システム・コード・デバッガは 64 ビット・アドレスを受け取り，64 ビット・アドレス全体を使用して情報を検索します。

9.3 Delta/XDelta

XDELTA は， OpenVMS Alpha 上で常に 64 ビット・アドレッシングをサポートしています。クォドワード表示モードは，情報をすべてクォドワードで表示します。また， 64 ビット・アドレス表示モードは，すべてのアドレスを 64 ビットの長さとして受け取り，表示します。

XDELTA は， PFN データベースの内容を表示するコマンド文字列をあらかじめ定義しています。 OpenVMS Alpha Version 7.0 における PFN データベース・レイアウトの変更に伴い，表示用のコマンド文字列およびその形式が合わせて変更されました。

Delta/Xdelta についての詳細は，『OpenVMS Delta/XDelta Debugger Manual』を参照してください。

9.4 OpenVMS ランタイム・ライブラリの LIB$および CVT$機能

OpenVMS RTL ライブラリの LIB$機能および CVT$機能の， 64 ビット・アドレッシング・サポートについての詳細は，『OpenVMS RTL Library (LIB$) Manual』を参照してください。

9.5 ウォッチポイント・ユーティリティ

ウォッチポイント・ユーティリティは， 64 ビット・アドレスにウォッチポイントを設定することにより，共用システム空間内の特定の位置に加えられた修正の履歴を保持するデバッグ・ツールです。これは S0， S1，または S2 空間の，任意のシステム・アドレスをウォッチします。

ウォッチポイント・ユーティリティに対する$QIO インタフェースは， 64 ビット・アドレスをサポートします。 WATCHPOINT コマンド・インタプリタ (WP) は， DCL 構文の標準規則に従うコマンドから， WATCHPOINT ドライバ (WPDRIVER) に対して，$QIO 要求を発行します。

コマンドは WATCHPOINT>プロンプトに対して入力し，ウォッチポイントからの情報の設定，削除，および取得を行います。 WATCHPOINT コマンド・インタプリタ (WP) を起動する前に，または WATCHPOINT ドライバをロードする前に，まず，SYSGEN MAXBUF ダイナミック・パラメータを 64000 に設定しなければなりません。この操作は次の手順で行います。

```
$ RUN SYS$SYSTEM:SYSGEN
SYSGEN> SET MAXBUF 64000
SYSGEN> WRITE ACTIVE
SYSGEN> EXIT
```

WP を起動する前に， SYSMAN で WPDRIVER をインストールしなければなりません。次の手順で行います。

```
$ RUN SYS$SYSTEM:SYSMAN
SYSMAN> IO CONNECT WPA0/DRIVER=SYS$WPDRIVER/NOADAPTER
SYSMAN> EXIT
```

次のコマンドで WP を起動します。

```
$ RUN SYS$SYSTEM:WP
```

WATCHPOINT>プロンプトに対してコマンドを入力し，ウォッチポイントからの情報の設定，削除，および取得を行います。

ターミナルの表示幅を 132 桁に設定すると，ウォッチポイント・ユーティリティへの出力および WP ヘルプ画面を最適に表示することができます。この設定は次の手順で行います。

```
$ SET TERM/WIDTH=132
```

9.6 SDA

OpenVMS Alpha Version 7.0 の場合， SDA は，ユーザが式の中で 64 ビット・アドレスおよび 64 ビット値を指定することを認めています。また，適切な場所で完全な 64 ビット値も表示します。

SDA の 64 ビット・アドレッシング・サポートの使用についての詳細は，『OpenVMS Alpha System Dump Analyzer Utility Manual』を参照してください。

10

言語およびポインタの 64 ビット・アドレッシング・サポート

DEC C および DEC C ランタイム・ライブラリ (RTL) は 64 ビット・アドレッシングを完全にサポートしているため， OpenVMS Alpha の 64 ビット・アプリケーション，ライブラリ，およびシステム・コードをプログラミングするには， C 言語が適しています。 64 ビット・ポインタは既存の C コードにシームレスに組み込むことができます。また，自然な C コーディング・スタイルで， OpenVMS Alpha が提供する 64 ビット・アドレス空間を利用して，新規の 64 ビット・アプリケーションを開発することができます。

すべてが 32 ビット・ポインタ・サイズ (省略時の設定)，すべてが 64 ビット・ポインタ・サイズ，および 32 ビットと 64 ビットのポインタ・サイズが混在する環境をそれぞれサポートすることによって， DEC C で 64 ビットの OpenVMS アプリケーションをプログラミングする上で，柔軟性に加えて互換性が提供されます。

32 ビットと 64 ビットのポインタが混在する環境をサポートする ANSI 準拠の#pragma手法は， Digital UNIX にも共通しています。 64 ビットの C サポート機能には，メモリ割り当てルーチン名のマッピング (_malloc64 および_malloc32 の透過的サポート)，および 32 ビットと 64 ビットのポインタ型についての C の型チェックが含まれます。

『OpenVMS Calling Standard』には，ルーチンの起動およびルーチン間でのデータの引き渡しについて，すべての OpenVMS 言語が使用する手法が説明されています。また，エラーおよび例外処理ルーチンにおける一貫性を保持するメカニズムも定義しています。

『OpenVMS Calling Standard』は，常に 64 ビット単位のパラメータを指定しています。 V7.0 より前のリリースの OpenVMS Alpha では，呼び出されたルーチンは，引数の上位 32 ビットを常に無視していました。 OpenVMS Alpha V7.0 以降，『OpenVMS Calling Standard』は，次の 64 ビット・アドレス・サポートを提供します。

- 呼び出されたルーチンは，完全な 64 ビット・アドレスを使用できる。
- 呼び出し者は， 32 ビット・ポインタと 64 ビット・ポインタのどちらも渡すことができる。
- 参照渡しで渡されるポインタは，元のルーチンの 64 ビット版を新しく必要とすることがしばしばある。

- ディスクリプタや項目リストなどを例として定義する**自己識別型**構造体によって，既存の API が互換性を伴って強化される。

混在するポインタのための OpenVMS Alpha の 64 ビット・アドレッシング・サポートには，次の機能も含まれます。

- OpenVMS Alpha 64 ビット仮想アドレス空間レイアウトが，すべてのプロセスに適用される (64 ビット・プロセスや 32 ビット・プロセスの区別は特にない)。
- 64 ビット・ポインタのアドレッシング・サポートは， 64 ビットの OpenVMS Alpha アドレス空間レイアウト全体を対象とし， P0， P1， P2 アドレス空間と，S0/S1， S2，ページ・テーブル・アドレス空間を含む。
- 32 ビット・ポインタは， P0， P1， S0/S1 アドレス空間のアドレッシングについて互換性を持つ。
- P0， P1， P2 空間アドレスをサポートする新規の 64 ビット・システム・サービスが多数追加された。
- 既存の多くのシステム・サービスが， 64 ビット・アドレッシングをサポートするように強化された。
- 32 ビット・ポインタだけのシステム・サービスに渡されるすべての引数について， 32 ビット符号拡張チェックが行われる。
- 64 ビット・アドレスを処理する C および MACRO-32 マクロが用意されている。

11

DEC C RTL の 64 ビット・アドレッシング・サポート

本章は OpenVMS Alpha バージョン 7.0 以降の DEC C ランタイム・ライブラリで提供される 64 ビット・アドレッシング・サポートについて説明します。

DEC C ランタイム・ライブラリは，64 ビット・ポインタをサポートする際に以下の特徴があります。

- 既存のプログラムのバイナリおよびソースの互換性を保証する。
- 64 ビット・サポートを利用するよう修正されていないアプリケーションには影響がない。
- 64 ビット・メモリを割り当てるメモリ割り当てルーチンが強化された。
- 64 ビット・ポインタに対応するために関数パラメータが拡張された。
- 呼び出し側が使用しているポインタ・サイズを知る必要がある関数について，2 種類の実装を用意する。
- シームレスに正しいインプリメンテーションを呼び出すための新しい情報が，DEC C バージョン 5.2 以降のコンパイラに用意されている。
- 32 ビット形式のポインタ・サイズと 64 ビット形式のポインタ・サイズが混在しているアプリケーションに対して，32 ビット形式または 64 ビット形式の関数を明示的に呼び出すことができる。
- 32 ビット・アプリケーションと 64 ビット・アプリケーションで 1 つの共用イメージを使用できる。

11.1 DEC C ランタイム・ライブラリの使用

OpenVMS Alpha バージョン 7.0 以降の DEC C ランタイム・ライブラリは，64 ビット・ポインタを生成し，受け取ることができます。64 ビット・ポインタを使用する第 2 のインタフェースを必要とする関数は，対応する 32 ビットと同じオブジェクト・ライブラリおよび共用イメージにあります。オブジェクト・ライブラリや共用イメージが新しく作成されることはありません。64 ビット・ポインタを使用することで，リンク・コマンドやリンク・オプション・ファイルに変更を加える必要はありません。

DEC C 64 ビット環境を使用すると，1 つのアプリケーションの中で 32 ビット・アドレスと 64 ビット・アドレスの両方を使用できます。ポインタ・サイズの操作についての詳細は，『DEC C User's Guide for OpenVMS Systems』の /POINTER_

SIZE 修飾子，および#pragma pointer_sizeまたは #pragma required_pointer_sizeプロセッサ・ディレクティブに関する記述を参照してください。

/POINTER_SIZE 修飾子に対して，ユーザは値 32 または 64 を指定します。ここで指定した値は，コンパイル・ユニットの中の省略時の設定のポインタ・サイズとして使用されます。アプリケーション・プログラマは，あるモジュール・セットは 32 ビット・ポインタを使用して，また別のセットは 64 ビット・ポインタを使用してコンパイルします。この 2 つのモジュール・グループがお互いを呼び出す場合，特に注意しなければばなりません。

/POINTER_SIZE 修飾子の使用は， DEC C RTL ヘッダ・ファイルの処理にも影響します。 32 ビット実装および 64 ビット実装を持つ関数の場合，修飾子に指定される実際の値に関わらず，/POINTER SIZE 修飾子によって関数プロトタイプは両方の関数にアクセスできます。さらに，修飾子に指定される値は，コンパイル・ユニットの中で呼び出す省略時の設定の実装を決定します。

