HP OpenVMS

HP C ランタイム・ライブラリ・リファレンス・マニュアル (上巻)

5991-6625.2

2011 年 5 月

本書は, OpenVMS システム用の HP C ランタイム・ライブラリの機能とマクロについて説明します。

改訂／更新情報: 本書は『HP C Run-Time Library Reference Manual for OpenVMS Systems』 V8.3 の改訂版です。

ソフトウェア・バージョン: HP OpenVMS Integrity V8.4
HP OpenVMS Alpha V8.4

日本ヒューレット・パッカード株式会社

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

HP Cランタイム・ライブラリの一部は，カリフォルニア大学バークレイ校およびその協力者 (contributors) が著作権を保有するソースを使用して実装されています。

次の条件が満たされる場合，変更されているかかどうかにかかわらず，ソースおよびバイナリ形式の再配布と使用が認められます。

1. ソース・コードを再配布する際は，上記の著作権に関する通告，再配布と使用に関するこの条件一覧，以下の免責事項を添付する必要があります。
2. バイナリ形式での再配布の際は，添付するドキュメントや他のマテリアルとともに，上記の著作権に関する通告，再配布と使用に関するこの条件一覧，以下の免責事項を添付する必要があります。
3. 本ソフトウェアの機能や本ソフトウェアを使用していることを記載した宣伝広告資料すべてに，次の情報を記載する必要があります。「本製品には，カリフォルニア大学バークレイ校およびその協力者が開発したソフトウェアが含まれています。」
4. 事前に書面による承認を受けない限り，本ソフトウェアを利用して開発された製品の宣伝や販売促進で，大学の名前や協力者の名前を使用することは認められません。

本ソフトウェアは，開発者および協力者が提供するものを「そのまま」提供するものであり，商品性や特定の目的への適合性の暗黙の保証も含めて (これらに限定されません)，いかなる表明や暗黙の保証も適用されません。いかなる場合も，開発者および協力者は，直接的または間接的損害，事故による損害，特別損害など (たとえば，代替商品やサービスの調達の必要性，製品が使用できなくなる障害，データや収益の消失，ビジネスの中断など) の発生の責任を負いません。さらに，本ソフトウェアの使用によって発生する契約上の責任，無過失責任，不法行為 (過失によるもの他) に関しても，そのような損害の可能性があらかじめ助言されていた場合でも，開発者および協力者は一切の責任を負いません。

目次

図

表

まえがき

本書では， OpenVMS Alpha および Intel Itanium プロセッサ対応の OpenVMS Integrity のHP C Run-Time Library (RTL) について説明します。 HP OpenVMS Industry Standard 64 for Integrity Servers は， Intel Itanium プロセッサ対応 OpenVMS オペレーティング・システムの正式な製品名です。

本書は，入出力 (I/O) 操作，文字および文字列操作，算術演算，エラー検出，サブプロセスの生成，システム・アクセス，画面管理， UNIX の一部の機能のエミュレーションを行う C RTL 関数およびマクロに関する参照情報を示します。また，オペレーティング・システム間の移植性に関する問題点も示します。

HP C RTL には， XPG4 準拠の国際化サポートが含まれており，異なる言語やカルチャで動作可能なソフトウェアを開発するのに役立つ関数が提供されます。

HP CコンパイラおよびC++コンパイラとともに使用する必要がある完全な HP C ランタイム・ライブラリ (C RTL) は， OpenVMS Alpha および OpenVMS Integrity システムで，共用イメージおよびオブジェクト・モジュール・ライブラリの両方の形式で提供されます。

本書では， TCP/IP サービス・プロトコルを使用したインターネット・アプリケーションを作成するためのソケット・ルーチンについては説明していません。ソケット・ルーチンのヘルプについては，次のコマンドを実行してください。

```
$ HELP TCPIP_Services Programming_Interfaces Sockets_API
```

また， HP TCP/IP Services for OpenVMS の製品マニュアルも参照してください。

本書の対象読者

本書は，HP C RTL で提供される関数とマクロに関する情報を必要とするプログラマを対象にしています。

本書の構成

本書は次の章と付録で構成されています。

- 第 1 章では，HP C RTL の概要を示します。
- 第 2 章では，標準 I/O，端末 I/O， UNIX I/O 関数について説明します。

- 第 3 章では，文字，文字列，引数リスト関数について説明します。
- 第 4 章では，エラー処理関数とシグナル処理関数について説明します。
- 第 5 章では，サブプロセスを生成するために使用される関数について説明します。
- 第 6 章では， Curses 画面管理関数について説明します。
- 第 7 章では，算術演算関数について説明します。
- 第 8 章では，メモリ割り当て関数について説明します。
- 第 9 章では，オペレーティング・システムとやり取りするために使用される関数について説明します。
- 第 10 章では，国際化ソフトウェアの開発のために OpenVMS システムのHP C環境で提供される機能について，その概要を紹介します。
- 第 11 章では，日付/時刻関数について説明します。
- 第 12 章では，シンボリック・リンクと POSIX パス名のサポートについて説明します。
- 付録 A では，異なる OpenVMS バージョンでサポートされるHP C RTL 関数の一覧を示した，バージョン依存の表を示します。
- 付録 B では，複数のヘッダ・ファイルに重複している関数プロトタイプを示します。

リファレンス・セクションについて

HP C RTL の各関数についての説明は，『HP C ランタイム・ライブラリ・リファレンス・マニュアル (下巻)』で提供しています。

関連ドキュメント

OpenVMS システム向けのプログラムをHP Cで作成する場合，次のドキュメントが役立ちます。

- 『HP C User's Guide for OpenVMS Systems』 — HP C for OpenVMS Systems の使用方法に関する情報が必要な C プログラマを対象にしています。
- 『HP C Language Reference Manual』 — HP システムでのHP Cの言語リファレンス情報を示します。
- 『VAX C to HP C Migration Guide』 — OpenVMS VAX アプリケーション・プログラマがVAX CからHP Cに移行するのに役立ちます。
- 『HP C Installation Guide for OpenVMS VAX Systems』 — VAX システムにHP Cソフトウェアをインストールする OpenVMS システム・プログラマを対象にしています。

- 『HP C Installation Guide for OpenVMS Alpha Systems』 — Alphaシステムにhp Cソフトウェアをインストールする OpenVMS システム・プログラマを対象にしています。
- 『OpenVMS Master Index』 — VAX および Alpha マシン・アーキテクチャや OpenVMS システム・サービスを使用する必要のあるプログラマを対象にしています。このインデックスには，OpenVMS オペレーティング・システムへのアクセスに関する個別のトピックを説明したドキュメントの一覧が示されています。
- 『HP TCP/IP Services for OpenVMS Sockets API and System Services Programming』 — HP TCP/IP Services for OpenVMS 製品または他の TCP/IP プロトコル実装用の，インターネット・アプリケーション・プログラムを作成するためのソケット・ルーチンについての情報を示します。
- 『HP TCP/IP Services for OpenVMS Guide to IPv6』 — HP TCP/IP Services for OpenVMS の Ipv6 機能や，システム上での IPv6 のインストールや構成方法，ソケット・アプリケーション・プログラミング・インタフェース (API) の変更，IPv6 環境で動作させるためにアプリケーションを移植する方法についての情報を示します。
- 『X/Open Portability Guide, Issue 3』 — 一般に XPG3 と呼んでいる仕様について解説しています。
- 『X/Open CAE Specification System Interfaces and Headers, Issue 4』 — 一般に XPG4 と呼んでいる仕様について解説しています。
- 『X/Open CAE Specification, System Interfaces and Headers, Issue 4, Version 2』 — 一般に XPG4 V2 と呼んでいる仕様に関して解説しています。
- 『X/Open CAE Specification, System Interfaces and Headers, Issue 5』 — 一般に XPG5 と呼んでいる仕様に関して解説しています。
- 『Technical Standard. System Interfaces, Issue 6』 — Open Group の技術標準と IEEE 標準を組み合わせたものです。 XPG6 とも呼ぶ IEEE Std 1003.1-2001 仕様に関して解説しています。
- 『Standard for Information Technology - Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API)—Amendment 2: Threads Extension [C Language]』 — 一般に POSIX 1003.1c-1995 と呼んでいる仕様に関して解説しています。
- 『ISO/IEC 9945-2:1993 - Information Technology - Portable Operating System Interface (POSIX) - Part 2: Shell and Utilities』 — 一般に ISO POSIX-2と呼んでいる仕様に関して解説しています。
- 『ISO/IEC 9945-1:1990 - Information Technology - Portable Operating System Interface (POSIX) - Part 1: System Application Programming Interface (API) (C Language)』 — 一般に ISO POSIX-1と呼んでいる仕様について解説しています。

- 『ANSI/ISO/IEC 9899:1999 - Programming Languages - C』 — 1999 年 12 月に ISO によって公開され， 2000 年 4 月に ANSI 標準として採用された C99 標準について解説しています。
- 『ISO/IEC 9899:1990-1994 - Programming Languages - C, Amendment 1: Integrity』 — 一般にISO C， Amendment 1 と呼んでいる仕様について解説しています。
- 『ISO/IEC 9899:1990[1992] - Programming Languages - C』 — 一般にISO Cと呼んでいる仕様について解説しています。標準的な部分 (normative part) はX3.159-1989，American National Standard for Information Systems - Programming Language C (ANSI Cとも呼ぶ) と同じです。

HP OpenVMS 製品およびサービスについての詳細は，弊社の Web サイトを参照してください。アドレスは次のとおりです。

https://www.hpe.com/jp/openvms (日本語)
https://www.hpe.com/info/openvms (英語)

本書で使用する表記法

表記法	意味
HP OpenVMS Integrity	Intel Itanium アーキテクチャで動作する OpenVMS オペレーティング・システムです。
OpenVMS システム	特に明記しない限り，サポートする全プラットフォームの OpenVMS オペレーティング・システムを指します。
Return	Return は端末上の Return キーを 1 回押すことを示します。
Ctrl/X	Ctrl/X (英字の X は端末の制御文字を表す) は， Ctrl キーを押したまま指定の端末文字キー (X) を押すことを示します。
switch文 intデータ型 fprintf関数 <stdio.h>ヘッダ・ファイル	モノスペース文字は言語キーワードおよびHP C関数とヘッダ・ファイルの名前を示します。また，例で使用している特定の変数名を参照するときも使用します。
arg1	斜体は引数やパラメータ名を示すプレースホルダとして使用し，新出用語を強調するときも使用します。
$ **RUN CPROG** Return	ユーザと対話する例では，ユーザ入力は太字で示します。
float x; . . . x = 5;	垂直方向の省略記号は，プログラムやプログラムからの出力の一部が省略されていることを示します。例では関連する部分だけが示されています。
option, . . .	水平方向の省略記号は，パラメータ，オプション，値を追加入力できることを示します。省略記号の前のコンマは，後続の項目の区切り文字を表します。
[output-source, . . .]	関数構文やその他の場所で使用している角括弧は，その構文要素が省略可能であることを示します。しかし， OpenVMS ファイル指定でディレクトリ名を区切るために使用する角括弧や，HP Cソース・コードで多次元配列の次元を区切るために使用する角括弧は省略できません。

表記法	意味
sc-specifier::= auto static extern register	構文定義で，別々の行に示されている項目は組み合わせて指定できないことを示します。
[a | b]	2 つ以上の項目が縦線(|)で区切られ，角括弧で囲まれている場合は， 2 つの構文要素のいずれかを選択しなければならないことを示します。
Δ	デルタ記号は， 1 文字の ASCII スペース文字を表します。

プラットフォーム・ラベル

プラットフォームは，異なる環境を提供するオペレーティング・システムとハードウェアの組み合わせです。本書では， VAX， Alpha， Itanium プロセッサで動作する OpenVMS オペレーティング・システムに適用される情報を示します。

次のように特に指定した場合を除き，本書の情報はすべてのプロセッサに適用されます。

ラベル	説明
(Alpha only)	Alpha プロセッサ固有。
(Integrity only)	OpenVMS オペレーティング・システムが動作している Intel Itanium プロセッサ固有。このプラットフォームでは，オペレーティング・システムの製品名は OpenVMS Integrity です。
(Integrity，Alpha)	Integrity サーバおよび Alpha プロセッサ固有。

新機能および変更された機能 - OpenVMS Version 8.4

以降の項で， OpenVMS Version 8.4 で提供する C ランタイム・ライブラリ (C RTL) の機能拡張について説明します。これらの機能拡張により， UNIX との互換性，標準規格への準拠，ユーザ制御機能の選択の柔軟性などが向上しています。新しい C RTL 関数も含まれています。

Unicode サポート

C RTL で， UNIX スタイルのファイル名に対する Unicode UTF-8 エンコーディングをサポートしています。たとえば次のようなファイル名が使用できるようになっています。

```
/disk/mydir/^U65E5^U672C^U8A9E.txt filename
```

この機能より， UTF-8 エンコードのファイル名を使用する国際化アプリケーションの UNIX 互換性が向上しています。

新しい論理名 DECC$FILENAME_ENCODING_UTF8 によりこの機能を有効にすることができます。

この論理名が未定義の場合，デフォルトの動作で ASCII および Latin-1 をファイル名に使用できます。

この機能は ODS-5 ディスク上でのみ動作します。この機能を有効にするには，DECC$FILENAME_ENCODING_UTF8 と DECC$EFS_CHARSET の両方の論理名を定義する必要があります。

セマフォ・サポート

C RTL は，以下の Open Group セマフォ制御操作をサポートします。

System V セマフォ・ルーチン:

```
semctl()
semget()
semop()
ftok()
```

POSIX セマフォ・ルーチン:

```
sem_close()
sem_destroy()
sem_getvalue()
sem_init()
sem_open()
sem_post()
sem_timedwait()
sem_trywait()
sem_unlink()
sem_wait()
```

注意

すべてのセマフォ・ルーチンに適用される注意事項

セマフォの API **である**semget**および**sem_open**が，エラー状態値** 28 **を返す場合があります。この値はデバイス上に空きスペースが残っていないことを示すため，** GBLSECTIONS SYSGEN **パラメータの増加を検討してください。セマフォは内部的にはグローバル・セクションで，システムに多数のセマフォ・セットが存在すると** GBLSECTIONS **を使い果たします。このため，** SYSGEN **パラメータの値を増加させる必要があります。**

System V セマフォの制限事項

System V セマフォには以下の制限事項があります。

- システムでサポートする System V セマフォ・セットの最大数は 1024 です。
- 1 つのセマフォ・セット内でサポートする System V セマフォ・セットの最大数は 1024 です。
- セマフォの最大値は 32767 です。
- 1 つのプロセスでサポートする System V SEM_UNDO 操作の最大数は 1024 です。

DECC$PRINTF_USES_VAX_ROUND 機能スイッチ

新しい機能スイッチ DECC$PRINTF_USES_VAX_ROUND が C RTL に追加されています。

このスイッチを設定すると，printfの F および E 書式指定子は， IEEE 浮動小数点でコンパイルしたプログラムに対して VAX の切り上げ切捨て規則を使用します。

シンボリック・リンクと POSIX 準拠のパス名のサポートの拡張

OpenVMS Version 8.3

OpenVMS Version 8.3 で， Open Group 準拠のシンボリック・リンクのサポートと POSIX 準拠のパス名のサポートが拡張されています。

拡張の内容は以下のとおりです。

- POSIX ファイル名およびシンボリック・リンクにおける論理名のサポート
- RMS ディレクトリ・ワイルドカードにおけるループの検知
- RMS ディレクトリ・ワイルドカード検索におけるシンボリック・リンクのサポート
- オンディスク symlink 表現の再設計
- その他のバグの修正

1
はじめに

ISO/ANSI C 標準では，ANSI C の実装で提供される関数，マクロ，関連する型を登録したライブラリを定義しています。『HP C Language Reference Manual』では，すべてのHP Cプラットフォームに共通の ANSI 準拠のライブラリの機能について説明しています。『HP Cランタイム・ライブラリ・リファレンス・マニュアル』では，これらのルーチンについてさらに詳しく説明し，OpenVMS 環境でこれらのルーチンを使用する方法についても説明します。また，OpenVMS システムで提供される追加のヘッダ・ファイル，関数，型，マクロについても説明します。

すべてのライブラリ関数は**ヘッダ・ファイル**で宣言されます。ヘッダ・ファイルの内容をプログラムで使用できるようにするには，#includeプリプロセッサ・ディレクティブを使用してヘッダ・ファイルを取り込みます。次の例を参照してください。

```
#include <stdlib.h>
```

各ヘッダ・ファイルには，関連する関数の集合に対する関数プロトタイプが格納されており，これらの関数を使用するのに必要な型とマクロを定義しています。

OpenVMS Alpha または Integrity システムでヘッダ・ファイルの一覧を表示するには，次のコマンドを使用します。

```
$ LIBRARY/LIST SYS$LIBRARY:SYS$STARLET_C.TLB
$ LIBRARY/LIST SYS$LIBRARY:DECC$RTLDEF.TLB
$ DIR SYS$COMMON:[DECC$LIB.REFERENCE.DECC$RTLDEF]*.H;
$ DIR SYS$LIBRARY:*.H;
```

最初のコマンドは，OpenVMS システム・インタフェースのヘッダ・ファイルのテキスト・モジュール形式を一覧表示します。2 番目のコマンドは，HP C言語インタフェースのヘッダ・ファイルのテキスト・モジュール形式を一覧表示します。3 番目のコマンドは，HP C言語イタンフェースの*.H ヘッダ・ファイルを一覧表示します。4 番目のコマンドは，レイヤード・プロダクトおよび他のアプリケーションの*.H ヘッダ・ファイルを一覧表示します。

注意

SYS$COMMON:[DECC$LIB.REFERENCE.DECC$RTLDEF]**ディレクトリは，表示のための単なる参照領域です。**#include**でファイルを検索する場合，コンパイラは**.TLB **ファイルを確認します。**

しかし，SYS$LIBRARY から検索される重複ファイル (<stdio.h>など) は，おそらくVAX C Version 3.2 環境をサポートするものであり，HP Cでは使用されません。