#pragma pointer_sizeおよび #pragma required_pointer_sizeプロセッサ・ディレクティブは，コンパイル・ユニットの中で使用されるポインタ・サイズを変更します。32 ビット・ポインタを省略時の設定のポインタとし，モジュール内の特定のポインタを 64 ビット・ポインタとして宣言することができます。 64 ビット・メモリ領域からメモリを取得するために， mallocの_malloc64形式を呼び出す必要があります。

11.2 メモリを指す 64 ビット・ポインタの取得

DEC C RTL には，新しく割り当てられたメモリへのポインタを返す関数が数多くあります。これらの各関数の中で，アプリケーションは指されているメモリを所有し，そのメモリを解放する責任があります。

メモリを割り当てる関数は次に示します。

```
malloc
calloc
realloc
strdup
```

各関数とも 32 ビット実装および 64 ビット実装を持ちます。 /POINTER SIZE 修飾子が使用される場合，次の関数が呼び出される可能性もあります。

```
_malloc32, _malloc64
_calloc32, _calloc64
_realloc32, _realloc64
_strdup32, _strdup64
```

/POINTER_SIZE=32 が指定されると，すべてのmalloc呼び出しの省略時の設定は_malloc32になります。

/POINTER_SIZE=64 が指定されると，すべてのmalloc呼び出しの省略時の設定は_malloc64になります。

アプリケーションが 32 ビットまたは 64 ビットのどちらのメモリ割り当てルーチンを呼び出した場合も， free関数があります。この関数はどちらのサイズのポインタも受け取ります。

メモリ割り当て関数は， 64 ビット・メモリへのポインタを返す唯一の関数です。(FILE， WINDOW， DIR などのように) 呼び出しアプリケーションに返されるすべての DEC C RTL 構造ポインタは，通常は 32 ビット・ポインタです。このため 32 ビットと 64 ビットの両方の呼び出し側とも，アプリケーションの中でこれらの構造ポインタを渡すことができます。

11.3 DEC C ヘッダ・ファイル

DEC C バージョン 5.2 以降で提供するヘッダ・ファイルは， 64 ビット・ポインタをサポートします。ポインタを含む各関数プロトタイプは，受け取るポインタのサイズを示すように構成されています。

32 ビット・ポインタまたは 64 ビット・ポインタのいずれか 1 つを引数として受け取る関数に対して， 32 ビット・ポインタをその引数として渡すことができます。

しかし， 32 ビット・ポインタを受け取る関数への引数として， 64 ビット・ポインタを渡すことはできません。この操作を試みると，コンパイラによって診断が行われ，MAYLOSEDATA メッセージが表示されます。なお， IMPLICITFUNC 診断メッセージが表示された場合，これは，その関数の呼び出しについて，ポインタ・サイズのチェックをこれ以上実行できないことを表します。

特に有効なポインタ・サイズについてのコンパイラの診断メッセージを次に示します。

- %CC-IMPLICITFUNC

  指定された関数の使用に先立ち，関数プロトタイプが見つからないことを表します。コンパイラおよびランタイム・システムは，プロトタイプ定義に基づいて，不適切なポインタ・サイズの使用を検出します。正しいヘッダ・ファイルがインクルードできないと，不適切な結果が生じたり，ポインタの切り捨てが発生する可能性があります。

- %CC-MAYLOSEDATA

  この操作を実行するには切り捨てが必要であることを表します。この操作では，指定されたコンテキストの中で 64 ビット・ポインタをサポートしない関数に対して， 64 ビット・ポインタを渡している可能性があります。または関数の戻り値は 64 ビット・ポインタを返し，ソースはその戻り値を 32 ビット・ポインタに保存しようとしている可能性があります。

- %CC-MAYHIDELOSS

  (有効な場合) このメッセージは，キャスト操作により表示が抑止されている本当の MAYLOSEDATA メッセージを出力することを可能にします。

11.4 影響のある関数

OpenVMS Alpha バージョン 7.0 が提供する DEC C RTL は， 32 ビット・ポインタだけを使用するアプリケーション， 64 ビット・ポインタだけを使用するアプリケーション，または両方を組み合わせて使用するアプリケーションにそれぞれ適応します。 64 ビット・メモリを使用するには，少なくともアプリケーションの再コンパイルと再リンクが必要です。ソース・コードの変更量は，アプリケーション，ほかのランタイム・ライブラリへの呼び出し，使用されているポインタ・サイズの組み合わせによってそれぞれ異なります。

DEC C RTL 関数は，次の 4 種類に分類できます。

- ポインタ・サイズの選択に影響を受けない関数
- どちらのポインタ・サイズも受け取れるよう拡張された関数
- 32 ビット実装と 64 ビット実装を持つ関数
- 32 ビット・ポインタだけを受け取る関数

アプリケーション開発者の立場からすると，上記の最初の 2 つの関数は単一ポインタ・モードと混合ポインタ・モードのいずれの場合も簡単に使用できます。

3 番目の関数は，単一ポインタ・モードで使用している場合はソース・コードを変更する必要はありません。ただし混合ポインタ・モードで使用している場合は，ソース・コードの変更が必要です。

4 番目の関数は， 64 ビット・ポインタを使用している場合は十分な注意が必要です。

11.4.1 ポインタ・サイズの影響がない関数

プロトタイプにポインタに関連するパラメータや戻り値が含まれていない場合，ポインタ・サイズの選択による関数への影響を考慮する必要はありません。算術関数がこれに相当します。

この種類の関数のうち，プロトタイプにポインタを含まない関数は，ポインタ・サイズの選択による影響はありません。たとえば strerror関数のプロトタイプは次のとおりです。

```
char * strerror (int error_number);
```

この関数は文字列へのポインタを返しますが，この文字列は DEC C RTL で割り当てられます。その結果， 32 ビット・アプリケーションと 64 ビット・アプリケーションを両方ともサポートするために，この種類のポインタは， 32 ビット・ポインタに対応することが常に保証されています。

11.4.2 両方のポインタ・サイズを受け取る関数

Alpha アーキテクチャは 64 ビット・ポインタをサポートします。 OpenVMS Alpha 呼び出し規則では，すべての引数が 64 ビット値として渡されることを指定しています。 OpenVMS Alpha Version 7.0 より前のバージョンでは，プロシージャに渡される 32 ビット・アドレスはすべて， 64 ビット・パラメータに符号拡張されていました。呼び出された関数はパラメータを 32 ビット・アドレスとして宣言し，これによってコンパイラは 32 ビット命令 (LDL など) を生成して，これらのパラメータを処理していました。

DEC C RTL 内の関数の多くは拡張され， 64 ビット・アドレス全体を受け取るようになりました。関数strlenを例に考えます。

```
size_t strlen (const char *string);
```

この関数内の唯一のポインタは文字列ポインタです。ユーザが 32 ビット・ポインタを渡すと，関数は符号拡張された 64 ビット・アドレスで動作します。ユーザが 64 ビット・アドレスを渡すと，関数はそのアドレスで直接動作します。

DEC C RTL は引き続き，この種類の関数に対して， 1 つのエントリ・ポイントだけを持ちます。この種類の関数の 4 つのポインタ・サイズ・オプションのいずれを追加する場合も，ソース・コードの変更は必要ありません。 OpenVMS マニュアルでは，これらの関数を 64 ビット・フレンドリと呼びます。

11.4.3 2 種類の実装を持つ関数

多くの理由から，関数は， 32 ビット・ポインタ用と 64 ビット・ポインタ用に 2 種類の実装を持つ必要があります。その理由を次に示します。

- 戻り値のポインタ・サイズは，引数の 1 つのポインタ・サイズと同じです。引数が 32 ビットの場合，戻り値は 32 ビットです。引数が 64 ビットの場合，戻り値は 64 ビットです。
- 引数の 1 つはオブジェクトへのポインタですが，そのオブジェクトのサイズがポインタ・サイズに影響を受けます。指しているバイト数を知るためには，コードが 32 ビット・ポインタ・サイズ・モードと 64 ビット・ポインタ・サイズ・モードのどちらでコンパイルされたのかを知る必要があります。
- 関数は動的に割り当てられたメモリのアドレスを返します。メモリは 32 ビット・ポインタでコンパイルされた場合は 32 ビット空間に割り当てられ， 64 ビット・ポインタでコンパイルされた場合は 64 ビット空間に割り当てられます。

アプリケーション開発者の立場から見ると，これらの各関数にはそれぞれ 3 種類の関数プロトタイプがあります。<string.h>ヘッダ・ファイルは，その戻り値が先頭引数のポインタ・サイズに依存する多くの関数を持ちます。memset関数を例に考えます。ヘッダ・ファイルは，この関数に対して 3 つのエントリ・ポインタを定義します。

```
void * memset   (void *memory_pointer, int character, size_t size);
void *_memset32 (void *memory_pointer, int character, size_t size);
void *_memset64 (void *memory_pointer, int character, size_t size);
```

最初のプロトタイプは，この関数を使用している場合に，アプリケーションが現在呼び出している関数です。コンパイラはmemsetへの呼び出しを， /POINTER_SIZE=32 でコンパイルされているときは_memset32 への呼び出しに，また，/POINTER_SIZE=64 コンパイルされているときは _memset64呼び出しにそれぞれ置換します。

関数の 32 ビット形式または 64 ビット形式を直接呼び出すことにより，この省略時の設定の動作を変更することができます。これにより/POINTER_SIZE 修飾子で指定した省略時の設定のポインタ・サイズに関係なく，ポインタ・サイズが混在しているアプリケーションに対応できます。

/POINTER_SIZE 修飾子を指定せずにコンパイルしている場合， 32 ビット固有のインタフェース関数プロトタイプも， 64 ビット固有のインタフェース関数プロトタイプも定義されないことに注意してください。この場合コンパイラは， 2 種類の実装を持つインタフェースに対して，自動的に 32 ビット・インタフェースを呼び出します。