関数定義自体がヘッダ・ファイルに含まれているわけではなく，これらの定義はOpenVMS オペレーティング・システムに付属しているHP C Run-Time Library (RTL) に格納されています。HP C RTL を使用する前に，次のことを十分理解しておく必要があります。

- リンク・プロセス
- マクロ置換プロセス
- 関数定義と関数呼び出しの違い
- 正しいファイル指定の形式
- OpenVMS 固有の入出力 (I/O) の方法
- HP C for OpenVMS の拡張機能と非標準機能

HP C RTL を効果的に使用するには，これらのすべてのトピックに関する知識が必要です。この章では，これらのトピックとHP C RTL との関連を示します。本書の他の章を読む前に，この章を必ずお読みください。

HP C RTL の基本的な目的は，C プログラムで I/O 操作を実行するための手段を提供することです。C 言語自体には情報の読み書き機能はありません。I/O のサポートの他に，HP C RTL では他の多くの作業を実行する手段も提供されます。

第 2 ～ 11 章では，HP C RTL でサポートされるさまざまなタスクについて説明します。「リファレンス・セクション」では，これらのタスクを実行するために提供されるすべての関数とマクロをアルファベット順に示し，詳しく説明します。

1.1 HP C Run-Time Library の使用

HP C RTL を使用する場合，実装固有の特徴を理解しておく必要があります。

まず，C プログラムでHP C RTL の関数を使用する場合，mainという名前の関数，またはmain_programオプションを使用する関数がプログラム内に存在することを確認してください。詳細については，『HP C Language Reference Manual』または『HP C User's Guide for OpenVMS Systems』を参照してください。

次に，HP C RTL 関数は実行時に実行されますが，これらの関数に対する参照はリンク時に解決されます。プログラムをリンクすると，OpenVMS リンカは，LINK コマンド・ラインに指定された共用可能コード・ライブラリやオブジェクト・コード・ライブラリを検索することにより，HP C RTL 関数に対するすべての参照を解決します。

HP C RTL は共用可能イメージとして使用でき，HP C RTL オブジェクト・ライブラリを使用することもできます。

HP C RTL を共用可能イメージとして使用する場合， RTL のコードは SYS$SHARE 内のイメージ・ファイルに存在し，すべてのHP Cプログラムで共用されます。実行後，制御はユーザ・プログラムに返されます。このプロセスには次のような多くの利点があります。

- プログラムの実行可能イメージのサイズが小さくなります。
- プログラムのイメージが使用するディスク空間が少なくなります。
- サイズが小さくなるため，メモリとの間のプログラムのスワップ速度が向上します。
- HP Cおよび HP C++ を使用すれば，共用可能イメージに対してプログラムをリンクするときにオプション・ファイルを定義する必要はありません。 RTL 共用可能イメージに対するリンクは，VAX Cの場合よりはるかに簡単になりました。実際に，このリンクはHP C RTL へのリンクのデフォルト方式になっています。

HP C RTL にリンクする場合， LNK$LIBRARY 論理名を定義する必要はありません。実際には，共用可能イメージとのリンクの方がHP C RTL オブジェクト・ライブラリとのリンクより便利であるため， LNK$LIBRARY の割り当ては解除する必要があります。

HP C RTL とのリンクに関する補足情報については， OpenVMS，HP C， HP C++ のリリース・ノートを参照してください。

1.2 RTL リンク・オプション

ここでは， OpenVMS Alpha システムと OpenVMS Integrity システムでHP CおよびHP C++ プログラムをHP C RTL とリンクする複数の方法について説明します。

1.2.1 共用可能イメージとのリンク

ほとんどのリンクでは， ALPHA$LIBRARY ディレクトリ*(Alpha only)*または IA64$LIBRARY *(Integrity only)* ディレクトリ内のHP C RTL 共用可能イメージ DECC$SHR.EXE を使用します。

共用可能イメージ VAXCRTL.EXE および VAXCRTLG.EXE は， OpenVMS Alpha と OpenVMS Integrity システムにはありません。唯一の C RTL 共用可能イメージは ALPHA$LIBRARY:DECC$SHR.EXE *(Alpha only)*または IA64$LIBRARY:DECC$SHR.EXE *(Integrity only)* であり，リンカは IMAGELIB.OLB を通じて自動的にこのイメージを検索します。

VAXCRTL*.EXE はAlphaシステムと Integrity システムに存在しないため，次のことに注意する必要があります。

- VAXCRTL*.EXE イメージに対する参照を除外するために，既存のVAX Cリンク・プロシージャを変更する必要があります。 IMAGELIB.OLB はリンカによって自動的に検索されるため， DECC$SHR.EXE を明示的に参照する必要はありません (『OpenVMS Linker Utility Manual』を参照)。
- DECC$SHR.EXE は接頭語の付いたユニバーサル・シンボル (DECC$から始まるシンボル) だけをエクスポートするので，正しくリンクするには，使用するすべてのHP C RTL エントリ・ポイントに対して接頭語を付ける必要があります。

  ANSI C 標準に定義されているHP C RTL 関数だけを使用する場合は，すべてのエントリ・ポイントに接頭語が付加されます。

  ANSI C 標準に定義されていないHP C RTL 関数を使用する場合は，次のいずかの方法でコンパイルすることにより，接頭語を付加する必要があります。

  – /PREFIX_LIBRARY_ENTRIES=ALL_ENTRIES修飾子を指定してコンパイルする方法。
  – /STANDARD=VAXC または/STANDARD=COMMON 修飾子を指定してコンパイルする方法。デフォルトは/PREFIX_LIBRARY_ENTRIES=ALL_ENTRIESです。

共用可能イメージに対してリンクするには， LINK コマンドを使用します。次の例を参照してください。

```
$ LINK PROG1
```

リンカは自動的に IMAGELIB.OLB から DECC$SHR.EXE を検索し，すべての C RTL 参照を解決します。

1.2.2 オブジェクト・ライブラリとのリンク*(Alpha only)*

OpenVMS Alpha システムのHP C RTL オブジェクト・ライブラリは，/PREFIX=ALL を使用せずにコンパイルされたプログラムをリンクする場合にだけ使用します。これらのオブジェクト・ライブラリは， OpenVMS Integrity システムには存在しない点に注意してください。

OpenVMS Alpha システムでは，HP C RTL は ALPHA$LIBRARY ディレクトリに次のオブジェクト・ライブラリを提供します。

- VAXCCURSE.OLB
- VAXCRTLD.OLB
- VAXCRTLT.OLB
- VAXCRTL.OLB

- VAXCRTLX.OLB
- VAXCRTLDX.OLB
- VAXCRTLTX.OLB

Curses 関数へのアクセスを可能にするオブジェクト・ライブラリ VAXCCURSE.OLB には，接頭語のないエントリ・ポイントが含まれており，これらのエントリ・ポイントは接頭語の付いた適切なエントリ・ポイントに変換されます。

オブジェクト・ライブラリ VAXCRTL.OLB， VAXCRTLD.OLB， VAXCRTLT.OLB， VAXCRTLX.OLB， VAXCRTLDX.OLB， VAXCRTLTX.OLB には，使用するオブジェクト・ライブラリに指定されている浮動小数点型に応じて，適切な接頭語の付いたエントリ・ポイントに変換される接頭語のないエントリ・ポイントも含まれています。

- VAXCRTL.OLB には，すべてのHP C RTL ルーチン名エントリ・ポイントの他に， VAX G-floating 倍精度浮動小数点エントリ・ポイントも含まれています。
- VAXCRTLD.OLB には， VAX D-floating 倍精度浮動小数点エントリ・ポイントの限定サポートが含まれています。
- VAXCRTLT.OLB には， IEEE T-floating倍精度浮動小数点エントリ・ポイントが含まれています。
- VAXCRTLX.OLB には， G_floating のサポートと，/L_DOUBLE_SIZE=128コンパイラ修飾子のサポートが含まれています。
- VAXCRTLDX.OLB には， D_floating のサポートと，/L_DOUBLE_SIZE=128コンパイラ修飾子のサポートが含まれています。
- VAXCRTLTX.OLB には， IEEE T_floating のサポートと，/L_DOUBLE_SIZE=128 コンパイラ修飾子のサポートが含まれています。

/L_DOUBLE_SIZE=128 がデフォルトです。

LINK コマンドには， VAXCRTL*.OLB ライブラリを 1 つだけ指定し，必要に応じて VAXCCURSE.OLB ライブラリも指定します。

コンパイラのデフォルト・モード (/STANDARD=RELAXED_ANSI89) および厳密なANSI Cモードでは，ANSI C標準ライブラリ・ルーチンに対するすべての呼び出しに，接頭語 DECC$が自動的に付加されます。/[NO]PREFIX_LIBRARY_ENTRIES 修飾子を使用すると，この動作を変更して，すべてのHP C RTL の名前に DECC$という接頭語を付けるか，HP C RTL の名前に接頭語を付けないようにすることができます。この修飾子には他のオプションも用意されています。詳細については，この章の/[NO]PREFIX_LIBRARY_ENTRIES 修飾子の説明を参照してください。

/NOSYSSHR を使用してリンクするときに，HP C RTL ルーチンの呼び出しに DECC$という接頭語を付ける場合，リンクが必要なのは STARLET.OLB 内のモジュールだけです。 STARLET.OLB はリンカで自動的に検索されるため (リンク修飾子/NOSYSLIB を使用しない限り)，接頭語の付いたすべての RTL 外部名は自動的に解決されます。

HP C RTL ルーチンの呼び出しに接頭語を付けない場合は，必要な浮動小数点型に応じて，あるいは Curses 関数が必要かどうかに応じて，VAXCRTL.OLB， VAXCRTLD.OLB， VAXCRTLT.OLB (または VAXCRTLX.OLB, VAXCRTLDX.OLB， VAXCRTLDX.OLB)， VAXCCURSE.OLB のいずれかに対して明示的にリンクする必要があります。/NOSYSSHR を使用してリンクする場合は，接頭語の付いたHP C RTL エントリ・ポイントは STARLET.OLB で解決されます。/SYSSHR (デフォルト) を使用してリンクする場合は，接頭語の付いたHP C RTL エントリ・ポイントは DECC$SHR.EXE で解決されます。

1.2.3 例

次の例では，HP C RTL とリンクするための複数の方法を示しています。図 1–1 は，これらの例を図でわかりやすく示しています。

1. ほとんどの場合，共用可能イメージに対するリンクだけが必要です。

```
$ CC/PREFIX_LIBRARY_ENTRIES=ALL_ENTRIES PROG1
$ LINK PROG1
```

   リンカは IMAGELIB.OLB から自動的に DECC$SHR.EXE を検索します。

2. オブジェクト・ライブラリだけを使用する場合 (たとえば，特権付きコードを作成するためや配布を簡単にするため)， LINK コマンドの/NOSYSSHR 修飾子を使用します。

```
$ CC/PREFIX_LIBRARY_ENTRIES=ALL_ENTRIES PROG1
$ LINK/NOSYSSHR PROG1
```

   ユーザ・プログラム内で接頭語の付いた RTL シンボル参照は， STARLET.OLB に含まれるHP C RTL オブジェクト・ライブラリで解決されます。

注意

- /NOSYSSHR **修飾子を使用して，**HP C**プログラムを**HP C RTL **オブジェクト・ライブラリに対してリンクする場合，未定義グローバルを含まずにすでにリンクされているアプリケーションでは，**CMA$TIS **シンボルの未定義グローバルが発生することがあります。これらの未定義グローバルを解決するには，次の行をリンク・オプション・ファイルに追加します。**

```
SYS$LIBRARY:STARLET.OLB/LIBRARY/INCLUDE=CMA$TIS
```

- /NOSYSSHR 修飾子を使用してリンクしたプログラムで，動的に起動されるイメージ内に常駐するルーチンを呼び出し，そのルーチンが異常終了状態を示す値を返した場合，errnoは ENOSYS に設定され，vaxc$errnoはC$_NOSYSSHRに設定されます。C$_NOSYSSHRに対応するエラー・メッセージは "Linking /NOSYSSHR disables dynamic image activation"です。たとえば，ソケット・ルーチンを呼び出すプログラムを，/NOSYSSHR を使用してリンクすると，このような状況が発生します。

---

3. *(Alpha only)* OpenVMS Alpha システムでは，接頭語を付加しないように設定してコンパイルした場合， C RTL 関数の別の実装を提供するオブジェクト・ライブラリを使用するには， VAXC*.OLB オブジェクト・ライブラリを使用する必要があります。この場合，コンパイルとリンクは次のように行います。

```
$ CC/NOPREFIX_LIBRARY_ENTRIES PROG1
$ LINK PROG1, MYLIB/LIBRARY, ALPHA$LIBRARY:VAXCRTLX.OLB/LIBRARY
```

ユーザ・プログラム内で接頭語のないHP C RTL シンボル参照は， MYLIB および VAXCRTL.OLB で解決されます。

VAXCRTLX.OLB 内で接頭語の付いたHP C RTL シンボル参照は，IMAGELIB.OLB を通じて DECC$SHR.EXE で解決されます。

この同じ例で， IEEE T-floating 倍精度浮動小数点をサポートするには，次のコンパイルおよびリンク・コマンドを使用します。

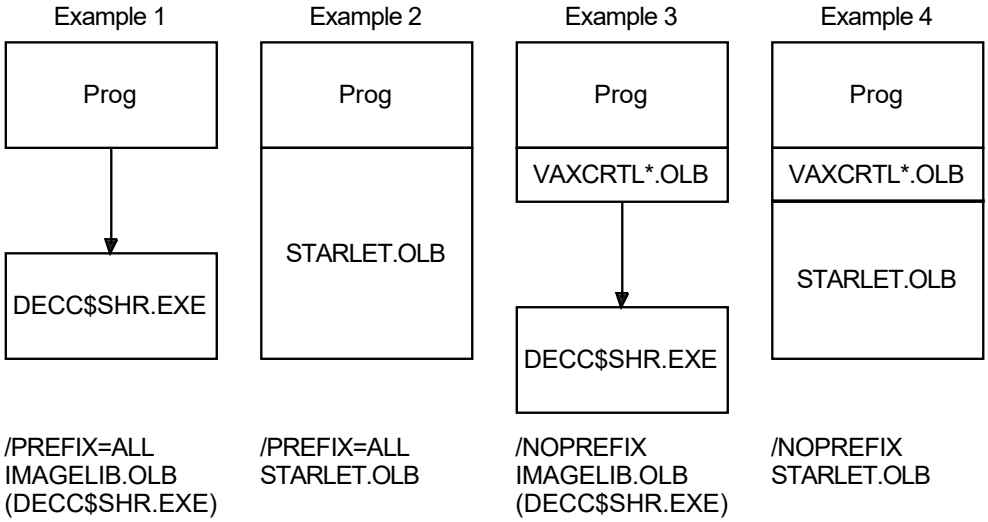
```
$ CC/NOPREFIX_LIBRARY_ENTRIES/FLOAT=IEEE_FLOAT PROG1
$ LINK PROG1, MYLIB/LIBRARY, ALPHA$LIBRARY:VAXCRTLTX.OLB/LIBRARY
```

4. *(Alpha only)*例 2 と例 3 を組み合わせて，オブジェクト・ライブラリ (特権付きコードを作成するためや配布を簡単にするため) だけを使用し， C RTL 関数を提供するオブジェクト・ライブラリを使用しなければならないことがあります。この場合，コンパイルとリンクは次の方法で行います。

```
$ CC/NOPREFIX_LIBRARY_ENTRIES PROG1
$ LINK/NOSYSSHR PROG1, MYLIB/LIBRARY, ALPHA$LIBRARY:VAXCRTLX.OLB/LIBRARY
```

VAXCRTL.OLB 内で接頭語の付いたHP C RTL シンボル参照は，STARLET.OLB で解決されます。

図 1–1 OpenVMS Alpha と OpenVMS Integrity での HP C RTL とのリンク



1.2.4 DECC$SHRP.EXE イメージ

OpenVMS は，保護モードで必要となる C RTL 関数を実装する新しい共有イメージ DECC$SHRP.EXE をインストールします。この共有イメージはすべての Alpha および Integrity システムにインストールされ， DECC$SHR.EXE あるいは DECC$SHR_EV56.EXE 共有イメージから起動されます。

1.3 HP C RTL 関数プロトタイプと構文

オブジェクト・モジュールをリンクする方法およびヘッダ・ファイルを取り込む方法を学習した後，プログラム内でHP C関数を参照する方法を学習する必要があります。本書の第 2 章以降では，HP C RTL 関数について詳しく説明します。

1.3.1 関数プロトタイプ

第 2 章以降のどの章でも，各関数を説明する構文は，関数を定義するための標準規約に従っています。この構文を**関数プロトタイプ** (または単に**プロトタイプ**) と呼びます。プロトタイプとは，関数の引数の順序 (引数がある場合)，引数の型，関数から返される値の型を簡略に表現したものです。プロトタイプを使用することを推奨します。

関数の戻り値を C データ型キーワードで簡単に表現できない場合は，説明文の中の戻り値の説明を参照してください。プロトタイプの記述では，関数の機能がわかりやすく示されています。しかし，これらの記述にはソース・コードで関数を呼び出す方法が示されていないことがあります。

たとえば，feof関数のプロトタイプについて考えてみましょう。

```
#include <stdio.h>
int feof(FILE *file_ptr);
```

この構文は次の情報を示します。

- feofプロトタイプは<stdio.h>ヘッダ・ファイルに存在します。feofを使用するには，このヘッダ・ファイルを取り込む必要があります (HP C RTL 関数を独自に宣言することは望ましくありません)。
- feof関数はint型の値を返します。
- この関数には*file_ptr*という 1 つの引数があり，データ型は "pointer to FILE" (FILE を指すポインタ) です。 FILE は<stdio.h>ヘッダ・ファイルに定義されています。

プログラムでfeofを使用するには，次の例に示すように，feof関数を呼び出す前に，任意の場所で<stdio.h>を取り込みます。

```
#include <stdio.h>                /* Include Standard I/O     */

main()
{
    FILE *infile;                 /* Define a file pointer    */
       .
       .
       .                          /* Call the function feof   */
    while ( ! feof(infile) )      /* Until EOF reached        */
       {                          /* Perform file operations  */
          .
          .
          .
       }
}
```

1.3.2 関数プロトタイプの構文規則

ライブラリに登録されている関数が使用するパラメータの数は同じではないため，関数プロトタイプの構文の記述は次の規則に従っています。

- パラメータの数が可変である場合，反復記号 (. . .) を使用してそのことを示します。
- パラメータの型が可変である場合，構文に型を示しません。

printfの構文記述について考えてみましょう。

```
#include <stdio.h>
int printf(const char *format_specification, . . . );
```

printfの構文記述には，1つ以上の省略可能なパラメータを指定できることが示されています。printfのパラメータに関する残りの情報は，関数の説明に示されています。

1.3.3 UNIX 形式のファイル指定

HP C RTL の関数やマクロでは，ファイルを操作することがよくあります。移植性に関する重大な問題の1つとして，各システムで使用されるファイル指定が異なるという問題があります。多くのC アプリケーションはUNIX システムとの間で移植されるため，すべてのコンパイラがUNIX システム・ファイル指定を読み込み，解釈できると便利です。

C プログラムをUNIX システムからOpenVMS システムに移植するのに役立つように，次のファイル指定変換関数がHP C RTL に用意されています。

- decc$match_wild
- decc$translate_vms
- decc$fix_time
- decc$to_vms
- decc$from_vms

これらのファイル指定変換関数をHP C RTL に取り込むと，UNIX システム・ファイル指定を含むC プログラムを書き直す必要がなくなるという利点があります。HP Cは大部分の有効なUNIX システム・ファイル指定をOpenVMS ファイル指定に変換できます。

UNIX システムとOpenVMS システムのファイル指定の違いだけでなく，RTL がファイルにアクセスする方法の違いについても注意してください。たとえば，RTL は有効なOpenVMS ファイル指定と有効な大部分のUNIX ファイル指定をそれぞれ受け付けますが，この2つの組み合わせを受け付けることはできません。表 1–1 は，UNIX システムとOpenVMS システムのファイル指定の区切り文字の違いを示しています。

表 1–1 UNIX と OpenVMS のファイル指定の区切り文字

説明	OpenVMS システム	UNIX システム
ノード区切り文字	::	!/
デバイス区切り文字	:	/
ディレクトリ・パス区切り文字	[ ]	/
サブディレクトリ区切り文字	[.]	/

(次ページに続く)

表 1–1 (続き) UNIX と OpenVMS のファイル指定の区切り文字

説明	OpenVMS システム	UNIX システム
ファイル拡張区切り文字	.	.
ファイル・バージョン区切り文字	;	適用されない

たとえば，表 1–2 は， 2 つの有効な指定と 1 つの無効な指定の形式を示しています。

表 1–2 有効および無効な UNIX と OpenVMS のファイル指定

システム	ファイル指定	有効/無効
OpenVMS	BEATLE::DBA0:[MCCARTNEY]SONGS.LIS	有効
UNIX	beatle!/usr1/mccartney/songs.lis	有効
—	BEATLE::DBA0:[MCCARTNEY.C]/songs.lis	無効

HP Cがファイル指定を変換する場合， OpenVMS システムと UNIX システムの両方のファイル指定を検索します。したがって，HP Cが UNIX システムのファイル指定を変換する方法と， UNIX システムが同じ UNIX ファイル指定を変換する方法には違いがあることがあります。

たとえば，表 1–2 に示すように 2 種類のファイル指定の方法を組み合わせた場合，HP C RTL は[MCCARTNEY.C]/songs.lis を[MCCARTNEY]songs.lis または[C]songs.lis として解釈する可能性があります。したがって，HP Cが複合ファイル指定を検出すると，エラーが発生します。

UNIX システムでは，デバイス名，ディレクトリ名，ファイル名に対して同じ区切り文字を使用します。このように UNIX ファイル指定にはあいまいさがあるため，HP Cはユーザの期待どおりに有効な UNIX システムのファイル指定を変換できないことがあります。

たとえば，/bin/todayを OpenVMS システムの形式で表現すると，[BIN]TODAY または[BIN.TODAY]になります。HP Cは存在するファイルからだけ正しい解釈を行うことができます。ファイル指定で 1 つのファイルまたはディレクトリに対して UNIX システムのファイル名構文が使用されている場合，次の条件のいずれかが満たされれば，対応する OpenVMS のファイル名に変換されます。

- 指定が既存の OpenVMS ディレクトリに対応する場合は，そのディレクトリ名に変換されます。たとえば， DEV:[DIR.SUB]が存在する場合は，/dev/dir/subは DEV:[DIR.SUB]に変換されます。
- 指定が既存の OpenVMS ファイル名に対応する場合は，そのファイル名に変換されます。たとえば， DEV:[DIR]FILE が存在する場合は，/dev/dir/fileは DEV:[DIR]FILE に変換されます。

- 指定が存在しない OpenVMS ファイル名に対応し，指定されたデバイスおよびディレクトリが存在する場合は，ファイル名に変換されます。たとえば，DEV:[DIR]が存在する場合は，/dev/dir/fileは DEV:[DIR]FILE に変換されます。

注意

OpenVMS Version 7.3 以降，HP C RTL に対して，UNIX 形式のファイル指定の先頭の部分をサブディレクトリ名またはデバイス名として解釈することを要求できるようになりました。

以前のリリースと同様に，foo/bar **(UNIX 形式の名前) のデフォルト変換は** FOO:BAR **(OpenVMS 形式のデバイス名) です。**

foo/bar **(UNIX 形式の名前) を**[.FOO]BAR **(OpenVMS 形式のサブディレクトリ名) に変換するように要求するには，論理名** DECC$DISABLE_TO_VMS_LOGNAME_TRANSLATION **に** ENABLE **を定義します。** DECC$DISABLE_TO_VMS_LOGNAME_TRANSLATION **はファイルごとにチェックされるのではなく，イメージを起動するときに** 1 **回だけチェックされます。この論理名の定義は，**decc$to_vms**関数に影響するだけでなく，**UNIX **形式と** OpenVMS **形式の両方のファイル名を引数として受け付けるすべての** HP C RTL **関数に影響します。**

UNIX システム環境では，数値のファイル記述子を使用してファイルを参照します。ファイル記述子の中には，標準 I/O デバイスを参照するものと，実際のファイルを参照するものがあります。ファイル記述子がオープンされていないファイルに属す場合は，HP C RTL はそのファイルをオープンします。HP Cは次の OpenVMS 論理名を使用してファイル記述子を評価します。

ファイル記述子	OpenVMS の論理名	意味
0	SYS$INPUT	標準入力
1	SYS$OUTPUT	標準出力
2	SYS$ERROR	標準エラー

1.3.4 拡張ファイル指定

ODS-5 ボリューム構造では， UNIX 形式と OpenVMS 形式が混在したファイル名のサポート機能が拡張されました。長いファイル名がサポートされるようになり，ファイル名でこれまでより広範囲にわたる文字を使用できるようになり，ファイル名で大文字と小文字の区別が保持されるようになりました。 OpenVMS Alpha Version 7.3-1 では， C RTL は ODS-5 文字のサポートを大幅に向上しており，これまでは 214 文字しかサポートされなかったのに対し， 256 文字中， 250 文字をサポートするようになりました。また，ファイル・タイプのないファイル名にもアクセスできるようになりました。

新しいサポートを有効にするには，1つ以上のC RTL機能論理名を定義する必要があります。これらの名前は次のとおりです。

DECC$EFS_CHARSET
DECC$DISABLE_TO_VMS_LOGNAME_TRANSLATION
DECC$FILENAME_UNIX_NO_VERSION
DECC$FILENAME_UNIX_REPORT
DECC$READDIR_DROPDOTNOTYPE
DECC$RENAME_NO_INHERIT

これらの機能論理名や他の機能論理名の詳細については，第 1.5 節を参照してください。

1.3.5 シンボリック・リンクと POSIX パス名

OpenVMS では，Open Group 準拠のシンボリック・リンクと，POSIX パス名の処理をサポートしています。詳細は，第 12 章を参照してください。

1.4 ヘッダ・ファイル制御のための機能テスト・マクロ

機能テスト・マクロは，移植可能なプログラムを作成するための手段を提供します。これらのマクロを使用すると，プログラムで使用するHP C RTL シンボル名が実装で提供されるシンボル名と競合しないかどうか確認できます。

HP C RTL のヘッダ・ファイルは，多くの機能テスト・マクロの使用をサポートするようにコーディングされています。アプリケーションで機能テスト・マクロを定義した場合，HP C RTL ヘッダ・ファイルはその機能テスト・マクロで定義されているシンボルとプロトタイプだけを提供し，それ以外のものは提供しません。プログラムでこのようなマクロを定義しないと，HP C RTL ヘッダ・ファイルは無制限にシンボルを定義します。

HP C RTL でサポートされる機能テスト・マクロは，次に示すようにヘッダ・ファイル内のシンボルの有効性を制御するために，大きく次のように分類されます。

- 標準
- 複数バージョンのサポート
- 互換性

1.4.1 標準マクロ

HP C RTL では，次の標準の一部が実装されています。

- XPG4 V2 (X/Open CAE Specification, System Interfaces and Headers, Issue 4, Version 2)
- XPG4 (X/Open CAE Specification, System Interfaces and Headers, Issue 4)
- POSIX 1003.1c-1995 またはIEEE 1003.1c-1995 (Standard for Information Technology - Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API)— Amendment 2:Threads Extension [C Language])
- ISO POSIX-2 (ISO/IEC 9945-2:1993 - Information Technology - Portable Operating System Interface (POSIX) - Part 2: Shell and Utilities)
- ISO POSIX-1 (ISO/IEC 9945-1:1990 - Information Technology - Portable Operating System Interface (POSIX) - Part 1: System Application Programming Interface (API) (C Language))
- ANSI/ISO/IEC 9899:1999—ISO が 1999 年 12 月に公開し， ANSI 標準で 2000 年 4 月に採用された C99 標準
- ISO C, Amendment 1 (ISO/IEC 9899:1990-1994 - Programming Languages - C, Amendment 1: Integrity)
- ISO C (ISO/IEC 9899:1990 - Programming Languages - C)— 標準部分はANSI C (X3.159-1989, American National Standard for Information Systems - Programming Language C) と同じ

1.4.2 標準の選択

機能テスト・マクロを定義することにより，各標準を選択することができます。このようにマクロを定義するには，ヘッダ・ファイルを取り込む前に， C ソースで`#define`プリプロセッサ・ディレクティブを使用するか， CC コマンド・ラインに/DEFINE 修飾子を指定します。

表 1–3 は，標準のサポートを制御するHP C RTL 機能テスト・マクロを示しています。

表 1–3 機能テスト・マクロ — 標準

マクロ名	選択される標準	暗黙に選択される他の標準	説明
_XOPEN_SOURCE_EXTENDED	XPG4 V2	XPG4, ISO POSIX-2, ISO POSIX-1, ANSI C	以前は X/Open で採用されていなかった従来の UNIX ベースのインタフェースも含めて, XPG4 拡張機能が有効になる。
_XOPEN_SOURCE	XPG4 (X/Open Issue 4)	ISO POSIX-2, ISO POSIX-1, ANSI C	XPG4 標準シンボルが有効になり, _POSIX_C_SOURCE が 2 より大きい値に定義されていない場合は, 2 に設定される[1] [2]。
_XOPEN_SOURCE=500	X/Open Issue 5	ISO POSIX-2, ISO POSIX-1, ANSI C	X/Open Issue 5 標準シンボルが有効になり, _POSIX_C_SOURCE が 2 より大きい値に定義されていない場合は, 2 に設定される[1] [2]。
_XOPEN_SOURCE=600	X/Open Issue 6	ISO POSIX-2, ISO POSIX-1, ANSI C	X/Open Issue 6 標準シンボルが有効になり, _POSIX_C_SOURCE が 2 より大きい値に定義されていない場合は, 2 に設定される[1] [2]。
_POSIX_C_SOURCE= =199506	IEEE 1003.1c-1995	ISO POSIX-2, ISO POSIX-1, ANSI C	ANSI C によって定義されるヘッダ・ファイルで, IEEE 1003.1c-1995 によって要求されるシンボルが有効になる。
_POSIX_C_SOURCE= =2	ISO POSIX-2	ISO POSIX-1, ANSI C	ANSI C によって定義されるヘッダ・ファイルで, ISO POSIX-2によって要求されるシンボルの他に, ISO POSIX-1によって要求されるシンボルも有効になる。
_POSIX_C_SOURCE= =1	ISO POSIX-1	ANSI C	ANSI Cによって定義されるヘッダ・ファイルで, ISO POSIX-1によって要求されるシンボルが有効になる。
__STDC_VERSION__= =199409	ISO C amdt 1	ANSI C	ISO C Amendment 1 シンボルが有効になる。
_ANSI_C_SOURCE	ANSI C	—	ANSI C 標準のシンボルが有効になる。

[1]ISO C Amendment 1には, XPG4 に指定されていないシンボルが含まれており, __STDC_VERSION__ = = 199409 および_XOPEN_SOURCE (または_XOPEN_SOURCE_EXTENDED) を定義すると, ISO Cと XPG4 API の両方が選択される。XPG4 とISO C Amendment 1の両方を指定してコンパイルしたときに競合が発生した場合, ISO C Amendment 1が優先される。

[2]XPG4 はISO C Amendment 1を拡張したものである。_XOPEN_SOURCE または_XOPEN_SOURCE_EXTENDED を定義すると, ISO C API の他に, ヘッダ・ファイルで提供される XPG4 拡張機能も選択される。このコンパイル・モードでは, XPG4 拡張機能が有効になる。

ここに示した標準のいずれでも定義されていない機能はHP C拡張機能であると解釈され, 標準関連の機能テスト・マクロを定義しないことによって選択されます。

ヘッダ・ファイル定義を制御するために機能テスト・マクロを明示的に定義しなかった場合は, HP C拡張機能も含めて, 定義されているすべてのシンボルが暗黙に取り込まれます。

1.4.3 /STANDARD 修飾子との相互関係

/STANDARD 修飾子はサポートされる C 言語の方言を選択します。

/STANDARD=ANSI89 および/STANDARD=ISOC94 を除き， C 言語の方言の選択と使用するHP C RTL API の選択はそれぞれ独立して行われます。/STANDARD に対して他の値を選択すると，拡張機能も含めて API 全体が使用可能になります。

/STANDARD=ANSI89 を指定すると，デフォルト API セットはANSI Cセットに制限されます。この場合，より広範囲にわたる API セットを選択するには，適切な機能テスト・マクロも指定する必要があります。拡張機能も含めて，ANSI Cの方言とすべての API を選択するには，ヘッダ・ファイルを取り込む前に，__HIDE_FORBIDDEN_NAMES の定義を取り消します。

/STANDARD=ISOC94 を使用してコンパイルすると，__STDC_VERSION__は 199409 に設定されます。 XPG4 とISO C Amendment 1の両方を指定してコンパイルしたときに競合が発生した場合は，ISO C Amendment 1が優先されます。ISO C Amendment 1に対する XPG4 拡張機能を選択するには，_XOPEN_SOURCE を定義します。

次の例はこれらの規則を理解するのに役立ちます。

- fdopen関数は，<stdio.h>に対する ISO POSIX-1の拡張機能です。したがって，<stdio.h>は，次の 1 つ以上の条件が満たされる場合だけ，fdopenを定義します。
  - この関数を含むプログラムが厳密なANSI Cモード(/STANDARD=ANSI89)でコンパイルされていない。
  - _POSIX_C_SOURCE が 1 以上の値として定義されている。
  - _XOPEN_SOURCE が定義されている。
  - _XOPEN_SOURCE_EXTENDED が定義されている。
- popen関数は，<stdio.h>に対する ISO POSIX-2の拡張機能です。したがって，次の 1 つ以上の条件が満たされる場合だけ，<stdio.h>はpopenを定義します。
  - この関数を含むプログラムが厳密なANSI Cモード(/STANDARD=ANSI89)でコンパイルされていない。
  - _POSIX_C_SOURCE が 2 以上の値として定義されている。
  - _XOPEN_SOURCE が定義されている。
  - _XOPEN_SOURCE_EXTENDED が定義されている。
- getw関数は，<stdio.h>に対するX/Openの拡張機能です。したがって，次の条件が 1 つ以上満たされる場合だけ，<stdio.h>はgetwを定義します。
  - プログラムが厳密なANSI Cモード(/STANDARD=ANSI89)でコンパイルされていない。

  - _XOPEN_SOURCE が定義されている。
  - _XOPEN_SOURCE_EXTENDED が定義されている。

- sysconf関数をサポートするために，X/Open の拡張シンボル定数_SC_PAGESIZE，_SC_PAGE_SIZE，_SC_ATEXIT_MAX，_SC_IOV_MAX が<unistd.h>に追加されました。しかし，これらの定数は_POSIX_C_SOURCE では定義されていません。

  プログラムで_POSIX_C_SOURCEを定義しておらず，_XOPEN_SOURCE_EXTENDEDを定義している場合にだけ，<unistd.h>ヘッダ・ファイルはこれらの定数を定義します。

  _POSIX_C_SOURCE が定義されている場合は，これらの定数は<unistd.h>で有効になりません。_POSIX_C_SOURCE は，厳密なANSI Cモードでコンパイルされたプログラムに対してだけ定義されます。

- fgetname関数は，<stdio.h>に対するHP C RTL の拡張機能です。したがって，プログラムが厳密なANSI Cモード (/STANDARD=ANSI89) でコンパイルされていない場合にだけ，<stdio.h>はfgetnameを定義します。

- マクロ_PTHREAD_KEYS_MAX は POSIX 1003.1c-1995で定義されています。このマクロは，_POSIX_C_SOURCE == 199506 と定義した状態で，この標準に対してコンパイルするか，標準を定義する機能テスト・マクロを指定せずにコンパイルしたときにデフォルトによって，<limits.h>で有効になります。

- <wchar.h>に定義されているマクロ WCHAR_MAX は，ISO C Amendment 1では必要ですが，XPG4 では必要ありません。したがって次のようになります。