DEC C RTL での 64 ビット・ポインタ・サイズのサポートの一環として， 2 種類の実装を持つ関数の一覧を表 11–1 に示します。 /POINTER_SIZE 修飾子を指定してコンパイルしている場合，修正されていない関数名を呼び出すと，その修飾子で指定されるポインタ・サイズに対応するインタフェースが呼び出されます。

表 11–1 2 種類の実装を持つ関数

basename	malloc	strpbrk	wcsncat
bsearch	mbsrtowcs	strptime	wcsncpy
calloc	memccpy	strrchr	wcspbrk
catgets	memchr	strsep	wcsrchr
ctermid	memcpy	strstr	wcsrtombs
cuserid	memmove	strtod	wcsstr
dirname	memset	strtok	wcstok
fgetname	mktemp	strtol	wcstol
fgets	mmap	strtoll	wcstoul
fgetws	qsort	strtoq	wcswcs

(次ページに続く)

表 11–1 (続き) 2 種類の実装を持つ関数

fullname	realloc	strtoul	wmemchr
gcvt	rindex	strtoull	wmemcpy
getcap	strcat	strtouq	wmemmove
getcwd	strchr	tgetstr	wmemset
getname	strcpy	tmpnam	
gets	strdup	wcscat	
index	strncat	wcschr	
longname	strncpy	wcscpy	

11.4.4 32 ビット・ポインタに制限されている関数

DEC C RTL には 64 ビット・ポインタをサポートしない関数もあります。 64 ビット・ポインタをサポートしない関数に 64 ビット・ポインタを渡そうとすると，コンパイラは%CC-W-MAYLOSEDATA 警告メッセージを生成します。 /POINTER_SIZE=64 修飾子を指定してコンパイルしたアプリケーションは， 64 ビット・ポインタをサポートしない関数に 64 ビット・ポインタを渡すことがないように，ソース・コードの修正が必要な場合があります。

32 ビット・ポインタしか使用できない関数を表 11–2 に示します。 DEC C RTL は，これらの関数に対して 64 ビット・サポートを提供していません。

表 11–2 32 ビット・ポインタに制限されている関数

atexit	getopt	modf	setstate
execve	iconv	recvmsg	setvbuf
execvp	initstate	sendmsg	
frexp	ioctl	setbuf	

関数呼び出し処理の一部として，ユーザが提供する関数へのコールバックを行う関数の一覧を 表 11–3 に示します。コールバック・プロシージャに 64 ビット・ポインタが渡されることはありません。

表 11–3 32 ビット・ポインタだけを渡すコールバック

from_vms	to_vms
ftw	tputs

11.5 ヘッダ・ファイルの読み込み

この節では，DEC C RTL ヘッダ・ファイルで使用されるポインタ・サイズ操作について説明します次の例を参考にヘッダ・ファイルを読み，必要に応じてヘッダ・ファイルを修正してください。

例

1.
```
:
#if INITIAL__POINTER_SIZE 1
#   if (VMS__VER < 70000000) || !defined __ALPHA 2
#      error " Pointer size usage not permitted before OpenVMS Alpha V7.0"
#   endif
#   pragma __pointer_size __save 3
#   pragma __pointer_size 32 4
#endif
:
:
#if __INITIAL_POINTER__SIZE 5
#   pragma __pointer_size 64
#endif
:
:
#if __INITIAL_POINTER_SIZE 6
#   pragma __pointer_size __restore
#endif
:
```

/POINTER_SIZE 修飾子をサポートするすべての DEC C コンパイラは，あらかじめマクロ__INITIAL_POINTER_SIZEを定義しています。このマクロが定義されていない場合，DEC C RTL ヘッダ・ファイルは，暗黙値 0 を使用するという ANSI 規則を利用します。

/POINTER_SIZE 修飾子が使用される場合，マクロは 32 または 64 として定義されます。この修飾子が使用されない場合は， 0 として定義されます。 1 の文は，"ユーザがコマンド・ラインで/POINTER_SIZE=32 または/POINTER_SIZE=64 を指定した場合 "という意味です。

DEC C バージョン 5.2 以降は，多くの OpenVMS プラットフォームでサポートされます。コンパイルのターゲットが 64 ビット・ポインタをサポートしない場合，2 の行はエラー・メッセージを生成します。

ヘッダ・ファイルは，このヘッダがインクルードされる時点では，作用する実際のポインタ・サイズ・コンテキストを決定することができません。さらに， DEC C コンパイラは，__INITIAL_POINTER_SIZEマクロと，ポインタ・サイズを変更するメカニズムだけを提供し，現在のポインタ・サイズを決定することはしません。

ポインタ・サイズに依存するヘッダ・ファイルはすべて，ポインタ・サイズ・コンテキストの保存3 ， 初期化4 ， 変更5 ， および復元6 を行う必要があります。

2.
```
:
#ifndef __CHAR_PTR32 1
#   define __CHAR_PTR32 1
    typedef char * __char_ptr32;
    typedef const char * __const_char_ptr32;
#endif
:
:
#if __INITIAL_POINTER__SIZE
#   pragma __pointer_size 64
#endif
:
:
#ifndef __CHAR_PTR64 2
#   define __CHAR_PTR64 1
    typedef char *__ char_ptr64;
    typedef const char * __const_char_ptr64;
#endif
:
```

64 ビット・ポインタ・サイズ・コンテキストの中で， 32 ビット・ポインタを参照する必要がある関数プロトタイプがあります。また， 32 ビット・ポインタ・サイズ・コンテキストの中で， 64 ビット・ポインタを参照する必要がある関数プロトタイプもあります。

DEC C コンパイラは，typedefが作成される時点で， typedefで使用されるポインタ・サイズを結合させます。 __char_ptr32のtypedef 宣言1 は 32 ビット・コンテキストで行われます。 __char_ptr64のtypedef宣言2 は 64 ビット・コンテキストで行われます。

3.
```
:
#if INITIAL__POINTER_SIZE
#   if (__VMS_VER < 70000000) || !defined __ALPHA
#      error " Pointer size usage not permitted before OpenVMS Alpha V7.0"
#   endif
#   pragma __pointer_size save
#   pragma __pointer_size 32
#endif
:
1
:
#if __INITIAL_POINTER_SIZE 2
#   pragma __pointer_size 64
#endif
:
3
:
int abs (int __j); 4
:
__char_ptr32 strerror (int __errnum); 5
:
```

64 ビット・ポインタをサポートする関数プロトタイプの宣言の前に，ポインタ・コンテキストが 32 ビット・ポインタから 64 ビット・ポインタに変更されています**2** 。

32 ビット・ポインタに制限されている関数は，ヘッダ・ファイルの 32 ビット・ポインタ・コンテキスト・セクション **1** に配置されています。ほかの関数はすべて，ヘッダ・ファイルの 64 ビット・ポインタ・コンテキスト・セクション **3** に配置されています。

ポインタ・サイズの影響を受けない関数 (**4** および**5**) は， 64 ビット・セクションに配置されています。これらの関数は 32 ビット・アドレスの戻り値を除き，ポインタ・サイズの影響は受けません**5** 。またこれらの関数は 64 ビット・セクションにあり，前述の 32 ビット固有のtypedefを使用します。

4.
```
:
#if __INITIAL_POINTER_SIZE
#   pragma __pointer_size 64
#endif
:
:
#if __INITIAL_POINTER_SIZE == 32 1
#   pragma __pointer_size 32
#endif
:
char *strcat (char *__s1, __const_char_ptr64 __s2); 2
:
#if __INITIAL_POINTER_SIZE
#   pragma __pointer_size 32
    :
    char *_strcat32  (char *__s1, __const_char_ptr64 __s2); 3
    :
#   pragma __pointer_size 64
    :
    char *_strcat64  (char *__s1, const char *__s2); 4
    :
#endif
:
```

この例は， 32 ビット実装と 64 ビット実装の両方を持つ関数に対する宣言です。これらの宣言は，ヘッダ・ファイルの 64 ビット・セクションに配置されています。

関数の通常の宣言**2** は， /POINTER_SIZE 修飾子で指定したポインタ・サイズを使用して行われます。ヘッダ・ファイルが 64 ビット・ポインタ・コンテキストにあり， **1** に文を指定しているため， **2** の宣言は/POINTER_SIZE 修飾子と同じポインタ・サイズ・コンテキストを使用して行われます。

32 ビット固有のインタフェース**3** および 64 ビット固有のインタフェース**4** は, 32 ビット・ポインタ・サイズ・コンテキストおよび 64 ビット・ポインタ・サイズ・コンテキストの中でそれぞれ宣言されています。

12
MACRO–32 プログラミングの 64 ビット・アドレッシング・サポート

本章では，MACRO–32 コンパイラおよび関連コンポーネントによる 64 ビット・アドレッシング・サポートについて説明します。主に引数の受け渡しとアドレス計算に対して変更が行われています。

12.1 64 ビット・アドレッシングのガイドライン

OpenVMS Alpha 用にコンパイルされるVAX MACROコード内での，64 ビット・アドレッシングの使用に関するガイドラインを次に示します。

- OpenVMS Alpha に移植されたコードに限って使用する。

  OpenVMS Alpha 上で新しく開発を行う場合は，高級言語の使用をおすすめします。

- コード内で 64 ビット・アドレッシングを明確に指定する。

  64 ビット・アドレッシングの修飾子，マクロ，ディレクティブ，ビルトインを使用すると，信頼性が高く管理しやすいコードを生成することができます。

12.2 64 ビット・アドレッシングのための新規コンポーネントおよび変更されたコンポーネント

MACRO-32 プログラミングでの 64 ビット・アドレッシング・サポートを提供する，新規コンポーネントおよび変更されたコンポーネントの一覧を 表 12–1 に示します。

表 12–1 64 ビット・アドレッシングのための新規コンポーネントおよび変更されたコンポーネント

コンポーネント	説明
$SETUP_CALL64	呼び出しシーケンスを初期化する新規マクロ
$PUSH_ARG64	引数プッシュに相当する処理を行う新規マクロ
$CALL64	ターゲット・ルーチンを起動する新規マクロ
$IS_32BITS	64 ビット値の下位 32 ビットの符合拡張をチェックする新規マクロ