  - ISO C Amendment 1に準拠してコンパイルすると，このシンボルは有効になりますが，XPG4 に準拠してコンパイルしても有効になりません。
  - ISO C Amendment 1と XPG4 の両方に準拠してコンパイルすると，このシンボルは有効になります。

  同様に，<wchar.h>内の関数wcsftimeとwcstokの定義は，ISO C Amendment 1と XPG4 で少し異なります。

  - ISO C Amendment 1に準拠するようにコンパイルすると，ISO C Amendment 1プロトタイプが有効になります。
  - XPG4 に準拠するようにコンパイルすると，XPG4 プロトタイプが有効になります。
  - ISO C Amendment 1と XPG4 の両方に準拠するようにコンパイルすると，このコンパイル・モードで発生した競合は ISO C を優先して解決されるため，ISO Cプロトタイプが選択されます。
  - 標準を選択する機能テスト・マクロを指定せずにコンパイルした場合は，ISO C Amendment 1の機能が有効になります。

  この例では，標準を選択する機能テスト・マクロを指定せずにコンパイルすると，wcsftimeとwcstokに対して WCHAR_MAX およびISO C Amendment 1プロトタイプが有効になります。

- wcswidth関数とwcwidth関数は，ISO C Amendment 1に対する XPG4 の拡張機能です。これらのプロトタイプは<wchar.h>にあります。

  これらのシンボルは，次の場合に有効になります。

  - _XOPEN_SOURCE または_XOPEN_SOURCE_EXTENDED を定義することにより， XPG4 に準拠するようにコンパイルした場合。
  - DEC C Version 4.0との互換性を維持してコンパイルするか，または OpenVMS Version 7.0 より前のシステムでコンパイルした場合。
  - 標準を選択する機能テスト・マクロを指定せずにコンパイルした場合。
  - ISO C Amendment 1と XPG4 の両方に準拠するようにコンパイルした場合 (これらのシンボルはISO C Amendment 1に対する XPG4 の拡張機能であるため)。

  厳密なISO C Amendment 1モードでコンパイルしても，これらは有効になりません。

1.4.4 複数バージョン・サポート・マクロ

デフォルト設定では，ヘッダ・ファイルは，コンパイルが行われるオペレーティング・システムのバージョンによって提供されるHP C RTL 内の API を有効にします。この処理は，『HP C User's Guide for OpenVMS Systems』に説明しているように，__VMS_VER マクロの定義済み設定によって行われます。たとえば，OpenVMS Version 6.2 でコンパイルすると， V6.2 およびそれより前のバージョンのHP C RTL API だけが使用可能になります。

__VMS_VER マクロの別の使用例として， OpenVMS Alpha Version 7.0 以降で提供されるHP C RTL 関数の64 ビット・バージョンのサポートがあります。すべてのヘッダ・ファイルで，64 ビットをサポートする関数は，__VMS_VER が OpenVMS Version 7.0 以降であることを示す場合にだけ有効になるように条件化されています。

オペレーティング・システムの以前のバージョンを対象にするには，次の操作を行います。

1. DECC$SHR の古いバージョンを指すように論理名 DECC$SHR を定義します。コンパイラは DECC$SHR のテーブルを使用してルーチン名に接頭語を付ける処理を実行します。
2. /DEFINE 修飾子を使用するか，#undefと#defineプリプロセッサ・ディレクティブの組み合わせを使用して，__VMS_VER を適切に定義します。/DEFINE 修飾子を使用する場合は，定義済みマクロの再定義に関する警告を無効にする必要があるかもしれません。

オペレーティング・システムの新しいバージョンを対象にする操作は，常に可能なわけではありません。一部のバージョンでは，まだ存在しないオペレーティング・システムの新機能が新しい DECC$SHR.EXE で要求されることがあります。このようなバージョンでは，ステップ 1 で論理名 DECC$SHR を定義すると，コンパイル・エラーになります。

__VMS_VER の値を上書きするには，コンパイラのコマンド・ラインで__VMS_VER_OVERRIDE を定義します。値を指定せずに__VMS_VER_OVERRIDE を定義すると，__VMS_VER は最大値に設定されます。

1.4.5 互換性モード

次の定義済みマクロは，DEC Cの以前のバージョンまたは OpenVMS オペレーティング・システムの以前のバージョンとのヘッダ・ファイルの互換性を選択するために使用します。

- _DECC_V4_SOURCE
- _VMS_V6_SOURCE

ヘッダ・ファイルでは，次の 2 種類の非互換性を制御できます。

- 標準に準拠するために行う一部の変更では，ソース・コード・レベルの互換性が維持されませんが，バイナリ・レベルの互換性は維持されます。DEC C Version 4.0 のソース互換性を選択するには，_DECC_V4_SOURCE マクロを使用します。
- 標準に準拠するためのその他の変更では，バイナリまたは実行時の非互換性が発生します。

  一般に，プログラムを再コンパイルすると，新しい動作が実行されるようになります。このような場合は，_VMS_V6_SOURCE 機能テスト・マクロを使用して，以前の動作を保持します。

  しかし，exit，kill，wait関数の場合，これらのルーチンを ISO POSIX-1準拠にするための OpenVMS Version 7.0 での変更は，デフォルトにするにはあまりに互換性がないと解釈されました。したがって，これらの場合は，デフォルトの動作は OpenVMS Version 7.0 より前のバージョンのシステムと同じです。 ISO POSIX-1に準拠したこれらのルーチンのバージョンにアクセスするには，_POSIX_EXIT 機能テスト・マクロを使用します。

次の例はこれらのマクロの使い方を理解するのに役立ちます。

- ISO POSIX-1 標準に準拠するために，次のものに対するtypedefが<types.h>に追加されました。

```
dev_t         off_t
gid_t         pid_t
ino_t         size_t
mode_t        ssize_t
nlink_t       uid_t
```

DEC C Version 5.2 より前のバージョンを使用する古い開発環境では，<types.h>にこれらのtypedefがないため，別のモジュールにこれらの定義を追加する必要があるかもしれません。このような場合は，DEC C Version 5.2 で提供される<types.h>を使用してコンパイルすると，コンパイル・エラーが発生する可能性があります。

現在の環境を維持し，DEC C Version 5.2 の<types.h>を取り込むには，_DECC_V4_SOURCE を定義してコンパイルします。このようにすると，DEC C Version 5.2 のヘッダから互換性のない参照が排除されます。たとえば，<types.h>では，前に示したtypedefsは有効になりません。

- OpenVMS Version 7.0 では，HP C RTL のgetuid関数とgeteuid関数は， UIC のグループの部分とメンバの部分の両方を含む OpenVMS UIC (ユーザ識別コード) を返すように定義されています。DEC C RTL の以前のバージョンでは，これらの関数は UIC コードのメンバ番号だけを返していました。

  <unistd.h> (ISO POSIX-1 標準で要求) および<unixlib.h> (HP C RTL 互換性の場合) 内のgetuidとgeteuidのプロトタイプは変更されていません。デフォルト設定では，getuidとgeteuidを呼び出すプログラムを新たにコンパイルすると，新しい定義が使用されます。つまり，これらの関数は OpenVMS の UIC を返します。

  プログラムでgetuidとgeteuidに関して， OpenVMS Version 7.0 より前のバージョンの動作を保持したい場合は，_VMS_V6_SOURCE 機能テスト・マクロを定義してコンパイルします。

- OpenVMS Version 7.0 では，HP C RTL のexit関数は ISO POSIX-1のセマンティックで定義されています。この結果，exitに対する入力状態引数は 0 ～ 255 の値になります (これより前のバージョンでは，exitは状態パラメータで OpenVMS の条件コードを受け付けることができました)。

  デフォルトでは， OpenVMS システムのexit関数の動作は以前と同じです。つまり，exitは OpenVMS の条件コードを受け付けます。 ISO POSIX-1と互換性のあるexit関数を有効にするには，_POSIX_EXIT 機能テスト・マクロを定義してコンパイルします。

1.4.6 Curses およびソケット互換性マクロ

次の機能テスト・マクロは，HP C RTL ライブラリの Curses およびソケット・サブセットを制御するために使用します。

- _BSD44_CURSES

  このマクロは，4.4BSD Berkeley Software Distribution から Curses パッケージを選択します。

- _VMS_CURSES

  このマクロは，VAX Cコンパイラをベースにした Curses パッケージを選択します。これはデフォルトの Curses パッケージです。

- _SOCKADDR_LEN

  このマクロは， 4.4BSD および XPG4 V2 準拠のソケット・インタフェースを選択するために使用します。これらのインタフェースでは，基礎になるTCP/IPソフトウェアでのサポートが必要です。稼動するTCP/IPソフトウェアのバージョンで4.4BSDソケットがサポートされるかどうかについては，TCP/IPのベンダにお問い合わせください。

XPG4 V2 に厳密に準拠するには， 4.4BSD と互換性のあるソケット・インタフェースが必要です。したがって， OpenVMS Version 7.0 以降で_XOPEN_SOURCE_EXTENDED が定義されている場合，_SOCKADDR_LEN は 1 に定義されます。

次の例はこれらのマクロの使い方を理解するのに役立ちます。

- AE， AL， AS， AM， BC など， BSD Curses パッケージで使用される termcap フィールドを指すポインタを表すシンボル定数は，_BSD44_CURSES が定義されている場合，<curses.h>でのみ有効になります。
- <socket.h>ヘッダ・ファイルは，_SOCKADDR_LEN または_XOPEN_SOURCE_EXTENDED が定義されている場合だけ， 4.4BSD sockaddr構造体を定義します。これらが定義されていない場合は，<socket.h>は 4.4BSD より前のsockaddr構造体を定義します。_SOCKADDR_LEN が定義され，_XOPEN_SOURCE_EXTENDED が定義されていない場合は，<socket.h>ヘッダ・ファイルはosockaddr構造体も定義します。この構造体は互換性を維持するために使用される 4.3BSD のsockaddr構造体です。 XPG4 V2 ではosockaddr構造体を定義していないため，_XOPEN_SOURCE_EXTENDED モードでは有効になりません。

1.4.7 2G バイトのファイル・サイズ・マクロ

C RTL では， 2G バイト(GB)以上のファイル・サイズおよびオフセットを使用するアプリケーションをコンパイルする機能がサポートされるようになりました。この機能は， 64 ビット整数のファイル・オフセットを可能にすることで実現されています。

fseeko関数とftello関数は，fseek関数およびftell関数と同じ動作を実行し，off_t型の値を受け付けるか，または返します。これにより，off_tの 64 ビット・バージョンを使用することができます。

C RTL 関数lseek，mmap，ftuncate，truncate，stat，fstat，ftwも 64 ビット・ファイル・オフセットに対応できます。

新しい 64 ビット・インタフェースをコンパイル時に選択するには，_LARGEFILE 機能マクロを定義します。

1.4.8 32 ビット UID および GID マクロ*(Integrity，Alpha)*

C RTL では，32 ビットのユーザ識別 (UID) とグループ識別 (GID) がサポートされます。32 ビットの UID/GID を使用するように設定してアプリケーションをコンパイルすると，UID と GID はオペレーティング・システムの以前のバージョンと同様に UIC から生成されます。

デフォルトで 32 ビットの UID/GID を使用するシステムで，16 ビットの UID/GID をサポートするようにアプリケーションをコンパイルするには，_DECC_SHORT_GID_T マクロに 1 を定義します。

1.4.9 標準準拠の stat 構造体*(Integrity，Alpha)*

C RTL は，X/Open 標準に準拠したstat構造体の定義および関連する定義をサポートしています。これらの新しい定義を使用するには，機能テスト・マクロ_USE_STD_STAT を使用してアプリケーションをコンパイルする必要があります。

_USE_STD_STAT を使用してコンパイルすると，stat構造体が以下のように変更されます。

- 型ino_tがunsigned quadword intとして定義されます。_USE_STD_STAT を使用しない場合は，unsigned shortとして定義されます。
- 型dev_tが 64 ビット整数として定義されます。_USE_STD_STAT を使用しない場合は，32 ビット文字ポインタとして定義されます。
- 型off_tが，_LARGEFILE マクロが定義されているかのように，64 ビット整数として定義されます。_USE_STD_STAT を使用しない場合は，32 ビット整数として定義されます。
- フィールドst_devおよびst_rdevが，デバイスごとに違った値となります。_USE_STD_STAT を使用しない場合は，一意性は保証されません。
- フィールドst_blksizeおよびst_blocksが追加されます。_USE_STD_STAT を使用しない場合は，これらのフィールドは存在しません。

1.4.10 _toupper と_tolower の従来の動作での使用*(Integrity，Alpha)*

OpenVMS Version 8.3 の_tolowerマクロと_touppperマクロでは，特に指定しない限りパラメータを 2 回以上評価しないようにして，C99 ANSI 標準と X/Open 仕様に準拠させるようになっています。つまり，これらのマクロでは，対応するtolowerまたは toupper関数を単に呼び出すだけです。そのため，式を評価する回数をユーザが指定できる場合の副作用 (i++や関数呼び出しなど) が回避できるようになっています。

_tolowerマクロと_touppperマクロを以前の仕様の最適化された動作のままで使用する場合は，/DEFINE=_FAST_TOUPPER を指定してコンパイルしてください。そうすれば，これらのマクロは，以前のリリースと同じように，実行時の呼び出しオーバヘ

ッドを避けるように最適化されます。ただし，その場合はマクロのパラメータが 2 回以上評価されて計算方法が決定されるので，望ましくない副作用が発生することもあります。

1.4.11 高速なインライン put 関数および get 関数の使用*(Integrity， Alpha)*

__UNIX_PUTC マクロを定義してコンパイルすれば，次の I/O 関数で高速なインライン関数を使用するように最適化できます。

```
fgetc
fputc
putc
putchar
fgetc_unlocked
fputc_unlocked
putc_unlocked
putchar_unlocked
```

1.5 機能論理名の使用による C RTL 機能の有効化

C RTL では，多くの機能スイッチが提供されます。これらの機能スイッチは，DECC$論理名を使用して設定でき，実行時に C アプリケーションの動作に影響を与えます。

機能スイッチにより，新しい動作を導入したり，以前の動作をそのまま保持することもできます。

大部分の機能は，論理名を ENABLE に設定することで有効にし，論理名を DISABLE に設定することで無効にすることができます。

```
$ DEFINE DECC$feature ENABLE
$ DEFINE DECC$feature DISABLE
```

一部の機能論理名は数値に設定することができます。次の例を参照してください。

```
$ DEFINE DECC$PIPE_BUFFER_SIZE 32768
```

注意

- **システム用の** C RTL **機能論理名は設定しないでください。** OpenVMS **コンポーネントなどの他のアプリケーションが，これらの論理名のデフォルト動作に依存しているため，機能論理名の設定を必要としているアプリケーション用の機能論理名だけを設定してください。**

- **C ランタイム・ライブラリの以前のリリースで提供されていた古い機能論理名では，任意の等価文字列を設定することで機能が有効になると説明されていました。この説明は以前は間違いではなかったのですが，今後は，有効にするには機能論理名に** "ENABLE"**を設定し，無効にするには** "DISABLE"**を設定することを強くお勧めします。このようにしないと，予期しない結果になることがあります。**

  予期しない結果になる理由は次のとおりです。

  - **以前のバージョンの** C RTL **では，任意の等価文字列** (DISABLE **であっても) により，機能論理名が有効になっていました。**
  - **以降のバージョンおよび現在のバージョンの** C RTL **では，以下の等価文字列で機能論理名が無効になります。機能論理名を有効にするために，以下の文字列を使用することは避けてください。**

    DISABLE
    0 (**ゼロ**)
    F
    FALSE
    N
    NO

  ここにリストされていない文字列では，機能論理名が有効になります。たとえば，誤って綴り間違いの "DSABLE"**という文字列を設定しても，機能論理名が有効になります。**

表 1–4 は， C RTL 機能論理名を，制御する機能の種類別に示しています。

表 1–4　C RTL **機能論理名**

機能論理名	**デフォルト**
性能の最適化	
DECC$ENABLE_GETENV_CACHE	DISABLE
DECC$LOCALE_CACHE_SIZE	0
DECC$TZ_CACHE_SIZE	2
従来の動作	
DECC$ALLOW_UNPRIVILEGED_NICE	DISABLE
DECC$NO_ROOTED_SEARCH_LISTS	DISABLE
DECC$PRINTF_USES_VAX_ROUND	DISABLE
DECC$THREAD_DATA_AST_SAFE	DISABLE
DECC$V62_RECORD_GENERATION	DISABLE
DECC$WRITE_SHORT_RECORDS	DISABLE
DECC$XPG4_STRPTIME	DISABLE

(**次ページに続く**)

表 1–4 (続き) C RTL 機能論理名

機能論理名	デフォルト
ファイル属性	
DECC$DEFAULT_LRL	32767
DECC$DEFAULT_UDF_RECORD	DISABLE
DECC$FIXED_LENGTH_SEEK_TO_EOF	DISABLE
DECC$ACL_ACCESS_CHECK	DISABLE
メールボックス	
DECC$MAILBOX_CTX_STM	DISABLE
UNIX 準拠のための変更	
DECC$SELECT_IGNORES_INVALID_FD	DISABLE
DECC$STRTOL_ERANGE	DISABLE
DECC$VALIDATE_SIGNAL_IN_KILL	DISABLE
一般的な UNIX 拡張機能	
DECC$UNIX_LEVEL	DISABLE
DECC$ARGV_PARSE_STYLE	DISABLE
DECC$PIPE_BUFFER_SIZE	512
DECC$PIPE_BUFFER_QUOTA	512
DECC$STREAM_PIPE	DISABLE
DECC$POPEN_NO_CRLF_REC_ATTR	DISABLE
DECC$STDIO_CTX_EOL	DISABLE
DECC$USE_RAB64	DISABLE
DECC$GLOB_UNIX_STYLE	DISABLE
UNIX 形式のファイル名をサポートするための拡張	
DECC$DISABLE_TO_VMS_LOGNAME_TRANSLATION	DISABLE
DECC$EFS_CHARSET	DISABLE
DECC$ENABLE_TO_VMS_LOGNAME_CACHE	ENABLE
DECC$FILENAME_ENCODING_UTF8	DISABLE
DECC$FILENAME_UNIX_NO_VERSION	DISABLE
DECC$FILENAME_UNIX_REPORT	DISABLE
DECC$READDIR_DROPDOTNOTYPE	DISABLE
DECC$RENAME_NO_INHERIT	DISABLE
DECC$RENAME_ALLOW_DIR	DISABLE

(次ページに続く)

表 1–4 (続き) C RTL 機能論理名

機能論理名	デフォルト
UNIX 形式のファイル属性をサポートするための拡張	
DECC$EFS_FILE_TIMESTAMPS	DISABLE
DECC$EXEC_FILEATTR_INHERITANCE	DISABLE
DECC$FILE_OWNER_UNIX	DISABLE
DECC$FILE_PERMISSION_UNIX	DISABLE
DECC$FILE_SHARING	DISABLE
UNIX 準拠モード	
DECC$DETACHED_CHILD_PROCESS	DISABLE
DECC$FILENAME_UNIX_ONLY	DISABLE
DECC$POSIX_STYLE_UID	DISABLE
DECCUSE_JPI_CREATOR	DISABLE
POSIX 準拠のための新しい動作	
DECC$ALLOW_REMOVE_OPEN_FILES	DISABLE
DECC$POSIX_SEEK_STREAM_FILE	DISABLE
DECC$UMASK	RMS のデフォルト
ファイル名の処理	
DECC$POSIX_COMPLIANT_PATHNAMES	DISABLE
DECC$DISABLE_POSIX_ROOT	ENABLE
DECC$EFS_CASE_PRESERVE	DISABLE
DECC$EFS_CASE_SPECIAL	DISABLE
DECC$EFS_NO_DOTS_IN_DIRNAME	DISABLE
DECC$READDIR_KEEPDOTDIR	DISABLE
DECC$UNIX_PATH_BEFORE_LOGNAME	DISABLE

この後， C RTL 機能論理名をアルファベット順に示し，その説明も示します。特に示した場合を除き，機能論理名は ENABLE で有効になり， DISABLE で無効になります。

DECC$ACL_ACCESS_CHECK
DECC$ACL_ACCESS_CHECK 機能論理名はaccess関数の動作を制御します。

DECC$ACL_ACCESS_CHECK が有効な場合，access関数は UIC 保護と OpenVMS アクセス制御リスト (ACL) の両方を確認します。

DECC$ACL_ACCESS_CHECK が無効な場合，access関数は UIC 保護のみを確認します。

DECC$ALLOW_REMOVE_OPEN_FILES
DECC$ALLOW_REMOVE_OPEN_FILES 機能論理名は，オープン・ファイルに対するremove関数の動作を制御します。一般的に，この操作は失敗します。ただし，POSIX の準拠条件では，この操作は成功することになっています。

DECC$ALLOW_REMOVE_OPEN_FILES が有効の場合，この POSIX 準拠の動作が行われます。

DECC$ALLOW_UNPRIVILEGED_NICE
DECC$ALLOW_UNPRIVILEGED_NICE が有効の場合，nice関数は，呼び出し元プロセスの特権をチェックしないという従来の動作を行います (つまり，任意のユーザが，nice値を小さくして，プロセスの優先順位を高くすることができます)。また，MAX_PRIORITY を超える優先順位を，呼び出し元が設定した場合，nice値にはベースの優先順位が設定されます。

DECC$ALLOW_UNPRIVILEGED_NICE が無効の場合，nice関数は X/Open 標準に準拠し，呼び出し元プロセスの特権をチェックします (つまり， ALTPRI 特権を持つユーザだけがnice値を小さくして，プロセスの優先順位を高くすることができます)。また， MAX_PRIORITY を超える優先順位を呼び出し元が設定した場合，nice値には MAX_PRIORITY が設定されます。

DECC$ARGV_PARSE_STYLE
DECC$ARGV_PARSE_STYLE を有効に設定すると，プロセスが SET PROCESS /PARSE_STYLE=EXTENDED を使用して拡張 DCL 解析用に設定されている場合，コマンド・ライン引数で大文字と小文字の区別が保持されます。

DECC$ARGV_PARSE_STYLE は論理名として外部で定義するか，または LIB$INITIALIZE 機能を使用して呼び出した関数の内部で設定する必要があります。これは，main関数が呼び出される前に，この論理名が評価されるからです。

DECC$DEFAULT_LRL
DECC$DEFAULT_LRL は，最長のレコード長に関する RMS 属性のデフォルト値を指定します。デフォルト値 32767 は RMS でサポートされる最大レコード・サイズです。

デフォルト値: 32767

最大値: 32767

DECC$DEFAULT_UDF_RECORD
DECC$DEFAULT_UDF_RECORD を有効に設定すると， STREAMLF を除き，他のすべてのファイルに対して，ファイル・アクセス・モードのデフォルトが STREAM モードではなく， RECORD モードになります。

DECC$DETACHED_CHILD_PROCESS
DECC$DETACHED_CHILD_PROCESS を有効に設定すると，vforkおよびexecを使用して生成される子プロセスは，サブプロセスではなく，独立プロセスとして生成されます。

この機能のサポートは限定されています。場合によっては，コンソールを親プロセスと独立プロセスの間で共用することができないため，execが異常終了することがあります。

DECC$DISABLE_POSIX_ROOT
DECC$DISABLE_POSIX_ROOT を有効に設定すると， SYS$POSIX_ROOT によって定義される POSIX ルート・ディレクトリのサポートが無効になります。

DECC$DISABLE_POSIX_ROOT を無効に設定すると， SYS$POSIX_ROOT 論理名はファイル・パス "/"と同じであると解釈されます。スラッシュ(/)で始まる UNIX パスが指定され，先頭のスラッシュの後の値を論理名として変換できない場合は，指定された UNIX のファイル・パスの親ディレクトリとして SYS$POSIX_ROOT が使用されます。

C RTL では，実際のディレクトリのように動作する UNIX 形式のルートがサポートされます。このため，次のような動作が可能です。

```
% cd /
% mkdir /dirname
% tar -xvf tarfile.tar /dirname
% ls /
```

以前は， C RTL は "/"をディレクトリ名として認識しませんでした。 "/"から始まるファイル・パスの通常の処理では，最初の要素は論理名またはデバイス名として解釈されていました。このような解釈ができないときは，/dev/nullという名前，および/binおよび/tmpから始まる名前に対して特殊な処理が行われていました。

```
/dev/null       NLA0:
/bin            SYS$SYSTEM:
/tmp            SYS$SCRATCH:
```

これらの動作は，互換性を維持するために保持されています。さらに，SYS$POSIX_ROOT という論理名を "/"に対応するものとして解釈するサポートが C RTL に追加されました。

C RTL で使用するためにこの機能を有効にするには， SYS$POSIX_ROOT を隠し論理名として定義します。次の例を参照してください。

```
$ DEFINE/TRANSLATION=(CONCEALED,TERMINAL) SYS$POSIX_ROOT "$1$DKA0:[SYS0.abc.]"
```

この機能を無効にするには，次のように指定します。

```
$ DEFINE DECC$DISABLE_POSIX_ROOT DISABLE
```

SYS$POSIX_ROOT を有効にすると，次の動作が実行されます。

- "/"から始まる UNIX パスの既存の変換が失敗し， SYS$POSIX_ROOT が定義されている場合は，名前は/sys$posix_rootから始まるかのように解釈されます。
- OpenVMS から UNIX 形式のファイル名に変換するときに， OpenVMS の名前が "SYS$POSIX_ROOT:"から始まる場合， "SYS$POSIX_ROOT:"は削除されます。たとえば， SYS$POSIX_ROOT:[dirname]は/dirnameになります。このようにして作成された名前が論理名または上記の特殊なケースのいずれかとして解釈可能な場合は，結果は/dirnameではなく，/./dirnameになります。

DECC$DISABLE_TO_VMS_LOGNAME_TRANSLATION
DECC$DISABLE_TO_VMS_LOGNAME_TRANSLATION を有効に設定すると，変換ルーチンdecc$to_vmsは， UNIX 形式の名前の先頭にスラッシュ(/)がある場合，最初の要素だけを論理名として取り扱います。

DECC$EFS_CASE_PRESERVE
DECC$EFS_CASE_PRESERVE を有効に設定すると， ODS-5 ディスクのファイル名で大文字と小文字の区別が保持されます。

DECC$EFS_CASE_PRESERVE を無効に設定すると， UNIX 形式のファイル名は常に小文字で報告されます。

しかし， DECC$EFS_CASE_SPECIAL を有効に設定すると， DECC$EFS_CASE_PRESERVE の設定は無効になります。

DECC$EFS_CASE_SPECIAL
DECC$EFS_CASE_SPECIAL を有効に設定すると，小文字を含むファイル名の場合にだけ，大文字と小文字の区別が保持されます。ファイル名の要素がすべて大文字の場合は， UNIX 形式ではすべて小文字で報告されます。

DECC$EFS_CASE_SPECIAL を有効に設定すると，この設定は DECC$EFS_CASE_PRESERVE の値より優先します。

DECC$EFS_CHARSET
DECC$EFS_CHARSET を有効に設定すると， UNIX の名前で ODS-5 拡張文字を使用できます。次の文字を除き，複数のドットと 0 ～ 255 の範囲のすべての ASCII 文字がサポートされます。

```
<NUL>)
/    *
"    ?
```

DECC$FILENAME_UNIX_ONLY が有効に設定されていない限り，一部の文字はコンテキストに応じて OpenVMS の文字として解釈されることがあります。このような文字は次のとおりです。

```
:   ^
[   ;
<
```

ファイル名に関して特定の文字が存在するものと仮定している既存のアプリケーションでは， EFS 拡張文字セットのサポートを有効に設定すると，非標準でドキュメントに記載されていない次のC RTL拡張機能が動作しないため， DECC$EFS_CHARSET が必要になることがあります。

- $HOME はユーザのログイン・ディレクトリとして解釈されます。

  DECC$EFS_CHARSET を有効に設定すると，$HOME はリテラルとして取り扱われるため， OpenVMS または UNIX 形式のファイル名でこの文字を使用できます。

- ~nameは，ユーザnameのログイン・ディレクトリとして解釈されます。

  DECC$EFS_CHARSET を有効に設定すると，~nameはリテラルとして取り扱われるため， OpenVMS または UNIX 形式のファイル名でこの文字を使用できます。

- [a-z]という形式のワイルド・カード正規表現。

  DECC$EFS_CHARSET を有効に設定すると， OpenVMS および UNIX 形式のファイル名で角括弧を使用できます。たとえば，openなどの関数では，abc[a-z]ef.txtは， OpenVMS 形式のabc^[a-z^]ef.txtという名前に等しい UNIX 形式の名前として解釈され，[a-z]bcは UNIX 形式の/sys$disk/a-z/bcという名前に等しい OpenVMS 形式の名前として解釈されます。

DECC$EFS_CHARSET を有効に設定すると， OpenVMS 形式のファイル名を UNIX 形式のファイル名に変換するときに， EFS 拡張文字の次のエンコーディングがサポートされます。

- すべての ODS-2 互換名。
- シングル・バイト文字または^abという形式の 2 桁の 16 進数を使用した 8 ビット文字のすべてのエンコード。 UNIX パスでは，これらは常にシングル・バイトとして表現されます。
- DEL のエンコード (^7F)。
- カレットの後に続く次の文字。

  space ! , _ & ' ( ) + @ { } ; # [ ] % ^ = $ - ~ .

- カレットが付いていない次の文字。

  $ - ~ .

- 実装では，関数readdir，ftw，getname，fgetname，getcwdなどに対して，OpenVMS から UNIX への必要な変換がサポートされます。

注意

C RTL ファイル名処理において，次のような特別なケースがあります。

- ^.dirで終わるパス名はディレクトリとして扱われ，これらの文字は変換時に文字列から切り捨てられます。
- ^/で始まるパス名は，次のトークンを論理名あるいはルートからのディレクトリ名として扱います。

以下のサンプル・プログラムはこれらについて示しています。

```
#include <stdio.h>
#include <dirent.h>
#include <unixlib.h>
#include <string.h>
main()
{
  char adir[80];
  DIR *dir;
  struct dirent *dp;
  int decc_feature_efs_charset_index = 0;
  int decc_feature_efs_charset_default_val = 0;

  if (
      ( (decc_feature_efs_charset_index =
        decc$feature_get_index("DECC$EFS_CHARSET")) == -1 )
      ||
      ( (decc_feature_efs_charset_default_val =
        decc$feature_get_value(decc_feature_efs_charset_index, 0)) == -1 )
      ||
      ( (decc$feature_set_value(decc_feature_efs_charset_index, 1, TRUE) == -1))
     )
    {
      printf("Error setting up DECC$EFS_CHARSET macro\n");
    }

  strcpy(adir, "SYS$SYSDEVICE:[SSHTEST.TEST.a^,test^.dir^;22]");
  printf("\n\nFor %s\n", adir);
mrb:  dir = opendir(adir);
  if(dir)
    {
      do
        {
          dp = readdir(dir);
          if(dp->d_name) printf("%s\n", dp->d_name);
        } while (dp);
    }

  closedir(dir);
```

```
  strcpy(adir, "SYS$SYSDEVICE:[SSHTEST.TEST.a^,test^.dir]");
  printf("\n\nFor %s\n", adir);
  dir = opendir(adir);
  if(dir)
    {
      do
        {
          dp = readdir(dir);
          if(dp->d_name) printf("%s\n", dp->d_name);
        } while (dp);
    }

  closedir(dir);

  strcpy(adir, "SYS$SYSDEVICE:[SSHTEST.TEST.a^\\test]");
  printf("\n\nFor %s\n", adir);
  dir = opendir(adir);
  if(dir)
    {
      do
        {
          dp = readdir(dir);
          if(dp->d_name) printf("%s\n", dp->d_name);
        } while (dp);
    }

  strcpy(adir, "SYS$SYSDEVICE:[SSHTEST.TEST.copies]");
  printf("\n\nFor %s\n", adir);
  dir = opendir(adir);
  if(dir)
    {
      do
        {
          dp = readdir(dir);
          if(dp->d_name) printf("%s\n", dp->d_name);
        } while (dp);
    }

  closedir(dir);

  strcpy(adir, "/SYS$SYSDEVICE/SSHTEST/TEST/copies");
  printf("\n\nFor %s\n", adir);
  dir = opendir(adir);
  if(dir)
    {
      do
        {
          dp = readdir(dir);
          if(dp->d_name) printf("%s\n", dp->d_name);
        } while (dp);
    }

  closedir(dir);

}
```

DECC$EFS_FILE_TIMESTAMPS
DECC$EFS_FILE_TIMESTAMPS を有効に設定すると，statとfstatは， SET VOLUME/VOLUME=ACCESS_DATES を使用して拡張ファイル時刻が有効に設定されている ODS-5 ボリュームのファイルに対して，新しいODS-5アクセス時刻 (st_atime)，属性改訂時刻 (st_ctime)，および変更時刻 (st_mtime) を報告します。

DECC$EFS_FILE_TIMESTAMPS が無効に設定されているか，ボリュームが ODS-5 でない場合，あるいは有効に設定されたこれらの追加時刻がボリュームでサポートされない場合，st_ctimeはファイルの作成時刻になり，st_atimeはst_mtimeと同じになります。

関数utimeとutimesはこれらの ODS-5 の時刻をstatと同じ方法でサポートします。

DECC$EFS_NO_DOTS_IN_DIRNAME
ODS-5 のファイル名での拡張文字のサポートにより， NAME.EXT などの名前は，NAME.EXT.DIR と解釈できます。 UNIX ファイル名の拡張文字サポートが有効の場合，[.name^.ext]というディレクトリが存在するかどうかを調べるために， UNIX 名の変換でオーバヘッドが発生します。

DECC$EFS_NO_DOTS_IN_DIRNAME 機能論理名を有効にすると，ドットを含むファイル名をディレクトリ名として解釈しなくなります。この論理名を有効にすると， NAME.EXT はファイル名と見なされ，ディレクトリ[.name^.ext]はチェックされません。

DECC$ENABLE_GETENV_CACHE
C RTL では，environテーブルにある環境変数の一覧の他に，プロセスで使用できるすべての論理名と DCL シンボルも使用されます。

デフォルトでは，environテーブルにない名前に対してgetenvが呼び出されると，この名前を論理名として解決しようとします。解決できない場合は， DCL シンボルとして解決しようとします。

DECC$ENABLE_GETENV_CACHE を有効に設定すると，論理名または DCL 名が正しく変換された後，その値はキャッシュに格納されます。同じ名前がgetenvの将来の呼び出しで要求された場合は，論理名や DCL シンボルを再評価するのではなく，キャッシュにある値が返されます。

DECC$ENABLE_TO_VMS_LOGNAME_CACHE
UNIX 名変換の性能を改善するには， DECC$ENABLE_TO_VMS_LOGNAME_CACHE を使用します。この値は，各キャッシュ・エントリの存続時間 (秒) です。等価文字列 ENABLE は， 1 秒と見なされます。

各エントリの存続時間を 1 秒としてキャッシュを有効にするには，DECC$ENABLE_TO_VMS_LOGNAME_CACHE に 1 を定義します。

各エントリの存続時間を 2 秒としてキャッシュを有効にするには，DECC$ENABLE_TO_VMS_LOGNAME_CACHE に 2 を定義します。

キャッシュ・エントリの有効期限なしでキャッシュを有効にするには，DECC$ENABLE_TO_VMS_LOGNAME_CACHE に–1 を定義します。

DECC$EXEC_FILEATTR_INHERITANCE
DECC$EXEC_FILEATTR_INHERITANCE 機能論理名は， C プログラムの子プロセスに影響します。

V7.3-2 より前の OpenVMS では， DECC$EXEC_FILEATTR_INHERITANCE は，有効か無効のどちらかです。

- DECC$EXEC_FILEATTR_INHERITANCE が有効の場合，現在のファイル・ポインタとファイル・オープン・モードが，exec呼び出しで子プロセスに渡されます。
- この論理名が無効の場合，子プロセスは追加モードやファイル・ポインタを継承しません。