(次ページに続く)

表 12–1 (続き) 64 ビット・アドレッシングのための新規コンポーネントおよび変更されたコンポーネント

コンポーネント	説明
$IS_DESC64	ディスクリプタが 64 ビット形式ディスクリプタか決定する新規マクロ
QUAD=NO/YES	64 ビット仮想アドレスをサポートするためのページ・マクロの新規パラメータ
/ENABLE=QUADWORD	QUADWORD パラメータは拡張され， 64 ビット・アドレス計算を含む
.CALL_ENTRY QUAD_ARGS=TRUE | FALSE	QUAD_ARGS=TRUE | FALSE は引数リストへのクォドワード参照を指定する新規パラメータ
.ENABLE QUADWORD /.DISABLE QUADWORD	QUADWORD パラメータは拡張され， 64 ビット・アドレス計算を含む
EVAX_SEXTL	64 ビット値の下位 32 ビットを，ターゲットに対して符合拡張する新規ビルトイン
EVAX_CALLG_64	可変サイズの引数リストで 64 ビット呼び出しをサポートするため新規ビルトイン
$RAB64 および$RAB64_STORE	64 ビット・アドレス空間でバッガを使用するための新規 RMS マクロ

12.3 64 ビット値の引き渡し

64 ビット値を渡す方法は，引数リストのサイズが固定，または可変であるかどうかによって異なります。次の節では，これらの方法について説明します。

12.3.1 固定長サイズ引数リストでの呼び出し

固定サイズ・リストでの呼び出しの場合， 表 12–2 の手順に従って新規マクロを使用します。

表 12–2 固定サイズ引数リストでの 64 ビット値の引き渡し

手順	使用するマクロ
1. 呼び出しシーケンスを初期化する	$SETUP_CALL64
2. 呼び出し引数を“プッシュ”する	$PUSH_ARG64
3. ターゲット・ルーチンを起動する	$CALL64

これらのマクロの使用例を次に示します。なお， 32 ビットの PUSHL 命令が使用される場合と同じように，逆の順序で引数がプッシュされることに注意してください。

```
MOVL           8(AP), R5        ; fetch a longword to be passed
$SETUP_CALL64  3                ; Specify three arguments in call
$PUSH_ARG64    8(R0)            ; Push argument #3
$PUSH_ARG64    R5               ; Push argument #2
$PUSH_ARG64    #8               ; Push argument #1
$CALL64        some_routine     ; Call the routine
```

$SETUP_CALL64 マクロは，64 ビット呼び出しの状態を初期化します。このマクロは，$PUSH_ARG64 または$CALL64 を使用する前に必要です。引数の個数が 6 個を越える場合，このマクロはローカル JSB ルーチンを作成し，これが起動されて呼び出しを実行します。引数の個数が 6 個以下の場合，引数のロードと呼び出しはインラインで行われ，きわめて効率的です。なお，$SETUP_CALL64 に指定される引数の個数には，#が含まれないことに注意してください。(これは，標準呼び出しシーケンスが，オクタワード・アライン・スタックの先頭のスタック引数の，オクタワード・アラインを求めるためです。JSB ルーチンは，このアライン操作を行います。)

インライン・オプションを使用すると，ローカル JSB ルーチンがなくても，6 個を越える引数を指定して呼び出しを強制的に実行できます。ただし，その使用には制限があります (詳細は 付録 B を参照してください)。

$PUSH_ARG64 マクロは，引数を正しい引数レジスタやスタック位置に直接移動します。これは実際にはスタック・プッシュではありませんが，32 ビット呼び出しの中で使用される PUSHL 命令に似ています。

$CALL64 マクロは引数カウント・レジスタを設定し，ターゲット・ルーチンを起動します。JSB ルーチンが作成された場合は，そのルーチンを終了させます。プッシュされた引数の個数が，$SETUP_CALL64 で指定された個数に一致しない場合，エラーが報告されます。$CALL64 および$PUSH_ARG64 を使用する前に，$SETUP_CALL64 が起動されていることを確認してください。

12.3.1.1 $SETUP_CALL64，$PUSH_ARG64，および$CALL64 の使用上の注意

$SETUP_CALL64，$PUSH_ARG64，$CALL64 を使用するときは，次の点に注意してください。

- 引数はアラインされたクォドワードとして読み込まれる。メモリからロングワードを渡す場合は，これをまずレジスタに移動し，第 12.3.1 項の例に示すように，そのレジスタを$PUSH_ARG64 で使用する。同様に，渡そうとしているクォドワードがアラインされていない場合，値をまずレジスタに移動する。また，(R4)[R0]などのインデックス・オペランドを $PUSH_ARG64 で使用する場合は，クォドワード・インデックスを使用して評価されるので注意する。
- 引数の個数が 6 個を越える場合，ローカル JSB ルーチンが作成され，$SETUP_CALL64 と$CALL64 ルーチンとの間で SP 参照や AP 参照は認められない。$PUSH_ARG64 マクロと$CALL64 マクロは，オペランド内でのこれらのレジスタの使用を報告するが，この範囲内のほかの命令で使用することはできず，フラグを設定することもできない。このような中で AP ベースまたは SP ベー

スの引数を渡す必要がある場合は， $SETUP_CALL64 を起動する前にこれをレジスタに移動する。

- 引数の個数が 6 個を越える場合，$SETUP_CALL64 の起動後も残っている，R15 より上のレジスタ内の値を使用してはならない。この代わりに，一時的なレジスタとして非スクラッチ・レジスタを使用する。たとえば，スタック位置から値を渡す場合を考える。呼び出しの引数の個数が 6 個を越える場合，値をレジスタに移動しなければならない。 R28 などのスクラッチ・レジスタを使用する代わりに， R0 などの VAX レジスタを使用する。 VAX レジスタがすべて使用されている場合は， R13， R14，または R15 を使用する。
- $SETUP_CALL64 と$CALL64 との間の範囲では， R16 より上のスクラッチ・レジスタを使用しても安全である。ただし，すでにロードされている引数レジスタを使用しないように注意しなければならない。引数レジスタは， R21 から R16 に向かって，降順にロードされる。そこで，呼び出しが 6 個の引数を渡す場合を考える。 R21 がロードされているため，最初の$PUSH_ARG64 の後で R21 を使用することは安全でない。$PUSH_ARG64 マクロは，オペランドが，すでにロードされている引数を参照していないかどうかをチェックする。参照しているオペランドがある場合は，コンパイラから警告メッセージが表示される。一時的なレジスタが要求される場合，レジスタ R22 〜 R28 を使用するのが最も安全である。

注意

$SETUP_CALL64，$PUSH_ARG64，および$CALL64 マクロは，インライン・シーケンスで使用することを目的としています。つまり，$SETUP_CALL64/$PUSH_ARG64/$CALL64 シーケンスの中央に分岐したり，$PUSH_ARG64 マクロの回りで分岐することはできず，シーケンスを出て$CALL64 を避けることもできません。

$SETUP_CALL64，$PUSH_ARG64，および$CALL64 についての詳細は， 付録 B を参照してください。

12.3.2 可変サイズ引数リストでの呼び出し

可変サイズ引数リストでの呼び出しの場合，次の手順にしたがって新しい EVAX_CALLG_64 ビルトインを使用します。

1. メモリ内引数リストを作成する。
2. メモリ内引数リストを渡し，ルーチンを呼び出す。次の例を参照。

```
EVAX_CALLG_64 (Rn), routine
```

EVAX_CALLG_64 ビルトイン中の引数リストは，クォドワード引数の個数を先頭に，一連のクォドワードとして読み込まれます。

12.4 64 ビット引数の宣言

.CALL_ENTRY の新パラメータ， QUAD_ARGS=TRUE を使用すると，ルーチンの引数リスト内でのクォドワード引数の使用を宣言できます。 QUAD_ARGS パラメータが指定されている場合，引数へのクォドワード参照が発生すると，コンパイラは通常とは異なる動作をします。まず，このような参照が通常要求する，引数リスト・ホーミングが強制されません。 (クォドワード値を含んでいる引数はホーミングできません。これは，定義により，ホーミングが引数をロングワード・スロットにパックするためです。) 第 2 に，**非アライン・メモリ参照**が，引数リストへのクォドワード参照で報告されません。

引数リスト参照用に生成された実際のコードは， QUAD_ARGS 句の指定によって変更を受けることはありません。ただし， MOVQ など VAX クォドワード命令の中で参照が行われる場合は除きます。多くの場合， QUAD_ARGS は，クォドワード参照に基づいて引数リストのホーミングを禁止し，不要なアライメント・メッセージの表示を抑制します。この表示の抑制は， EVAX_ビルトインと， MOVQ などの VAX クォドワード命令の両方に適用されます。

VAX クォドワード命令の場合， QUAD_ARGS 句は， EVAX_ビルトインの場合と同じように，実際のクォドワードとして，クォドワード引数の読み込みをコンパイラに対して指定します。次の例を考えます。

```
MOVQ    4(AP), 8(R2)
```

QUAD_ARGS 句が指定されている場合， MOVQ は，引数 1 の 64 ビット全体を，8(R2) のクォドワードに格納します。 QUAD_ARGS 句が指定されていない場合，MOVQ は，引数 1 および 2 の下位ロングワードを， 8(R2) のクォドワードに格納します。

QUAD_ARGS も AP に基づいた defferd モード・オペランドのために生成されたコードに影響を与えます。メモリ中の引数から有効なアドレスをロードしなければならない場合に， QUAD_ARGS が有効であれば，これはロングワードではなくクォドワードとして読み込まれます。

12.4.1 QUAD_ARGS の使用上の注意

QUAD_ARGS を使用する場合は，次の点に注意してください。

- AP ベースのクォドワード配列リスト参照は，これらが重なり合うため，奇妙な状態となる。この状態は，たとえば FIRST_ARG， SECOND_ARG，などのように，引数リストのオフセットのシンボル名を定義することによって改善できる。ユーザは，引数の内容を表す意味のあるシンボル名を定義し，ソース・コードを理解しやすいものにすることが望まれる。また，これまでのように直接引数レジスタ参照を使用して，先頭の 6 個の引数を参照することもできる。どちらの場合