V7.3-2 およびそれ以降の OpenVMS では， DECC$EXEC_FILEATTR_INHERITANCE に 1 または 2 を定義するか，無効にすることができます。

- DECC$EXEC_FILEATTR_INHERITANCE に 1 を定義すると，子プロセスは，追加モード以外のすべてのファイル・アクセス・モードについてファイル位置を継承します。
- DECC$EXEC_FILEATTR_INHERITANCE に 2 を定義すると，子プロセスは，追加モードも含め，すべてのファイル・アクセス・モードについてファイル位置を継承します。
- DECC$EXEC_FILEATTR_INHERITANCE が無効の場合，子プロセスはいずれのアクセス・モードについてもファイル位置を継承しません。

DECC$FILENAME_ENCODING_UTF8
ファイル名を扱う C RTL ルーチンでは， UNIX スタイルのファイル名を使用する場合に UTF-8 エンコーディングのサポートします。

たとえば， ODS-5 ディスクでは， OpenVMS の DIRECTORY コマンドで次のような文字を含むファイル名をサポートします。

```
disk:[mydir]^U65E5^U672C^U8A9E.txt
```

このファイル名には，日，出身, 言語を意味する 3 つの UCS-2 文字が含まれています (ここではそれぞれ xxx， yyy， zzz と表記します)。

UTF-8 サポートが有効な場合， C プログラムは， VMS ディレクトリからこのファイル名を UTF-8 でエンコードされた文字列として読み取り、扱うことが可能になりました。

たとえば，後ろにreaddirを伴うopendir("/disk/mydir")は，提供されるdirent構造体のd_nameフィールドに次のようなファイルを置くことができます。

```
"\xE6\x97\xA5\xE6\x9C\xAC\xE8\xAA\x9E.txt"
```

このファイルは次のいずれかの呼び出しでオープンできます。

```
open("/disk/mydir/\xE6\x97\xA5\xE6\x9C\xAC\xE8\xAA\x9E.txt",O_RDWR,0)
open("/disk/mydir/xxxyyyzzz.txt", O_RDWR,0)
```

上記の "\ xE6\ x97\ xA5"は， UTF-8 エンコーディングで xxx 文字を表す，バイト・ストリーム E697A5 です。次の例では実際の文字でファイル名が示されています。

図 1–2 Unicode ファイル名の例

```
$ DIR $1$DKA100:[ENCODE].TXT
Directory $1$DKA100:[ENCODE]

UTF8.TXT;1    ^U65E5^U672C^U8A9A.TXT;1

Total of 2 files.
$ MCR JSYS$CONTROL SET RMS/FILE=SDEC
$ DIR $1$DKA100:[ENCODE].TXT

Directory $1$DKA100:[ENCODE]

UTF8.TXT;1   日本語.              TXT;1

Total of 2 files
$
```

この機能は， UTF-8 でエンコードされたファイル名を使用する内部ソフトウェアの UNIX との互換性を拡張します。

DECC$FILENAME_ENCODING_UTF8 機能論理名は， UNIX スタイルで指定された Unicode の UTF-8 エンコーディングのファイル名を C RTL が正しく解釈するかどうかを制御します。

この機能論理名はデフォルトでは定義されておらず，ファイル名を ASCII と Latin-1 の形式として扱う C RTL の動作が適用されます。

この機能は ODS-5 ディスクでのみ機能します。このため， Unicode UTF-8 エンコーディングを有効にするには， DECC$FILENAME_ENCODING_UTF8 と DECC$EFS_CHARSET 論理名の両方を ENABLE と定義する必要があります。

DECC$FILENAME_UNIX_ONLY
DECC$FILENAME_UNIX_ONLY を有効に設定すると，ファイル名は OpenVMS 形式の名前として解釈されません。このため，以下の文字が OpenVMS の特殊文字として解釈されるのを防止できます。

: [^

DECC$FILENAME_UNIX_NO_VERSION
DECC$FILENAME_UNIX_NO_VERSION を有効に設定すると， UNIX 形式のファイル名で OpenVMS のバージョン番号はサポートされません。

DECC$FILENAME_UNIX_NO_VERSION を無効に設定すると， UNIX 形式の名前でバージョン番号を報告するときに，その前にピリオド(.)が付加されます。

DECC$FILENAME_UNIX_REPORT
DECC$FILENAME_UNIX_REPORT を有効に設定すると，呼び出し元が特に OpenVMS 形式を選択しない限り，ファイル名はすべて UNIX 形式で報告されます。これは`getpwnam`，`getpwuid`，`argv[0]`，`getname`，`fgetname`，`tempnam`に適用されます。

DECC$FILENAME_UNIX_REPORT を無効に設定すると，関数呼び出しで指定しない限り，ファイル名は OpenVMS 形式で報告されます。

DECC$FILE_PERMISSION_UNIX
DECC$FILE_PERMISSION_UNIX を有効に設定すると，新しいファイルとディレクトリのファイル・アクセス許可は，ファイルの作成モードおよび`umask`に従って設定されます。これにはモード 0777 も含まれます。ファイルの以前のバージョンが存在する場合は，新しいファイルのファイル・アクセス許可は古いバージョンから継承されます。 WRITE アクセス許可が有効に設定されていると，このモードは新しいディレクトリに対して DELETE アクセス許可を設定します。

DECC$FILE_PERMISSION_UNIX を無効に設定すると，モード 0 およびモード 0777 は RMS のデフォルト保護またはファイルの前のバージョンの保護を使用することを示します。また，新しいディレクトリのアクセス許可は， DELETE アクセス許可の無効化も含めて， OpenVMS の規則に従います。

DECC$FILE_SHARING
DECC$FILE_SHARING を有効に設定すると，すべてのファイルは完全な共用を有効にした状態でオープンされます (FAB$M_DEL | FAB$M_GET | FAB$M_PUT | FAB$M_UPD)。これは，呼び出し元が指定した共用モードとの論理和 (OR) として指定されます。

DECC$FIXED_LENGTH_SEEK_TO_EOF
DECC$FIXED_LENGTH_SEEK_TO_EOF を有効に設定すると，directionパラメータが SEEK_END に設定された`lseek`，`fseeko`，`fseek`は，固定長レコードのファイル内の最後のバイトを基準に位置を設定します。

DECC$FIXED_LENGTH_SEEK_TO_EOF を無効に設定すると，固定長レコードのファイルに対して SEEK_EOF を設定して呼び出した場合，`lseek`，`fseek`，`fseeko`はファイル内の最後のレコードの末尾を基準にして位置を設定します。

DECC$GLOB_UNIX_STYLE
DECC$GLOB_UNIX_STYLE を有効にすると，glob関数の UNIX モードが選択され， OpenVMS スタイルのファイル名やワイルドカードの代わりに， UNIX スタイルのファイル名やワイルドカードが使用されます。

DECC$LOCALE_CACHE_SIZE
DECC$LOCALE_CACHE_SIZE は，ロケール・データをキャッシュに格納するために割り振るメモリ容量をバイト数で定義します。デフォルト値は 0 であり，その場合はロケール・キャッシュは無効になります。

デフォルト値: 0

最大値: 2147483647

DECC$MAILBOX_CTX_STM
デフォルト設定では，パイプではないローカル・メールボックスに対するopenは，FAB$M_CR というレコード属性を持つものとして，メールボックス・レコードを取り扱います。

DECC$MAILBOX_CTX_STM を有効に設定すると，レコード属性 FAB$M_CR は設定されません。

DECC$NO_ROOTED_SEARCH_LISTS
decc$to_vms関数は， UNIX 形式のパス文字列を評価する際に，論理名となる 1 番目の要素を検出すると，次の処理を行います。

- 最上位の論理名またはデバイスの場合は，論理名に ":[000000]"を付加します。

  たとえば，log1が最上位の論理名 ($DEFINE LOG1 [DIR_NAME.]) の場合，/log1/filename.extは LOG1:[000000]FILENAME.EXT に変換されます。
- 最上位でない論理名の場合は，論理名にコロン(:)だけを付加します。

  たとえば，log2が最上位でない論理名 ($ DEFINE LOG2 [DIR_NAME]) の場合，/log2/filename.extは LOG2:FILENAME.EXT に変換されます。
- 1 番目の要素が検索リスト論理名の場合，変換は，検索リスト内の 1 番目の要素を評価して，以前に説明したようにパスを変換することにより進められます。

上記の 3 つのケースでは，予想可能な，期待どおりの結果が得られます。

1 番目の要素が，最上位の論理名と最上位でない論理名が混在した検索リストの場合，以前の説明のとおりにパスを変換すると，古いバージョン (7.3-1 よりも前) の OpenVMS とは異なる動作になることがあります。

- OpenVMS Version 7.3-1 より前の場合，論理名の内容とは関係なく，decc$to_vms関数はコロン(:)だけを付加していました。最上位の論理名と最上位でない論理名が混在した検索リストの場合，期待どおりの結果が得られます。

- OpenVMS Version 7.3-1 およびそれ以降では，混在している検索リストの 1 番目の要素が最上位の論理名の場合，decc$to_vmsが論理名に ":[000000]"を付加します。その結果， OpenVMS Version 7.3-1 より前のリリースとは，異なる動作になります。

DECC$NO_ROOTED_SEARCH_LISTS は，decc$to_vms関数が検索リスト論理名を解決する方法を制御します。また， OpenVMS の動作を V7.3-1 より前の状態に戻すこともできます。

DECC$NO_ROOTED_SEARCH_LISTS を有効にすると，次のようになります。

- 論理名がファイル指定内で検出され，検索リストだった場合， OpenVMS ファイル指定を作成するときに，コロン(:)が付加されます。
- 論理名が検索リストでない場合，動作は， DECC$NO_ROOTED_SEARCH_LISTS が無効の場合と同じです。

この機能論理名を有効にすると，検索リスト論理名の動作が， 7.3-1 以前の状態になります。

DECC$NO_ROOTED_SEARCH_LISTS を無効にすると，次のようになります。

- ファイル指定内で論理名が検出され，最上位の論理名 (つまり， 1 番目の要素が最上位論理名である検索リスト) だった場合， OpenVMS ファイル指定の作成時に，":[000000]"が付加されます。
- 最上位でない論理名の場合 (つまり， 1 番目の要素が最上位でない論理名である検索リストの場合) は，コロン(:)だけが追加されます。

この機能論理名を無効にすると， OpenVMS Version 7.3-1 およびそれ以降の動作になります。

DECC$PIPE_BUFFER_SIZE
パイプ書き込み操作の場合， 512 バイトというシステムのデフォルト・バッファ・サイズでは， 512 バイトより長いメッセージを取り扱うときに，性能が低下し，余分なライン・フィードが生成される可能性があります。

DECC$PIPE_BUFFER_SIZE を指定すると，pipeやpopenなどのパイプ関数に対して，大きなバッファ・サイズを使用できます。 512 ～ 65535 バイトまでの値を指定できます。

DECC$PIPE_BUFFER_SIZE を指定しないと，デフォルト・バッファ・サイズである 512 が使用されます。

デフォルト値: 512

最小値: 512

最大値: 65024

DECC$PIPE_BUFFER_QUOTA

OpenVMS Version 7.3-2 では，パイプのメールボックスのバッファ・クォータを指定する，int型の第 4 引数 (オプション) が，pipe関数に追加されました。以前のバージョンの OpenVMS では，バッファ・クォータはバッファ・サイズと同じでした。

DECC$PIPE_BUFFER_QUOTA を使用すると，この関数のオプションの第 4 引数を省略した場合に，pipe関数が使用するバッファ・クォータを指定できます。

pipeのオプションの第 4 引数を省略し， DECC$PIPE_BUFFER_QUOTA も定義していない場合，バッファ・クォータは，デフォルトで，以前と同じようにバッファ・サイズになります。

デフォルト値: 512

最小値: 512

最大値: 2147483647

DECC$POPEN_NO_CRLF_REC_ATTR
DECC$POPEN_NO_CRLF_REC_ATTR を無効に設定すると，popen関数を使用してオープンしたパイプのレコード属性には， CR/LF キャリッジ制御 (fab$b_rat |= FAB$M_CR) が設定されます。これがデフォルトの動作です。

DECC$POPEN_NO_CRLF_REC_ATTR を有効に設定すると， CR/LF キャリッジ制御がパイプ・レコードに追加されません。これは， UNIX の動作と互換性がありますが，この機能を有効にすると，キャリッジ・リターン文字に依存しているgetsのようなほかの関数で望ましくない動作になる可能性があります。

DECC$POSIX_COMPLIANT_PATHNAMES
DECC$POSIX_COMPLIANT_PATHNAMES を有効に設定すると，アプリケーションから C RTL 関数に渡すパス名を POSIX 準拠のパス名にすることができます。

DECC$POSIX_COMPLIANT_PATHNAMES は特に指定しない限り無効になっていて，通常の C RTL 動作が優先されます。この無効になる動作の中には UNIX 仕様のパス名を解釈する方法も含まれていて， POSIX 準拠のパス名処理とは無関係な，異なる規則が使用されます。

DECC$POSIX_COMPLIANT_PATHNAMES を有効にするには，この機能論理名の値として，次のいずれかの値を設定します。

1 パス名をすべて， POSIX のスタイルで表す。
2 ":"で終わっているかまたは任意のかっこ "[ ]<>"を含んでいて，しかも SYS$FILESCAN サービスで解析可能なパス名は，そのすべてを OpenVMS のスタイルで表し，それ以外のパス名は， POSIX のスタイルで表す。

3 "/"を含むパス名と，パス名 "."および ".."は POSIX のスタイルで表し，それ以外のパス名は， OpenVMS のスタイルで表す。
4 パス名をすべて， OpenVMS のスタイルで表す。

詳細は，第 12.3.1 項を参照してください。

DECC$POSIX_SEEK_STREAM_FILE
DECC$POSIX_SEEK_STREAM_FILE を有効に設定すると， STREAM ファイルでファイルの終端 (EOF) を超えた位置設定を行った場合，次の書き込みまでファイルへの書き込みが行われません。書き込みが現在のファイルの終端を超える場合，古いファイルの終端を超えた位置から書き込みの先頭までの位置に 0 が埋め込まれます。

DECC$POSIX_SEEK_STREAM_FILE を無効に設定すると，ファイルの終端を超える位置設定を行った場合，現在のファイルの終端から新しい位置までの間に 0 がただちに書き込まれます。

DECC$POSIX_STYLE_UID
DECC$POSIX_STYLE_UID が有効の場合， 32 ビットの UID と GID は， POSIX 形式の識別子として解釈されます。

この論理名が無効の場合， UID と GID はプロセスの UIC から得られます。

この機能は， POSIX 形式の UID と GID をサポートしている OpenVMS システムでのみ利用できます。

DECC$PRINTF_USES_VAX_ROUND
DECC$PRINTF_USES_VAX_ROUND を有効すると，printfの F および E 形式の指定子は， IEEE 浮動小数点でのプログラムのコンパイルに VAX の切り上げ切捨て規則を使用します。

DECC$READDIR_DROPDOTNOTYPE
DECC$READDIR_DROPDOTNOTYPE を有効に設定すると，readdirは UNIX 形式でファイル名を報告するときに，ファイル名にピリオド(.)が含まれている場合，ファイル・タイプのないファイルに対して末尾のピリオドだけを報告します。

この論理名を無効に設定すると，ファイル・タイプのないすべてのファイル名の末尾にピリオドが付加されて報告されます。

DECC$READDIR_KEEPDOTDIR
readdirから UNIX 形式のファイルを報告する場合，デフォルト動作では，ファイル・タイプを付けずにディレクトリだけが報告されます。

DECC$READDIR_KEEPDOTDIR を有効に設定すると，ディレクトリは UNIX 形式でファイル・タイプ ".DIR"を付けて報告されます。

DECC$RENAME_NO_INHERIT
DECC$RENAME_NO_INHERIT は，rename関数で，より UNIX に準拠した動作を提供します。 DECC$RENAME_NO_INHERIT が有効な場合は，次のような動作を強要します。

- 古い引数がファイルのパス名を指す場合，新しい引数がディレクトリのパス名を指すことはありません。
- 新しい引数で既存のディレクトリを指定することはできません。
- 古い引数がディレクトリのパス名を指す場合，新しい引数がファイルのパス名を指すことはありません。
- ファイルの新しい名前は古い名前から何も継承しません。新しい名前は完全に指定する必要があります。次に例を示します。

  "A.A"の名前を "B"に変更すると， "B"になります。

この論理名が無効の場合， OpenVMS で期待されている動作になります。例を次に示します。

"A.A"の名前を "B"に変更すると， "B.A"になります。

DECC$RENAME_ALLOW_DIR
DECC$RENAME_ALLOW_DIR を有効にすると，rename関数の動作が以前の OpenVMS の状態に戻ります。以前の動作では，あいまいなファイル指定が第 2 引数で論理名として渡された場合，ディレクトリ指定へ変換されます。あいまい性は，論理名が UNIX ファイル指定と OpenVMS ファイル指定のどちらかという点にあります。

例を次に示します。

```
rename("file.ext", "logical_name") /* where logical_name = dev:[dir.subdir] */
                                   /* and :[dir.subdir] exists.            */
```

次のような結果になります。

```
dev:[dir.subdir]file.ext
```

この例では，ファイル名の変更により，あるディレクトリから別のディレクトリへファイルが移動されます。この動作は， V7.3-1 より前の OpenVMS の古い動作と同じです。またこの例では，dev:[dir.subdir]が存在しない場合，renameがエラーを返します。

DECC$RENAME_ALLOW_DIR を無効にすると，renameのlogical_name引数の変換が，より UNIX に準拠したものになります。

例を次に示します。

```
rename("file.ext", "logical_name")   /* where logical_name = dev:[dir.subdir] */
```

次のような結果になります。

```
dev:[dir]subdir.ext
```

この例では，logical_name引数のsubdir部分を新しいファイル名として使用して，ファイル名が変更されます。これは， UNIX システム上ではファイルからディレクトリへの名前変更 (rename) はできないためです。このためrenameは内部的に"logical_name"をファイル名に変換し， to a file name, dev:[dir]subdirが最も妥当な変換結果となります。