も， QUAD_ARGS を宣言して，引数リストがホーミングされないことを確実にすることが望ましい。

- コードを共用するルーチンは， QUAD_ARGS の設定と同じ設定値を持たなければならない。設定値が異なる場合，コンパイラから警告メッセージが表示される。
- 呼び出し側が QUAD_ARGS を持つ場合， JSB ルーチンは呼び出し側の引数リストを参照できない。 JSB ルーチン内で AP を参照するには，最後の CALL_ENTRY がその引数リストをホーミングしなければならない。 HOME_ARGS と QUAD_ARGS は相互に排他的である。
- QUAD_ARGS は，コンパイラがデバッグ・シンボル・テーブルに配置する$ARGn シンボルを，ロングワードではなくクォドワードとして定義する。これらのシンボルを使用することで，受け取った引数値に簡単にアクセスできる。また，シンボリック・デバッガでデバッグしているときに，レジスタ番号やスタック・オフセットの代わりに使用できる。

12.5 64 ビット・アドレス計算の指定

MACRO-32 に明示的なポインタの型宣言はありません。 64 ビット・ポインタ値は，各種の方法でレジスタに作成できます。もっとも一般的な方法は，メモリに保存されているアドレスを EVAX_LDQ ビルトインでロードし，指定されたオペランドのアドレスを MOVA*x*で取得する方法です。

レジスタに 64 ビット・ポインタ値を作成したら，通常の命令で 64 ビット・アドレスにアクセスします。そのアドレスから読み込まれるデータの量は，使用される命令によってそれぞれ異なります。次の例を考えます。

```
MOVL    4(R1), R0
```

R1 が含むポインタが 32 ビットの場合も 64 ビットの場合も， MOVL 命令は R1 からのオフセットが 4 の位置でロングワードを読み込みます。

しかし，特定のアドレッシング・モードでは，算術命令を生成して，有効なアドレスを計算することを要求します。 VAX との互換性を目的として，コンパイラはこれらをロングワード操作として計算します。たとえば，`4 + <1@33>`の結果は，シフトされた値が 32 ビットを越えるため，値 4 となります。クォドワード・モードが有効の場合，上位ビットが失われることはありません。

以前のバージョンの OpenVMS Alpha と共に出荷されていたコンパイラの場合，/ENABLE=QUADWORD 修飾子 (および対応する.ENABLE QUADWORD ディレクティブと.DISABLE QUADWORD ディレクティブ) は，定数式の評価が行われるモードに影響するだけでした。 OpenVMS Alpha Version 7.0 の場合，これらは拡張され，アドレス計算に影響します。この結果， SxADDQ および ADDQ などのクォドワード命令でアドレスが計算されます。

モジュール全体でクォドワード操作を使用するには，コマンド行で /ENABLE=QUADWORD を指定します。特定のセクションにだけクォドワード操作を適用するには，該当するセクションを.ENABLE QUADWORD および .DISABLE QUADWORD ディレクティブで囲みます。

/ENABLE=QUADWORD を使用しても，性能が低下することはありません。

12.5.1 ロングワード操作の折り返し動作への影響

コンパイラは，クォドワード計算を使用してすべてのアドレッシング計算を行うことはできません。これは，既存のコードは 32 ビット操作の折り返し動作に依存するためです。つまり，コードは，実際には 32 ビットをオーバフローするアドレッシング操作を実行する場合もあり，このとき上位ビットは廃棄されます。クォドワード・モードでの計算は，互換性がありません。

/ENABLE を使用してモジュール全体に対してクォドワード評価を設定する前に，既存のコードをチェックして，ロングワードの折り返しへの影響を確認してください。これを簡単に行うことはできませんが，プログラミング手法としてこれはきわめて珍しいもので，コードの外側で呼び出されている可能性もあります。

次の例は折り返しの問題を示しています。

```
MOVAL   (R1)[R0], R2
```

R1 は値 7FFFFFFF を含み， R0 は 1 を含むとします。 MOVAL 命令は S4ADDL 命令を生成します。シフトと追加によって 32 ビットを越えますが，保存される結果は符号拡張された下位の 32 ビットです。

クォドワード計算が使用されると (S4ADDQ)，次の例に示すように，本当のクォドワード値が算出されます。

```
S4ADDL  R0, R1, R2   =>  FFFFFFFF 80000003
S4ADDQ  R0, R1, R2   =>  00000000 80000003
```

折り返しの問題はインデックス・モード・アドレッシングに限りません。次の例を考えます。

```
MOVAB   offset(R1), R0
```

シンボル・オフセットがコンパイル時定数でない場合，この命令によって，値はリンケージ・セクションから読み込まれ， (ADDL 命令を使用して) R1 内の値に追加されます。これを ADDQ に変更すると，値が 32 ビットを越える場合に結果が変化する可能性があります。

12.6 符合拡張およびチェック

新しいビルトインの EVAX_SEXTL (符号拡張ロングワード) は， 64 ビット値の下位 32 ビットをデスティネーションに対して符号拡張します。このビルトインは，デスティネーションに対して，ソースの下位ロングワードの符号拡張を明示的に行うものです。

EVAX_SEXTL は 64 ビット値の下位 32 ビットを取り，上位 32 ビットを符号拡張 (値のビット 31 の内容) で埋め， 64 ビットの結果をデスティネーションに書き込みます。

次の例は，すべて正しい使用例です。

```
evax_sextl r1,r2
evax_sextl r1,(r2)
evax_sextl (r2), (r3)[r4]
```

これらの例に示すように，オペランドは必ずしもレジスタでなくても構いません。

新規マクロ$IS_32BITS を使用すると， 64 ビット値の下位 32 ビットの符合拡張をチェックすることができます。詳細は付録 B を参照してください。

12.7 Alpha 命令ビルトイン

コンパイラは，多くの Alpha 命令をビルトインとしてサポートします。 (OpenVMS Alpha と共に最初に出荷されたコンパイラ以来利用できる) これらのビルトインの多くは， 64 ビットを操作できます。各ビルトインの関数およびその有効なオペランドについては，『Porting VAX MACRO Code to OpenVMS Alpha』を参照してください。また，各 Alpha 命令についての詳細は，『MACRO–64 Assembler for OpenVMS AXP Systems Reference Manual』を参照してください。

12.8 ページ・サイズ依存値の計算

64 ビット仮想アドレスをサポートする新しい修飾子 QUAD=NO/YES は，次に示す各ページ・マクロで使用できます。

- $BYTES_TO_PAGES
- $NEXT_PAGE
- $PAGES_TO_BYTES
- $PREVIOUS_PAGE
- $START_OF_PAGE

これらのマクロは，標準の，アーキテクチャに依存しない手段を提供して，ページ・サイズ依存値を計算します。これらのマクロについての詳細は，『Porting VAX MACRO Code to OpenVMS Alpha』を参照してください。

12.9　64 ビット・アドレス空間でのバッガの作成および使用

64 ビット・アドレス空間でのバッファの作成および使用を目的として，$RAB および$RAB_STORE 制御ブロック・マクロが拡張されました。 64 ビット・バージョンは，それぞれ$RAB64 および$RAB64_STORE です。現時点では，残りの RMS インタフェースは 32 ビットに制限されています。$RAB64 および$RAB64_STORE についての詳細は， 第 5 章を参照してください。

12.10　64K バイトを越える転送のコーディング

MOVC3 および MOVC5 MACRO-32 命令は， 64 ビット・アドレスを正しく処理しますが，転送は 64K バイト長に限られます。この制限が適用されるのは， MOVC3 と MOVC5 がワード・サイズ長を受け取ることが原因です。

64K バイトを越える転送については， OTS$MOVE3 および OTS$MOVE5 を使用します。 OTS$MOVE3 および OTS$MOVE5 は，ロングワード・サイズ長を受け取ります (LIB$MOVC3 および LIB$MOVC5 は， MOVC3 および MOVC5 と同じ 64K バイト長の制限を受けます)。次の例では， MOVC3 を OTS$MOVE3 で置換します。

MOVC3 を使用するコード

```
MOVC3      BUF$W_LENGTH(R5), (R6), OUTPUT(R7)   ; Old code, word length
```

ロングワード長を伴う同じ内容の 64 ビット・コード

```
$SETUP_CALL64  3              ; Specify three arguments in call
EVAX_ADDQ      R7, #OUTPUT, R7
$PUSH_ARG64    R7             ; Push destination, arg #3
$PUSH_ARG64    R6             ; Push source, arg #2
MOVL           BUF$L_LENGTH(R5), R16
$PUSH_ARG64    R16            ; Push length, arg #1
$CALL64        OTS$MOVE3

MOVL           BUF$L_LENGTH(R5), R16
EVAX_ADDQ      R6, R16, R1    ; MOVC3 returns address past source
EVAX_ADDQ      R7, R16, R3    ; MOVC3 returns address past destination
```

MOVC3 は R0， R2， R4 および R5 をクリアするので，その内容がもはや必要ないことを確認してください。

OTS$MOVE3 および OTS$MOVE5 は，『OpenVMS RTL General Purpose (OTS$) Manual』の中でほかの LIBOTS ルーチンと共に説明されています。

12.11 MACRO–32 コンパイラの使用

OpenVMS Alpha 64 ビット・アドレッシング機能を使用するには，OpenVMS Alpha バージョン 7.0 に含まれている MACRO–32 コンパイラを使用しなければなりません。

最新バージョンのコンパイラを使用する場合，64 ビット・アドレッシング機能を使用しているかどうかに関わらず，最新バージョンの ALPHA$LIBRARY:STARLET.MLB も使用しなければなりません。システム上に最新バージョンがインストールされ，論理名が正しいディレクトリを示していることを確認してください。

A

64 ビット・アドレッシングのための C マクロ

ここでは **64** ビット・アドレスの操作，**64** ビット値の下位 **32** ビットの符号拡張のチェック，および **64** ビット形式のディスクリプタをチェックする **C** マクロについて説明します。