この新しい機能には，名前をファイルに変更するのではなくディレクトリに変更するという副作用があります。次に例を示します。

```
rename ( "file1.ext", "dir2" )      /* dir2 is not a logical */
```

DECC$RENAME_ALLOW_DIR が無効な場合，この例では，サブディレクトリ[.dir2]が存在するかどうかにかかわらずdir2.extという結果になります。

DECC$RENAME_ALLOW_DIR が有効な場合，[.dir2]が存在しなければdir2.extという結果になります。サブディレクトリ[.dir2]が存在する場合は，[.dir2]file1.extという結果になります。

注意

DECC$RENAME_NO_INHERIT が有効の場合は，UNIX 準拠の動作が行われます。このため，DECC$RENAME_ALLOW_DIR は無視され，ファイルからディレクトリへの名前変更 (rename) は許されません。

DECC$SELECT_IGNORES_INVALID_FD
DECC$SELECT_IGNORES_INVALID_FD を有効に設定すると，記述子セットのいずれかに不正なファイル記述子が指定されている場合，selectは異常終了し，errnoは EBADF に設定されます。

DECC$SELECT_IGNORES_INVALID_FD を無効に設定すると，selectは不正なファイル記述子を無視します。

DECC$STDIO_CTX_EOL
DECC$STDIO_CTX_EOL を有効に設定すると，ストリーム・アクセスのためのstdoutおよびstderrへの書き込みは，区切り文字が確認されるか，バッファが満杯になるまで実行されません。

DECC$STDIO_CTX_EOL を無効に設定すると，fwriteを実行するたびに個別に書き込みが実行され，メールボックスとレコード・ファイルの場合，個別のレコードが作成されます。

DECC$STREAM_PIPE
DECC$STREAM_PIPE を有効に設定すると， C RTL のpipe関数は， UNIX との互換性が高いストリーム入出力を使用します。

DECC$STREAM_PIPE を無効に設定すると，pipeは従来の OpenVMS のレコード入出力を使用します。これがデフォルトの動作です。

DECC$STRTOL_ERANGE
DECC$STRTOL_ERANGE を有効に設定すると， ERANGE エラーに対するstrtolの動作は，文字列内の残りのすべての桁を使用するように修正されます。

DECC$STRTOL_ERANGE を無効に設定すると，問題が発生した桁にポインタを保持するというこれまでの動作が有効になります。

DECC$THREAD_DATA_AST_SAFE
C RTL には， AST のために割り振られるデータ用とは別に，非 AST レベルでスレッドによって割り振られるスレッド固有のデータのための記憶域を割り振るモードがあります。このモードでは，スレッド固有のデータにアクセスするたびに，LIB$AST_IN_PROG を呼び出す必要があり，C RTLでスレッド固有のデータにアクセスする場合，大きなオーバヘッドが発生する可能性があります。

代替モードでは，スレッド固有のデータは，別の関数がロックしている場合にだけ保護されます。このモードでは，C RTLの内部で使用中のデータは保護されますが，AST がデータを変更するのを呼び出し元で保護することはできません。

この 2 番目のモードは，strtok，ecvt，fcvt関数の C RTL デフォルトになりました。

DECC$THREAD_DATA_AST_SAFE を有効に設定すると，従来の AST セーフ・モードを選択できます。

DECC$TZ_CACHE_SIZE
DECC$TZ_CACHE_SIZE は，メモリに保持できるタイム・ゾーンの数を指定します。

デフォルト値: 2

最大値: 2147483647

DECC$UMASK
DECC$UMASK は，アクセス許可マスクumaskのデフォルト値を指定します。デフォルト設定では，親の C プログラムは，プロセスの RMS デフォルト・アクセス許可からumaskを設定します。子プロセスはumaskの値として，親の値を継承します。

値を 8 進数として入力するには，先頭に 0 を追加します。 0 を追加しないと， 10 進数として変換されます。次の例を参照してください。

```
$ DEFINE DECC$UMASK 026
```

最大値: 0777

DECC$UNIX_LEVEL
DECC$UNIX_LEVEL 論理名を使用すると，複数の C RTL 機能論理名を一度に操作できます。設定した値は蓄積効果があり，値が大きいほど影響を受けるグループが多くなります。たとえば値を 20 に設定すると， 20， 10，および 1 の DECC$UNIX_LEVEL に対応するすべての機能論理名が有効になります。

UNIX 風の動作に影響する主な論理名は，次のグループに分類できます。

1	全般的な補正
10	拡張
20	UNIX 形式のファイル名
30	UNIX 形式のファイル属性
90	完全な UNIX 動作 - OpenVMS に対する配慮なし

BASH や GNV などの UNIX 風のプログラムに対しては，レベル 30 が適切です。

DECC$UNIX_LEVEL 値と，影響を受ける機能論理名のグループの対応は，次のとおりです。

```
General Corrections          (DECC$UNIX_LEVEL 1)

   DECC$FIXED_LENGTH_SEEK_TO_EOF   1
   DECC$POSIX_SEEK_STREAM_FILE     1
   DECC$SELECT_IGNORES_INVALID_FD  1
   DECC$STRTOL_ERANGE              1
   DECC$VALIDATE_SIGNAL_IN_KILL    1

General Enhancements        (DECC$UNIX_LEVEL 10)

   DECC$ARGV_PARSE_STYLE           1
   DECC$EFS_CASE_PRESERVE          1
   DECC$STDIO_CTX_EOL              1
   DECC$PIPE_BUFFER_SIZE           4096
   DECC$USE_RAB64                  1

UNIX style file names       (DECC$UNIX_LEVEL 20)

   DECC$DISABLE_TO_VMS_LOGNAME_TRANSLATION 1
   DECC$EFS_CHARSET                1
   DECC$FILENAME_UNIX_NO_VERSION   1
   DECC$FILENAME_UNIX_REPORT       1
   DECC$READDIR_DROPDOTNOTYPE      1
   DECC$RENAME_NO_INHERIT          1

UNIX like file attributes     (DECC$UNIX_LEVEL 30)
```

```
   DECC$EFS_FILE_TIMESTAMPS        1
   DECC$EXEC_FILEATTR_INHERITANCE  1
   DECC$FILE_OWNER_UNIX            1
   DECC$FILE_PERMISSION_UNIX       1
   DECC$FILE_SHARING               1
UNIX compliant behavior       (DECC$UNIX_LEVEL 90)
   DECC$FILENAME_UNIX_ONLY         1
   DECC$POSIX_STYLE_UID            1
   DECC$USE_JPI$_CREATOR           1
   DECC$DETACHED_CHILD_PROCESS     1
```

注意

- **個々の機能論理名の論理名を定義すると，**DECC$UNIX_LEVEL **で指定されたその機能のデフォルト値より優先されます。**
- C RTL **の将来のリビジョンでは，**DECC$UNIX_LEVEL **に新しい機能論理名が追加される可能性があります。**UNIX **レベルを指定するアプリケーションに対しては，デフォルトでこれらの新しい機能論理名が有効になります。**

DECC$UNIX_PATH_BEFORE_LOGNAME
DECC$UNIX_PATH_BEFORE_LOGNAME を有効に設定すると，先頭がスラッシュ (/) でない UNIX ファイル名を変換するときに，この名前は現在のディレクトリのファイルまたはディレクトリに対応付けられます。このようなファイルやディレクトリが見つからず，名前が OpenVMS ファイル名で論理名として有効な場合は，論理名変換が実行されます。論理名が検索されて変換されると，ファイル名の一部として使用されます。

DECC$UNIX_PATH_BEFORE_LOGNAME を有効に設定すると，DECC$DISABLE_TO_VMS_LOGNAME_TRANSLATION の設定は無効になります。

DECCUSE_JPI_CREATOR
DECCUSE_JPI_CREATOR が有効の場合， JPI$_OWNER の代わりに JPI$_CREATOR 項目を使用して$GETJPI を呼び出し，getppidの親プロセス ID が決定されます。

この機能は， POSIX 形式のセッション識別子をサポートしているシステムでのみ利用できます。

DECC$USE_RAB64
DECC$USE_RAB64 を有効に設定すると，オープン関数は従来の RAB 構造体ではなく， RAB64 構造体を割り振ります。

これは， 64 ビット・メモリ内のファイル・バッファに対する潜在的なサポート機能を提供します。

DECC$VALIDATE_SIGNAL_IN_KILL
DECC$VALIDATE_SIGNAL_IN_KILL を有効に設定すると， 0 ～_SIG_MAX の範囲内であっても，C RTLでサポートされないシグナル値はエラーを生成し，errnoはEINVAL に設定されます。この結果，raiseと同じ動作が実行されます。

この論理名を無効に設定すると，シグナルのチェックは，シグナル値が 0 ～_SIG_MAX の範囲内であるかどうかというチェックに制限されます。sys$sigprcが異常終了すると，errnoはsys$sigprcの終了状態をもとに設定されます。

DECC$V62_RECORD_GENERATION
OpenVMS Version 6.2 以降では，異なる規則を使用してレコード・ファイルを出力できるようになりました。

DECC$V62_RECORD_GENERATION を有効に設定すると，出力機能は OpenVMS Version 6.2 に対して使用される規則に従います。

DECC$WRITE_SHORT_RECORDS
DECC$WRITE_SHORT_RECORDS 機能論理名は，固定長ファイルへの従来のレコードの書き込み方式をデフォルトの動作として維持しながら，fwrite関数への以前の変更 (最大レコード・サイズより小さいサイズのレコード書き込みへの対応) をサポートしています。

DECC$WRITE_SHORT_RECORDS が有効の場合， EOF に書き込まれたショート・サイズ・レコード (サイズが最大レコード・サイズ未満のレコード) は，レコードをレコード境界に合わせるために，ゼロでパディングされます。これは， OpenVMS Version 7.3-1 と，この時期の一部の ACRTL ECO に見られる動作です。

DECC$WRITE_SHORT_RECORDS が無効の場合，パディングなしでレコードを書き込む従来の動作が実行されます。これが，推奨される，デフォルトの動作です。

DECC$XPG4_STRPTIME
XPG5 でのstrptimeのサポートでは，ピボット年のサポートが導入され， 0 ～ 68 の年は 21 世紀， 69 ～ 99 の年は 20 世紀であると解釈されるようになりました。

DECC$XPG4_STRPTIME を有効に設定すると， XPG5 のピボット年のサポートは無効になり， 0 ～ 99 のすべての年が現在の世紀であると解釈されます。

1.6 32 ビットの UID/GID と POSIX 形式の識別子

OpenVMS オペレーティング・システムのバージョンで POSIX 形式の識別子がサポートされる場合， POSIX 形式の識別子はユーザ識別子 (UID)，グループ識別子 (GID)，プロセス・グループを参照します。スコープには実識別子と実効識別子が含まれます。

HP C RTL で POSIX 形式の識別子をサポートするには，32 ビットのユーザ ID とグループ ID のサポートが必要であり，サポートされるかどうかは，OpenVMS の基本バージョンの機能に応じて異なります。POSIX 形式の ID は，OpenVMS Version 7.3-2 およびそれ以降でサポートされています。

POSIX 形式の識別子をサポートしているバージョンの OpenVMS でこの識別子を使用するには，アプリケーションが 32 ビット UID/GID 用にコンパイルされていなければなりません。32 ビット UID/GID がデフォルトの OpenVMS バージョンでも，ユーザやアプリケーションは，DECC$POSIX_STYLE_UID 機能論理名を定義して，POSIX 形式の ID を有効にしなければなりません。

```
$ DEFINE DECC$POSIX_STYLE_UID ENABLE
```

POSIX 形式の ID を有効にすると，コンパイル時に，個々の関数に対して，従来 (UIC ベース) の定義を呼び出すこともできます。この場合は，decc$が前についたエントリ・ポイント (POSIX 形式の動作を行う，decc$__long_gid_が前に付いたエントリ・ポイントではなく) を明示的に呼び出します。

POSIX 形式の ID を無効にするには，次の定義を行います。

```
$ DEFINE DECC$POSIX_STYLE_UID DISABLE
```

OpenVMS Version 7.3-2 およびそれ以降では，POSIX 形式の ID と 32 ビットの UID/GID の両方がサポートされます。32 ビットの UID/GID を使用するように設定してアプリケーションをコンパイルした場合，UID と GID はオペレーティング・システムの以前のバージョンと同様に UIC から取得されます。場合によっては，getgroups関数の場合のように，アプリケーションで 32 ビットの GID がサポートされる場合，より多くの情報が返されることがあります。

デフォルトで 32 ビットの UID/GID を使用するシステムで，16 ビットの UID/GID をサポートするように設定したアプリケーションをコンパイルするには，マクロ_DECC_SHORT_GID_T に 1 を定義します。

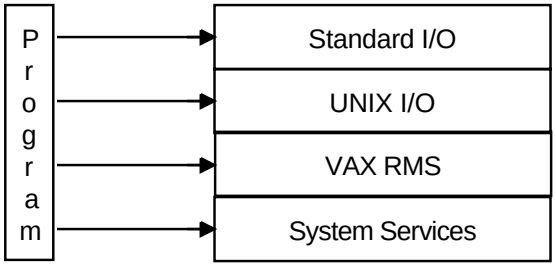
1.7 OpenVMS システムでの入出力

HP C RTL とリンクする方法，およびHP C関数とマクロを呼び出す方法を学習したら，次に主要な目的である入出力 (I/O) のためにHP C RTL を使用できるようになります。

システムごとに I/O の方法は異なっているため，OpenVMS 固有のファイル・アクセス方式を十分理解しておくことが必要です。十分理解しておけば，ソース・プログラムをあるオペレーティング・システムから別のオペレーティング・システムに移植するときに，機能上の相違点をあらかじめ予測することができます。

図 1–3 は，HP C RTL で使用できる I/O 方式を示しています。 OpenVMS システム・サービスは OpenVMS オペレーティング・システムと直接通信するため，オペレーティング・システムに最も近い位置にあります。 OpenVMS RMS (Record Management Services) 関数はシステム・サービスを使用し，それらのシステム・サービスがオペレーティング・システムを操作します。HP Cの標準 I/O および UNIX I/O 関数とマクロでは， RMS 関数を使用します。HP C RTL 標準 I/O および UNIX I/O 関数およびマクロは，システムを操作するまでに複数の関数呼び出しのレイヤを通過しなければならないため，オペレーティング・システムから最も遠い位置にあります。

図 1–3 C プログラムからの I/O インタフェース



ZK-0493-GE

C プログラミング言語は UNIX オペレーティング・システムで開発されており，標準 I/O 関数は，ほとんどのアプリケーションで十分効率よく強力で便利な I/O 方式を提供できるように設計されており，さらに C 言語コンパイラが稼動するどのシステムでも関数を使用できるように，移植可能になるように設計されています。

HP C RTL では，このもともとの仕様にさらに機能が追加されています。HP C RTL で実装されている標準 I/O 関数は，行区切り文字を認識するので，HP C RTL の標準 I/O 関数は特に，テキスト操作の場合に便利です。HP C RTL では，一部の標準 I/O 関数はプリプロセッサ定義マクロとして実装されています。

同様に， UNIX の I/O 関数はもともと， UNIX オペレーティング・システムにより直接的にアクセス可能になるように設計されています。これらの関数では，数値のファイル記述子を使用してファイルを表現します。 UNIX システムでは，統一されたアクセス方式を可能にするために，すべての周辺デバイスがファイルとして表現されます。

HP C RTL では，もともとの仕様にさらに機能が追加されています。HP Cで実装されている UNIX I/O 関数は，特にバイナリ・データを操作するのに便利です。HP C RTL ではまた，一部の I/O 関数はプリプロセッサ定義マクロとして実装されています。

HP C RTL には，すべての C コンパイラにある標準 I/O 関数が用意されており，その他にできるだけ多くの他の C の実装と互換性を維持するために UNIX I/O 関数も用意されています。しかし，標準 I/O と UNIX I/O のどちらも，ファイルにアクセスするために RMS を使用します。標準 I/O 関数と UNIX I/O 関数が RMS でフォーマットされたファイルを操作する方法を理解するには， RMS の基礎を学習する必要があります。 RMS ファイルに関連する標準 I/O と UNIX I/O の詳細については，第 1.7.1 項を参照してください。 RMS の概要については，『Guide to OpenVMS File Applications』を参照してください。

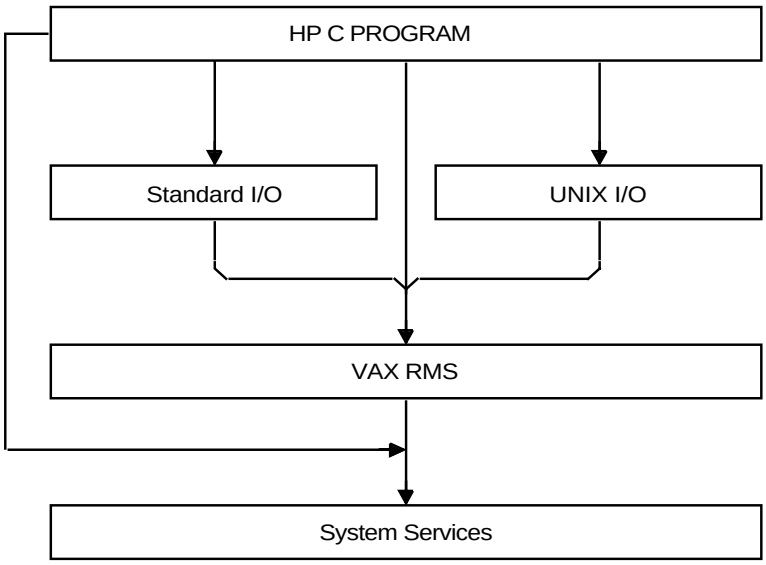
どの方法が適切であるかを判断する前に，まず，「UNIX との互換性が重要なのか，OpenVMS オペレーティング・システムのもとで単独に動作するコードを開発するのが重要なのか」という問題について検討してください。