- $DESCRIPTOR64
- $is_desc64
- $is_32bits

DESCRIPTOR64

64 ビット文字列ディスクリプタを構成します。

フォーマット

$DESCRIPTOR64 *name, string*

説明

name：変数の名前。
string：文字列のアドレス。

例:

```
int status;
$DESCRIPTOR64 (gblsec, "GBLSEC_NAME");

...

/* Create global page file section */
status = sys$create_gpfile (&gblsec, 0, 0, section_size, 0, 0);

...
```

このマクロは， **SYS$LIBRARY:DECC$RTLDEF.TLB** 内の **descrip.h** にあります。

$is_desc64

64 ビット・ディスクリプタを識別します。

フォーマット

$is_desc64 *desc*

説明

desc： 32 ビットまたは 64 ビット・ディスクリプタのアドレス。

戻り値:

ディスクリプタが 32 ビット・ディスクリプタの場合は 0。
ディスクリプタが 64 ビット・ディスクリプタの場合は 1。

例:

```
#include <descrip.h>
#include <far_pointers.h>
...
        if ($is_desc64 (user_desc))
        {
                /* Get 64-bit address and 64-bit length from descriptor */
                ...
        }
        else
        {
               /* Get 32-bit address and 16-bit length from descriptor */
               ...
        }
```

このマクロは， SYS$LIBRARY:DECC$RTLDEF.TLB 内の descrip.h にあります。

$is_32bits

クォドワードが 32 ビット符号拡張されているかどうかをチェックします。

フォーマット

$is_32bits *arg*

説明

入力: arg **64** ビット値。

出力:

arg が **32** ビット符号拡張されている場合は **1**。
arg が **32** ビット符号拡張されていない場合は **0**。

例:

```
#include <starlet_bigpage.h>
...
if ($is_32bits(user_va))
        counter_32++;   /* Count number of 32-bit references */
else
        counter_64++;  /* Count number of 64-bit references */
```

このマクロは, **SYS$LIBRARY:SYS$STARLET_C.TLB** 内の **starlet_bigpage.h** にあります。

B

64 ビット・アドレッシングのための MACRO-32 マクロ

ここでは 64 ビット・アドレスの操作，64 ビット値の下位 32 ビットの符合拡張のチェック，および 64 ビット形式のディスクリプタをチェックする MACRO-32 マクロについて説明します。

これらのマクロはディレクトリ ALPHA$LIBRARY:STARLET.MLB (たいていは SYS$LIBRARY:STARLET.MLB と同じ) にあり，アプリケーション・コードとシステム・コードの両方で使用できます。ページ・マクロも 64 ビット・アドレス用に強化されています。このサポートは新しいパラメータ QUAD=NO/YES によって提供されます。

この付録で説明されているマクロに対して特定の引数を使用することで，レジスタ・セットを指定できることに注意してください。レジスタ・セットを表すには，レジスタをカンマで区切ってリストし，これを不等号で囲みます。次に例を示します。

```
<R1,R2,R3>
```

レジスタ・セット内のレジスタが 1 つだけの場合は，これを囲む不等号を省略します。

B.1 64 ビット・アドレスを操作するマクロ

この節では，64 ビット・アドレスの操作を目的として設計されている次のマクロについて説明します。

- $SETUP_CALL64
- $PUSH_ARG64
- $CALL64

$SETUP_CALL64

呼び出しシーケンスを初期化します。

フォーマット

$SETUP_CALL64　*arg_count, inline=true* **または** *false*

パラメータ

arg_count
呼び出し内の引数の個数。

inline
TRUE が設定されている場合， JSB ルーチンを作成せずインライン展開を実行します。引数が 6 個以下の場合，省略時の設定は INLINE=FALSE です。

説明

このマクロは， 64 ビット CALL の状態を初期化します。$PUSH_ARG64 および$CALL64 を使用する前に，これを使用しなければなりません。

引数の個数が 6 個以下の場合，コードは常にインラインです。

省略時の設定では，引数の個数が 6 個を越える場合， JSB ルーチンが作成され，このルーチンを起動して実際の呼び出しを実行します。しかし， INLINE=TRUE でインライン・オプションが指定されると，コードはインラインで生成されます。このオプションは，オプションの対象となるコードのスタックの深さが固定の場合に限って有効です。 RUNTIMSTK または VARSIZSTK メッセージが表示されない場合，スタックの深さが固定であると仮定できます。それ以外の場合，スタック・アライメントが少なくともクォドワードでないと，呼び出されたルーチン，および呼び出されたルーチンが呼び出すものの中で，アライメント異常が発生する可能性があります。省略時の設定の動作 (INLINE=FALSE) ではこの問題は発生しません。

引数の個数が 6 個を越える場合，$SETUP_CALL64 と対応する$CALL64 との間で，AP または SP への参照は行われません。これは，$CALL64 が別の JSB ルーチンにある場合があるからです。また，一時的なレジスタ (R16 以上) は$SETUP_CALL64 以降，有効ではありません。ただし， R16 〜 R21 がすでに設定されている引数レジスタと関係する場合を除いて，これらを範囲の中で使用することはできます。このような場合，上位の一時レジスタを代わりに使用します。

注意

$SETUP_CALL64，$PUSH_ARG64，および$CALL64 マクロは，インライン・シーケンスの中で使用することを目的としています。つまり，$SETUP_CALL64/$PUSH_ARG64/$CALL64 シーケンスの中央に分岐したり，$PUSH_ARG64 マクロの回りで分岐することはできず，シーケンスを出て$CALL64 を避けることもできません。

$PUSH_ARG64

呼び出しの引数プッシュに相当する機能を実行します。

フォーマット

$PUSH_ARG64　**引数**

パラメータ

引数
プッシュされる引数。

説明

このマクロは，64 ビット呼び出しの 64 ビット引数をプッシュします。なお，$PUSH_ARG64 を使用する前にマクロ$SETUP_CALL64 を使用しなければなりません。

引数はアラインされたクォドワードとして読み込まれます。つまり，$PUSH_ARG64 4(R0)という指定は，4(R0) でクォドワードを読み込み，クォドワードをプッシュします。インデックス操作はクォドワード・モードで行われます。

メモリのロングワード値をクォドワード値としてプッシュする場合，まずこれを，ロングワード命令でレジスタに移動し，レジスタ上で$PUSH_ARG64 を使用します。同様に，アラインされていないクォドワード値をプッシュするには，まずこれを一時レジスタに移動し$PUSH_ARG64 を使用します。

呼び出しの引数の個数が 6 個を越える場合，このマクロは引数内の SP 参照または AP 参照をチェックします。呼び出しの引数の個数が 6 個を越える場合，SP 参照は認められず，また，AP 参照はインライン・オプションが使用されている場合に限って認められます。

マクロは，現在の$CALL64 に対してすでに設定されている，引数レジスタへの参照についてもチェックします。このような参照が検出されると，引数レジスタが$PUSH_ARG64 内でのソースとして使用される前に，これを上書きすることがないよう，注意を促す警告メッセージが表示されます。

引数の個数が 6 個以下の場合にも， AP 参照について同じチェックが行われます。これは認められますが，コンパイラではその使用前に上書きすることを禁止できません。このため，このような参照が検出されると，情報メッセージが表示されます。

オペランドが， R16 ～ R21 のいずれかの文字列をその名前に含むシンボルを，レジスタ参照以外で使用する場合，このマクロは，間違ったエラーを報告することがあります。たとえば， R21 が設定された後で`$PUSH_ARG64 SAVED_R21`を起動すると，このマクロは， (必要でないのに) 引数レジスタの上書きに関する情報メッセージを表示します。

$PUSH_ARG64 は条件付きコードの中で使用できないことにも注意してください。$PUSH_ARG64 は，引数の個数などを追跡するシンボルを更新します。$SETUP_CALL64/$CALL64 シーケンスの中央で，$PUSH_ARG64 の回りに分岐するコードを作成しようとしても，正しく動作しません。

$CALL64

ターゲット・ルーチンを起動します。

フォーマット

$CALL64　*call_target*

パラメータ

call_target
起動するルーチン。

説明

このマクロは，**$SETUP_CALL64** で引数の個数を指定し，**$PUSH_ARG64** でクォドワード引数をプッシュしたものとして，指定されたルーチンを呼び出します。このマクロは，プッシュの回数がセットアップ呼び出しの中で指定された回数に一致するかチェックします。

call_target オペランドが AP ベースまたは SP ベースであってはなりません。

B.2　符合拡張とディスクリプタ形式をチェックするマクロ

ここで説明するマクロは，特定の値を含んでいるかチェックし，そのチェックの結果に基づきプログラム・フローを決めます。

$IS_32BITS

64 ビット値の下位 32 ビットの符号拡張をチェックし，チェックの結果に基づいて，プログラム・フローを指示します。

フォーマット

$IS_32BITS *quad_arg, leq_32bits, gtr_32bits, temp_reg=22*

パラメータ

quad_arg
レジスタまたはアラインされたクォドワード・メモリ位置のいずれかにある 64 ビット値。

leq_32bits
quad_arg が 32 ビット符号拡張値の場合の分岐先のラベル。

gtr_32bits
quad_arg が 32 ビットを越える場合の分岐先のラベル。

temp_reg=22
ソース値の下位ロングワードを保持する一時レジスタとして使用するレジスタ。省略時の設定の値は R22 です。

説明

$IS_32BITS は 64 ビット値の下位 32 ビットの符号拡張をチェックし，チェックの結果に基づいて，プログラム・フローを指示します。

使用例

1. `$is_32bits  R9, 10$`

   この例では，省略時の設定の一時レジスタ R22 を使用して， R9 の 64 ビット値の下位 32 ビットの符号拡張をチェックします。分岐の種類およびチェックの結果によって，プログラムは分岐，またはインラインを継続します。

2. `$is_32bits  4(R8), 20$, 30$, R28`

   この例では，一時レジスタ **R28** を使用して， **4(R8)** の **64** ビット値の下位 **32** ビットの符号拡張をチェックし，その結果によって， **20$**または **30$**に分岐します。

$IS_DESC64

指定されたディスクリプタをチェックし，これが 64 ビット形式のディスクリプタかどうかを判断し，チェックの結果に基づいてプログラム・フローを指示します。

フォーマット

$IS_DESC *desc_addr, target, size=long or quad*

パラメータ

desc_addr
チェックするディスクリプタのアドレス。

target
ディスクリプタが 64 ビット形式の場合の分岐先のラベル。

size=long
ディスクリプタを示しているアドレスのサイズ。"long" (省略時の設定) および"quad"が有効。

説明

$IS_DESC64 は， 64 ビット・ディスクリプタと 32 ビット・ディスクリプタを区別するフィールドをチェックします。これが 64 ビット形式の場合，指定されたターゲットに分岐が行われます。 SIZE=QUAD が指定されていない限り，チェックするアドレスはロングワードとして読み込まれます。

使用例

1. `$is_desc64 r9, 10$`

   この例では， R9 によって示されるディスクリプタがチェックされ，これが 64 ビット形式の場合は 10$への分岐が行われます。

2. `$is_desc64 8(r0), 20$, size=quad`

   この例では， 8(R0) のクォドワードが読み込まれ，これが示すディスクリプタがチェックされます。これが 64 ビット形式の場合は 20$への分岐が行われます。

C

64 ビット・プログラム例

このサンプル・プログラムは， **64** ビット・リージョンの作成および削除のシステム・サービスを紹介します。 **SYS$CREATE_REGION_64** を使用してリージョンを作成し， **SYS$EXPREG_64** を使用してそのリージョンの中で仮想アドレスを割り当てます。仮想アドレス空間およびリージョンは， **SYS$DELETE_REGION_64** を呼び出すことによって削除されます。

```
/*
 *****************************************************************************
 *
 * Copytight (c) Digital Equipment Corporation, 1995 All Rights Reserved.
 * Unpublished rights reserved under the copyright laws of the United States.
 *
 * The software contained on this media is proprietary to and embodies the
 * confidential technology of Digital Equipment Corporation.  Possession, use,
 * duplication or dissemination of the software and media is authorized only
 * pursuant to a valid written license from Digital Equipment Corporation.
 *
 * RESTRICTED RIGHTS LEGEND   Use, duplication, or disclosure by the U.S.
 * Government is subject to restrictions as set forth in Subparagraph
 * (c)(1)(ii) of DFARS 252.227-7013, or in FAR 52.227-19, as applicable.
 *
 *****************************************************************************
/*
    This program creates a region in P2 space using the region creation
    service and then creates VAs within that region.  The intent is to
    demonstrate the use of the region services and how to allocate virtual
    addresses within a region.  The program also makes use of 64-bit
    descriptors and uses them to format return values into messages with the
    aid of SYS$GETMSG.

    To build and run this program type:

    $ CC/POINTER_SIZE=32/STANDARD=RELAXED/DEFINE=(__NEW_STARLET=1) -
        REGIONS.C
    $ LINK REGIONS.OBJ
    $ RUN REGIONS.EXE
*/
```

```
#include    <descrip.h>           /* Descriptor Definitions               */
#include    <far_pointers.h>      /* Long Pointer Definitions             */
#include    <gen64def.h>          /* Generic 64-bit Data Type Definition  */
#include    <iledef.h>            /* Item List Entry Definitions          */
#include    <ints.h>              /* Various Integer Typedefs             */
#include    <iosbdef.h>           /* I/O Status Block Definition          */
#include    <psldef.h>            /* PSL$ Constants                       */
#include    <ssdef.h>             /* SS$_ Message Codes                   */
#include    <starlet.h>           /* System Service Prototypes            */
#include    <stdio.h>             /* printf                               */
#include    <stdlib.h>            /* malloc, free                         */
#include    <string.h>            /* memset                               */
#include    <syidef.h>            /* $GETSYI Item Code Definitions        */
#include    <vadef.h>             /* VA Creation Flags and Constants      */

/*  Module-wide constants and macros.                                     */

#define     BUFFER_SIZE         132
#define     HW_NAME_LENGTH       32
#define     PAGELET_SIZE        512
#define     REGION_SIZE         128

#define     good_status(code)   ((code) & 1)

/*  Module-wide Variables                                                 */

int
    page_size;

$DESCRIPTOR64 (msgdsc, "");

/*  Function Prototypes                                                  */

int get_page_size (void);
static void print_message (int code, char *string);

main (int argc, char **argv)
{
    int
        i,
        status;

    uint64
        length_64,
        master_length_64,
        return_length_64;

    GENERIC_64
        region_id_64;

    VOID_PQ
        master_va_64,
        return_va_64;

/*  Get system page size, using SYS$GETSYI.                               */

    status = get_page_size ();
    if (!good_status (status))
        return (status);
```

```
/*  Get a buffer for the message descriptor.                              */

    msgdsc.dsc64$pq_pointer = malloc (BUFFER_SIZE);
    printf ("Message Buffer Address = %016LX\n\n", msgdsc.dsc64$pq_pointer);

/*  Create a region in P2 space.                                          */

    length_64 = REGION_SIZE*page_size;
    status = sys$create_region_64 (
        length_64,                /* Size of Region to Create          */
        VA$C_REGION_UCREATE_UOWN, /* Protection on Region              */
        0,                        /* Allocate in Region to Higher VAs  */
        &region_id_64,            /* Region ID                         */
        &master_va_64,            /* Starting VA in Region Created     */
        &master_length_64);       /* Size of Region Created            */
    if (!good_status (status))
    {
        print_message (status, "SYS$CREATE_REGION_64");
        return (status);
    }

    printf ("\nSYS$CREATE_REGION_64 Created this Region:  %016LX - %016LX\n",
        master_va_64,
        (uint64) master_va_64 + master_length_64 - 1);

/*  Create virtual address space within the region.                       */

    for (i = 0; i < 3; ++i)
    {
        status = sys$expreg_64 (
            &region_id_64,      /* Region to Create VAs In            */
            page_size,          /* Number of Bytes to Create          */
            PSL$C_USER,         /* Access Mode                        */
            0,                  /* Creation Flags                     */
            &return_va_64,      /* Starting VA in Range Created       */
            &return_length_64); /* Number of Bytes Created            */
        if (!good_status (status))
        {
            print_message (status, "SYS$EXPREG_64");
            return status;
        }
        printf ("Filling %016LX - %16LX with %0ds.\n",
            return_va_64,
            (uint64) return_va_64 + return_length_64 - 1,
            i);
        memset (return_va_64, i, page_size);
    }

/*  Return the virtual addresses created within the region, as well as the
    region itself.                                                        */

    printf ("\nReturning Master Region:  %016LX - %016LX\n",
        master_va_64,
        (uint64) master_va_64 + master_length_64 - 1);
```

```
    status = sys$delete_region_64 (
        &region_id_64,      /* Region to Delete                               */
        PSL$C_USER,         /* Access Mode                                    */
        &return_va_64,      /* VA Deleted                                     */
        &return_length_64); /* Length Deleted                                 */

    if (good_status (status))
        printf ("SYS$DELETE_REGION_64 Deleted VAs Between:  %016LX - %016LX\n",
            return_va_64,
            (uint64) return_va_64 + return_length_64 - 1);
    else
    {
        print_message (status, "SYS$DELTE_REGION_64");
        return (status);
    }

/*  Return message buffer.                                                    */

    free (msgdsc.dsc64$pq_pointer);
}

/*  This routine obtains the system page size using SYS$GETSYI.  The return
    value is recorded in the module-wide location, page_size.                 */

int get_page_size ()
{
int
    status;

IOSB
    iosb;

ILE3
    item_list [2];

/* Fill in SYI item list to retrieve the system page size.                    */

    item_list[0].ile3$w_length      = sizeof (int);
    item_list[0].ile3$w_code        = SYI$_PAGE_SIZE;
    item_list[0].ile3$ps_bufaddr    = &page_size;
    item_list[0].ile3$ps_retlen_addr = 0;
    item_list[1].ile3$w_length      = 0;
    item_list[1].ile3$w_code        = 0;

/* Get the system page size.                                                  */

    status = sys$getsyiw (
                0,                          /* EFN                    */
                0,                          /* CSI address            */
                0,                          /* Node name              */
                &item_list,                 /* Item list              */
                &iosb,                      /* I/O status block       */
                0,                          /* AST address            */
                0);                         /* AST parameter          */
```

```
    if (!good_status (status))
    {
        print_message (status, "SYS$GETJPIW");
        return (status);
    }
    if (!good_status (iosb.iosb$w_status))
    {
        print_message (iosb.iosb$w_status, "SYS$GETJPIW IOSB");
        return (iosb.iosb$w_status);
    }

    return SS$_NORMAL;
}

/*  This routine takes the message code passed to the routine and then uses
    SYS$GETMSG to obtain the associated message text.  That message is then
    printed to stdio along with a user-supplied text string.              */

#pragma inline (print_message)
static void print_message (int code, char *string)
{
    msgdsc.dsc64$q_length = BUFFER_SIZE;
    sys$getmsg (
        code,                                    /* Message Code         */
        (unsigned short *) &msgdsc.dsc64$q_length,  /* Returned Length      */
        &msgdsc,                                 /* Message Descriptor   */
        15,                                      /* Message Flags        */
        0);                                      /* Optional Parameter   */
    *(msgdsc.dsc64$pq_pointer+msgdsc.dsc64$q_length) = '\0';
     printf ("Call to %s returned:  %s\n",
        string,
        msgdsc.dsc64$pq_pointer);
}
```

D

VLM プログラム例

このサンプル・プログラムは，第 **4** 章で説明したメモリ管理 **VLM** 機能を紹介します。

```
/*
 ****************************************************************************
 *
 * Copyright (c) Digital Equipment Corporation, 1996 All Rights Reserved.
 * Unpublished rights reserved under the copyright laws of the United States.
 *
 * The software contained on this media is proprietary to and embodies the
 * confidential technology of Digital Equipment Corporation.  Possession, use,
 * duplication or dissemination of the software and media is authorized only
 * pursuant to a valid written license from Digital Equipment Corporation.
 *
 * RESTRICTED RIGHTS LEGEND   Use, duplication, or disclosure by the U.S.
 * Government is subject to restrictions as set forth in Subparagraph
 * (c)(1)(ii) of DFARS 252.227-7013, or in FAR 52.227-19, as applicable.
 *
 ****************************************************************************
/*
    This program creates and maps to a memory resident global section using
    shared page tables.  The program requires a reserved memory entry
    (named In_Memory_Database) in the Reserved Memory Registry.

    The entry in the registry is created using SYSMAN as follows:

    $ MCR SYSMAN
    SYSMAN> RESERVED_MEMORY ADD "In_Memory_Database"/ALLOCATE/PAGE_TABLES -
        /ZERO/SIZE=64/GROUP=100

    The above command creates an entry named In_Memory_Database that is
    64 Mbytes in size and requests that the physical memory be allocated
    during system initialization.  This enables the physcial memory to
    be mapped with granularity hints by the SYS$CRMPSC_GDZRO_64 system
    service.  It also requests that physical memory be allocated for
    page tables for the named entry, requests the allocated memory be
    zeroed, and requests that UIC group number 100 be associated with
    the entry.

    Once the entry has been created with SYSMAN, the system must be
    re-tuned with AUTOGEN.  Doing so allows AUTOGEN to re-calculate
    values for SYSGEN parameters that are sensitive to changes in
    physical memory sizes.  (Recall that the Reserved Memory Registry
    takes physical pages away from the system.)  Once AUTOGEN has
    been run, the system must be rebooted.

    Using the following commands to compile and link this program:
```

```
    $ CC/POINTER_SIZE=32 shared_page_tables_example
    $ LINK shared_page_tables_example

    Since 64-bit virtual addresses are used by this program, a Version
    5.2 DEC C compiler or later is required to compile it.
*/
#define     __NEW_STARLET   1

#include    <DESCRIP>
#include    <FAR_POINTERS>
#include    <GEN64DEF>
#include    <INTS>
#include    <PSLDEF>
#include    <SECDEF>
#include    <SSDEF>
#include    <STARLET>
#include    <STDIO>
#include    <STDLIB>
#include    <STRING>
#include    <VADEF>

#define     bad_status(status) (((status) & 1) != 1)
#define     ONE_MEGABYTE    0x100000


main ()
{
    int
        status;

    $DESCRIPTOR (section_name, "In_Memory_Database");

    uint32
        region_flags = VA$M_SHARED_PTS,   /* Shared PT region. */
        section_flags = SEC$M_EXPREG;

    uint64
        mapped_length,
        requested_size = 64*ONE_MEGABYTE,
        section_length = 64*ONE_MEGABYTE,
        region_length;

    GENERIC_64
        region_id;

    VOID_PQ
        mapped_va,
        region_start_va;

    printf ("Shared Page Table Region Creation Attempt:  Size = %0Ld\n",
        requested_size);

/*  Create a shared page table region. */
```

```
    status = sys$create_region_64 (
        requested_size,                 /* Size in bytes of region */
        VA$C_REGION_UCREATE_UOWN,       /* Region VA creation and owner mode */
        region_flags,                   /* Region Flags:  shared page tables */
        &region_id,                     /* Region ID */
        &region_start_va,               /* Starting VA for region */
        &region_length);                /* Size of created region */

    if (bad_status (status))
    {
        printf ("ERROR:  Unable to create region of size %16Ld\n\n",
            requested_size);
        return;
    }

    printf ("Shared Page Table Region Created:  VA = %016LX, Size = %0Ld\n\n",
        region_start_va,
        region_length);

/* Create and map a memory resident section with shared page tables
       into the shared page table region. */

    printf ("Create and map to section %s\n", section_name.dsc$a_pointer);
    status = sys$crmpsc_gdzro_64 (
            &section_name,              /* Section name */
            0,                          /* Section Ident */
            0,                          /* Section protection */
            section_length,             /* Length of Section */
            &region_id,                 /* RDE */
            0,                          /* Section Offset; map entire section */
            PSL$C_USER,                 /* Access Mode */
            section_flags,              /* Section Creation Flags */
            &mapped_va,                 /* Return VA */
            &mapped_length);            /* Return Mapped Length */

    if (bad_status (status))
        printf ("ERROR:  Unable to Create and Map Section %s, status = %08x\n\n",
                section_name.dsc$a_pointer,
                status);
    else
    {
        printf ("Section %s created, Section Length = %0Ld\n",
                section_name.dsc$a_pointer,
                section_length);
        printf ("    Mapped VA = %016LX, Mapped Length = %0Ld\n\n",
                mapped_va,
                mapped_length);
    }

    /* Delete the shared page table.  This will cause the mapping to the
       section and the section itself to be deleted. */
```

```
    printf ("Delete the mapping to the memory resident global section");
    printf (" and the shared\n    page table region.\n");
    status = sys$delete_region_64 (
            &region_id,
            PSL$C_USER,
            &region_start_va,
            &region_length);

    if (bad_status (status))
        printf ("ERROR:  Unable to delete shared page table region, status = %08x\n\n", status);
    else
        printf ("Region Deleted, Start VA = %016LX, Length = %016LX\n\n",
                region_start_va,
                region_length);
    printf ("\n");
}
```

This example program displays the following output:

```
Shared Page Table Region Creation Attempt:  Size = 67108864
Shared Page Table Region Created:  VA = FFFFFFFBFC000000, Size = 67108864

Create and map to section In_Memory_Database
Section In_Memory_Database created, Section Length = 67108864
    Mapped VA = FFFFFFFBFC000000, Mapped Length = 67108864

Delete the mapping to the memory resident global section and the shared
    page table region.
Region Deleted, Start VA = FFFFFFFBFC000000, Length = 0000000004000000
```

索引

C

$CALL64 マクロ 12–1, B–6
  64 ビット値の引き渡し 12–2
.CALL_ENTRY ディレクティブ
  QUAD_ARGS パラメータ
    64 ビット値の宣言 12–1, 12–5

D

.DISABLE ディレクティブ
  QUADWORD オプション 12–1

E

/ENABLE 修飾子
  QUADWORD オプション 12–1
.ENABLE ディレクティブ
  QUADWORD オプション 12–1
EVAX_CALLG_64 ビルトイン
  64 ビット・アドレス・サポート 12–2, 12–4
EVAX_SEXTL ビルトイン
  64 ビット・アドレス・サポートのための符合拡張 12–2, 12–8

I

$IS_32BITS マクロ
  符合拡張 12–1
  符合拡張のチェック 12–8, B–7
$IS_DESC64 マクロ
  形式ディスクリプタの確認 12–1
$IS_DESC マクロ
  ディスクリプタが 64 ビットかチェック . . . B–9

L

LIB$MOVC3 ルーチン 12–9
LIB$MOVC5 ルーチン 12–9
LIBOTS ルーチン 12–9

M

MACRO
  VAX MACRO 64 ビット・アドレッシング・サポート 12–1, B–1
MOVC3 命令 12–9
MOVC5 命令 12–9

O

OTS$MOVE3 ルーチン 12–9
OTS$MOVE5 ルーチン 12–9

P

P0 空間
  定義 2–2
P1 空間
  定義 2–2
$PUSH_ARG64 マクロ 12–1, B–4
  64 ビット値の引き渡し 12–2

R

RAB64(64 ビット・レコード・アクセス・ブロック)
  RAB64$PQ_xフィールド 5–2, 5–3
  RAB64$Q_xフィールド 5–2, 5–3
  データ構造 5–2
  マクロ 5–4
$RAB64_STORE マクロ 12–2, 12–9
$RAB64 マクロ 12–2, 12–9
$RAB_STORE マクロ 12–9
$RAB マクロ 12–9
RMS
  「RAB64(64 ビット・レコード・アクセス・ブロック)」を参照
  インタフェースの強化 5–1
  64 ビット・アドレッシングを対象とするインタフェースの強化 5–1
RMS マクロ
  64 ビット・アドレス空間のデータ・バッファ・サポート 12–2, 12–9

S

S0 空間
  定義 2–2
$SETUP_CALL64 マクロ 12–1, B–2
  64 ビット値の引き渡し 12–2

V

VAX MACRO
  「MACRO」を参照

ア

アセンブル言語命令
  Alpha ビルトイン 12–8
アドレッシング・ガイドライン
  64 ビット 12–1
アドレス
  64 ビット値の引き渡し 12–2, B–2
  64 ビットの計算 12–6

ク

クォドワード引数
  宣言 12–5
クォドワード・アドレス
  計算 12–6
クォドワード引数
  引き渡し 12–2

シ

システム空間
  定義 2–2
システム・サービス
  C 関数プロトタイプ 3–6
  MACRO-32 3–6

テ

ディスクリプタ形式
  $IS_DESC マクロでチェック B–9
デバッガ
  クォドワード 9–1

ヒ

引数
  クォドワード宣言 12–5
引数リスト
  可変サイズ 12–4
  固定サイズ 12–2
  ホーミング 12–5
ビルトイン
  Alpha アセンブル言語命令 12–8

フ

符合拡張
  EVAX_SEXTL ビルトインの使用 12–8
  $IS_32BITS マクロでチェック 12–8, B–7

プロセス・プライベート空間
  定義 2–2

へ

ページ・サイズ
  に基づく計算 12–1, 12–8
  64 ビット・アドレッシングのためのマクロ・パラメータ 12–1, 12–8

ホ

ポインタ
  64 ビットをサポート 10–1, 11–1
ポインタ・タイプ宣言 12–6

メ

命令
  Alpha アセンブル言語のためのコンパイラ・ビルトイン 12–8

ヨ

呼び出し標準 10–1

OpenVMS Alpha **オペレーティング・システム**
64 **ビット・アドレッシングおよび** VLM **機能説明書**

1999 年 4 月　発行

コンパックコンピュータ株式会社

〒 140-8641 東京都品川区東品川 2-2-24 天王洲セントラルタワー

電話　(03)5463-6600 (大代表)

AA-QTQ0C-TE