- UNIX との互換性が重要である場合は，最高レベルの I/O，つまり標準 I/O とUNIX I/Oを使用することが必要でしょう。なぜなら，このレベルはオペレーティング・システムからの独立性がかなり高いからです。また，最高レベルの I/O は学習するのも簡単です。初心者のプログラマの場合，このことは重要な要素です。
- UNIX との互換性が重要でない場合や，標準 I/O および UNIX I/O 方式で提供されない高度なファイル処理が必要な場合は， RMS を使用することが望ましいでしょう。

システム・レベルのソフトウェアを開発する場合，システム・サービスへの呼び出しを使用して OpenVMS オペレーティング・システムに直接アクセスしなければならないことがあります。たとえば，$QIO (Queue I/O Request) システム・サービスを通じて，直接ユーザ作成デバイス・ドライバにアクセスしなければならないことがあります。この場合は， OpenVMS レベルの I/O を使用します。経験の豊富な OpenVMS プログラマの場合は，このレベルを推奨します。 OpenVMS システム・サービスを呼び出すプログラムの例については，『HP C User's Guide for OpenVMS Systems』を参照してください。

おそらく， RMS や OpenVMS システム・サービスを使用しないこともあるでしょう。多くのアプリケーションでは，標準 I/O 関数と UNIX I/O 関数が十分効率的に機能します。図 1–4 は，標準 I/O 関数および UNIX I/O 関数と RMS の依存関係を示しており，使用できるさまざまな I/O 方式も示しています。

図 1–4 標準 I/O および UNIX I/O と RMS の対応関係



1.7.1 RMS のレコード・フォーマットとファイル・フォーマット

標準 I/O および UNIX I/O の関数とマクロの機能および制約事項を理解するには，OpenVMS RMS (Record Management Services) について理解する必要があります。

RMS では次のファイル編成がサポートされます。

- 順編成
- 相対編成
- 索引順編成

順編成ファイルにはレコードが連続的に記録され，レコードとレコードの間に空のレコードは存在しません。相対編成ファイルには固定長のセルが記録され，各セルにはレコードが格納されていることも，格納されていないこともあります。索引順編成ファイルには，データ，キャリッジ制御情報，さまざまなアクセス順序を可能にするキーを格納したレコードが記録されます。

HP C RTL の関数は順編成ファイルにだけアクセスできます。他のファイル編成を使用する場合は， RMS 関数を使用する必要があります。 RMS 関数の詳細については，『HP C User's Guide for OpenVMS Systems』を参照してください。

RMS はレコードの内容を考慮せず，レコードのフォーマットを考慮します。レコードのフォーマットとは，記憶媒体の記録面にレコードが物理的に記録される方法です。

RMS では次のレコード・フォーマットがサポートされます。

- 固定長
- 可変長
- 固定長制御部付可変長 (VFC)
- ストリーム

固定長レコード・フォーマットはファイルの作成時に指定できます。このフォーマットでは，すべてのレコードがファイル内で同じサイズの領域を使用します。ファイルの作成後にレコード・フォーマットを変更することはできません。

可変長，VFC，ストリーム・ファイル・フォーマットのレコードの長さは，最大サイズまでの範囲で変化することができ，最大サイズはファイルの作成時に指定しなければなりません。可変長レコードまたは VFC フォーマットのファイルでは，レコードのサイズはデータ・レコードの先頭にあるヘッダ・セクションに格納されます。ストリーム・ファイルでは，キャリッジ制御文字やライン・フィード文字など，特定の文字が検出されたときに，RMS はレコードを終了します。ストリーム・ファイルはテキストを格納するのに便利です。

RMS では，ファイル内のレコードのキャリッジ制御属性を指定できます。このような属性としては，暗黙のキャリッジ・リターンや Fortran でフォーマットされたレコードがあります。ファイルを端末やライン・プリンタ，他のデバイスに出力するときに，RMS はこれらのキャリッジ制御を解釈します。キャリッジ制御情報はデータ・レコードに格納されません。

デフォルト設定では，ファイルの前のバージョンが存在する場合，ファイルは RMS レコード・フォーマット，最大レコード・サイズ，レコード属性を前のバージョンから継承します。OpenVMS システム・プログラマの場合，継承された属性は FAB$B_RFM，FAB$W_MRS，FAB$B_RAT と呼びます。前のバージョンが存在しない場合，新たに作成されたファイルのデフォルトはストリーム・フォーマットになり，レコードの終端はライン・フィード・レコード区切り文字および暗黙のキャリッジ・リターン属性で決定されます (本書では，この種のファイルを**ストリーム・ファイル**と呼びます)。ストリーム・ファイルは，HP C RTL の標準 I/O および UNIX I/O 関数を使用して操作することができます。これらのファイルや，キャリッジ制御を含まない固定長レコード・ファイルを使用する場合，fseek関数やlseek関数を使用して，ファイルのランダムなバイトまでシークする機能に制限はありません。しかし，可変長レコード・フォーマットなど，ファイルに他の RMS レコード・フォーマットのいずれかが含まれる場合は，RMS の制限により，これらの関数はレコード境界までしかシークできません。他の VAX 言語やユーティリティで使用するファイルを作成またはアクセスしなければならない場合を除き，デフォルトの VAX ストリーム・フォーマットを使用してください。

1.7.2 RMS ファイルへのアクセス

RMS の順編成ファイルはレコード・モードまたはストリーム・モードでオープンすることができます。デフォルト設定では， STREAM_LF ファイルはストリーム・モードでオープンされます。他のすべてのファイル・タイプはレコード・モードでオープンされます。ファイルをオープンするときに，省略可能な引数 "ctx=rec"を指定することで，これらのデフォルト設定をレコード・モードに変更したり， "ctx=stm"を指定することでストリーム・モードに設定することができます。 RMS の相対編成ファイルと索引順編成ファイルは常にレコード・モードでオープンされます。アクセス・モードによって，HP C RTL でのさまざまな I/O 関数の動作が決定されます。

RMS で定義されているファイル・タイプの 1 つに，RMS–11ストリーム・フォーマット・ファイルがあります。このファイル・タイプは，レコード・フォーマットの FAB$C_STM の値に対応します。このフォーマットは， SYS$GET が各レコードから先頭のヌル・バイトを削除する RMS レコード操作として定義されています。このファイル・タイプはHP C RTL によってレコード・モードで処理されるため，明示的に "ctx=stm"を指定してオープンしない限り，バイナリ・データのファイル・フォーマットとしては不適切です。 "ctx=stm"を指定した場合は，ファイルのデータ・バイトがそのまま返されます。

注意

OpenVMS Version 7.0 **で，ストリーム・ファイルの** LRL **のデフォルト値は** 0 **から** 32767 **に変更されました。この変更により，ソートなどの特定のファイル操作で性能が著しく低下しました。**

しかし，この問題は回避することができます。HP C RTL **では，論理名** DECC$DEFAULT_LRL **を定義することで，ストリーム・ファイルのレコード長のデフォルト値を変更できるようになりました。**

HP C RTL **は最初にこの論理名を検索します。この論理名が検索され，** 0 **～** 32767 **の範囲の数値に変換されると，その値がデフォルト** LRL **として使用されます。**

OpenVMS Version 7.0 **より前の動作に戻すには，次のコマンドを入力します。**

```
$ DEFINE DECC$DEFAULT_LRL 0
```

1.7.2.1 ストリーム・モードでの RMS ファイルへのアクセス

RMS ファイルへのストリーム・アクセスは， RMS のブロック I/O 機能を使用して実行されます。 RMS ファイルからのストリーム入力は，ディスクに格納されているファイルの各バイトをプログラムに渡すことにより実行されます。 RMS ファイルへのストリーム出力は，プログラムからファイルに各バイトを渡すことにより実行されます。HP C RTL はデータに対して特殊な処理を何も実行しません。

ファイルをストリーム・モードでオープンすると，HP C RTL は大きな内部バッファ領域を割り当てます。データは単一読み込みを使用してファイルからバッファ領域に読み込まれ，必要に応じてプログラムに渡されます。内部バッファが満杯になるか，またはfflush関数が呼び出されると，データはファイルに書き込まれます。

1.7.2.2 レコード・モードでの RMS レコード・ファイルへのアクセス

レコード・ファイルへのレコード・アクセスは，RMS のレコード I/O 機能を使用して実行されます。HP C RTL は，レコードの読み込み処理と書き込み処理でキャリッジ制御文字を変換することにより，バイト・ストリームをエミュレートします。すべてのレコード・ファイルに対してランダム・アクセスが可能ですが，VFC ファイル，可変長レコード・ファイル，ヌル以外のキャリッジ制御付きファイルの場合，位置設定 (fseekおよびlseek) はレコード境界で行う必要があります。レコード・ファイルの位置設定を行うと，バッファに格納されているすべての入力が破棄され，バッファに格納されている出力はファイルに書き込まれます。

RMS レコード・ファイルからのレコード入力は，次の 2 つのステップでHP C RTL によってエミュレートされます。

1. HP C RTL はファイルから論理レコードを読み込みます。

   レコード・フォーマットが固定長制御部付可変長 (RFM = VFC) で，レコード属性がプリント・キャリッジ制御でない (RAT が PRN でない) 場合は，HP C RTL は固定長制御領域をレコードの先頭に結合します。

2. HP C RTL はレコードのキャリッジ制御情報 (そのような情報がある場合) を変換することにより，レコードを拡張してバイト・ストリームをシミュレートします。

RMS の用語で表現すると，HP C RTL は次のいずれかの方法を使用して，レコードのキャリッジ制御情報を変換します。

- レコード属性が暗黙のキャリッジ制御 (RAT = CR) の場合，HP C RTL は改行文字をレコードの末尾に追加します。

  この改行文字はレコードの一部であると解釈されます。したがって，たとえばfgetc関数を使用して取得することができ，fgets関数では行区切り文字として解釈されます。fgetsは改行文字までファイルを読み込むため，RAT=CR のファイルの場合，この関数はレコード境界をまたがる文字列を検索することができません。

- レコード属性がプリント・キャリッジ制御 (RAT = PRN) の場合は，HP C RTL はレコードを拡張し，レコードの前後にある先頭と末尾のキャリッジ制御を結合します。

  この変換は，RMS で指定されている規則に従って行われます。ただし，1 つの例外があります。接頭文字が x01 で接尾文字が x8D の場合，レコードの先頭には何も付加されず，レコードの末尾には 1 つの改行文字が付加されます。このような処理が行われるのは，この接頭文字/接尾文字の組み合わせが，通常は行を表現するために使用されるからです。

- レコード属性が Fortran キャリッジ制御 (fRAT = FTN) の場合は，HP C RTL は最初の制御バイトを削除し， RMS の定義に従ってデータの前後に適切なキャリッジ制御文字を付加します。ただし，スペースおよびデフォルトのキャリッジ制御文字は例外です。これらの文字は行を表現するために使用されるので，HP C RTL はデータに 1 つの改行文字を付加します。

  Fortran キャリッジ制御のマッピングは， "ctx=nocvt"を使用することで無効に設定することができます。

- レコード属性がヌル (RAT = NONE) で，入力が端末から取り込まれる場合は，HP C RTL は区切り文字をレコードに追加します。区切り文字がキャリッジ・リターンまたは Ctrl/Z の場合は，HP Cはその文字を改行文字 (\ n) に変換します。

  入力が端末以外のファイルから取り込まれる場合は，HP C RTL はレコードを変更せずにそのままプログラムに渡し，接頭文字や接尾文字は付加しません。

ファイルから読み込む場合，HP C RTL は変換から作成されたバイト・ストリームを渡します。 1 回の関数呼び出しで拡張されたレコードから読み込まれなかった情報は，次回の入力関数呼び出しで渡されます。

HP C RTL は RMS レコード・ファイルに対するレコード出力を次の 2 つのステップで実行します。

レコード出力エミュレーションの最初の部分は，論理レコードの作成です。データ・バイトをレコード・ファイルに書き込む場合，エミュレータは書き込む情報からレコード境界を調べます。バイト・ストリームでの情報の処理は，次に示すように，出力先のファイルやデバイスの属性に応じて異なります。

- どのファイルの場合も，出力されるバイト数がHP C RTL で割り振られた内部バッファより大きい場合は，レコードが出力されます。
- 固定長レコード (RFM = FIX) のファイルや， "ctx=bin"または "ctx=xplct"を指定してオープンされたファイルの場合は，レコードは内部バッファが満杯になるか，または`flush`関数が呼び出された場合にだけ出力されます。
- STREAM_CR レコード・フォーマット (RFM = STMCR) のファイルの場合は，HP C RTL は，キャリッジ・リターン文字 (\ r) を検出したときにレコードを出力します。
- STREAM レコード・フォーマット (RFM = STM) のファイルの場合は，HP C RTL は改行 (\ n)，改ページ (\ f)，垂直タブ (\ v) 文字を検出したときにレコードを出力します。
- 他のすべてのファイル・タイプの場合，HP C RTL は改行文字 (\ n) を検出したときにレコードを出力します。

レコード出力エミュレーションの 2 番目の部分では，最初のステップで作成した論理レコードを書き込みます。HP C RTL は出力レコードを次のように作成します。

- レコード属性がキャリッジ制御 (R AT = CR) で，論理レコードが改行文字 (\ n) で終了する場合は，HP C RTL は改行文字を削除し，暗黙のキャリッジ制御を付けて論理レコードを書き込みます。
- レコード属性がプリント・キャリッジ制御 (RAT = PRN) の場合は，HP C RTL は RMS で指定されている規則に従ってプリント・キャリッジ制御を付けてレコードに書き込みます。論理レコードが 1 文字の改行文字 (\ n) で終了する場合は，改行文字を x01 接頭文字および x8D 接尾文字に変換します。これは，プリント・キャリッジ制御属性の付いたレコード・ファイルを入力する場合と逆の変換です。
- レコード属性が Fortran キャリッジ制御 (RAT = FTN) の場合は，HP C RTL は先頭および末尾のキャリッジ制御文字を削除し， RMS の定義に従ってレコードの先頭に 1 バイトのキャリッジ制御バイトを付加します。ただし， 1 つの例外があります。出力レコードが改行文字 (\ n) で終了する場合は，HP C RTL は改行文字を削除し，スペース・キャリッジ制御バイトを使用します。これは， Fortran キャリッジ制御属性を持つレコード・ファイルを入力する場合の逆の変換です。

  Fortran キャリッジ制御の変換は， "ctx=nocvt"を使用することで無効に設定することができます。
- 論理レコードが端末デバイスに書き込まれ，レコードの最後の文字が改行文字 (\ n) の場合は，HP C RTL は改行文字をキャリッジ・リターン (\ r) に置き換え，レコードの先頭に改行文字 (\ n) を付加します。その後，HP C RTL はキャリッジ制御を付けずにレコードを書き込みます。
- 出力されるファイル・レコード・フォーマットが固定長制御部付可変長 (RFM = VFC) で，レコード属性にプリント・キャリッジ制御が含まれてない (RAT が PRN でない) 場合は，HP C RTL は論理レコードの先頭を固定長制御ヘッダとして解釈し，書き込むバイト数をヘッダの長さだけ少なくします。これらのバイトは固定長制御ヘッダを作成するために使用されます。論理レコードのバイト数が少なすぎる場合はエラーが報告されます。

1.7.2.2.1 レコード・モードでの可変長または VFC レコード・ファイルへのアクセス

レコード・モードで可変長または VFC レコード・ファイルにアクセスする場合，多くの I/O 関数がストリーム・モードの場合と異なる動作をします。ここでは，これらの相違点について説明します。

一般に，どのレコード・モードでも改行文字 (\ n) がレコード区切り文字として使用されます。出力時には，改行文字が検出されると，改行の解釈に影響を与える省略可能な引数 (たとえば "ctx=bin"や "ctx=xplct"など) を指定していない限り，レコードが作成されます。

read関数とdecc$record_read関数は常に最大 1 つのレコードを読み込みます。write関数とdecc$record_write関数は常に少なくとも 1 つのレコードを作成します。

decc$record_read関数はread関数に対応し，decc$record_write関数はwrite関数に対応します。ただし，これらの関数はファイル記述子ではなく，ファイル・ポインタを操作する点が異なります。

read関数は最大 1 つのレコードを読み込みますが，fread関数では複数のレコードにわたる読み込みが可能です。fread関数は，number_itemsによって指定される数のレコードを読み込むのではなく (number_itemsはfreadの 3 番目のパラメータ)，number_items × size_of_itemに等しいバイト数を読み込もうとします (size_of_itemはfreadの 2 番目のパラメータ)。freadから返される値は，読み込んだバイト数をsize_of_itemで除算した値に等しくなります。

fwrite関数は常に少なくともnumber_itemsで指定される数のレコードを作成します。

fgets関数とgets関数は改行文字またはレコード境界まで読み込みます。

バッファに書き込まなければならないデータがある場合，fflush関数は常にレコードを生成します。close，fclose，fseek，lseek，rewind，fsetposの場合も同様で，これらの関数はすべて，暗黙にfflush関数を実行します。

最大レコード・サイズに指定されている文字より多くの文字を書き込もうとした場合も，レコードが生成されます。

これらの関数の詳細については，『HP C ランタイム・ライブラリ・リファレンス・マニュアル (下巻)』「リファレンス・セクション」を参照してください。

1.7.2.2.2 レコード・モードでの固定長レコード・ファイルへのアクセス

レコード・モードで固定長レコード・ファイルにアクセスする場合， I/O 関数は一般に，第 1.7.2.2.1 項で説明したように動作します。

省略可能な引数 "ctx=xplct"を指定してファイルをオープンした場合を除き，指定したレコード・サイズが最大レコード・サイズの整数倍でないと，write関数，fwrite関数，decc$record_write関数はエラーになります。他のすべての出力関数はnバイトごとにレコードを生成します。ただし，nは最大レコード・サイズです。

fflushによって新しいレコードが生成される場合，最大レコード・サイズになるようにバッファ内のデータにヌル文字が付加されます。

注意

ファイルの終端 (EOF) を検索するプログラムの場合，このヌル文字の付加が問題になることがあります。たとえば，プログラムでファイルの終端にデータを追加した後，ファイルを逆向きにシークし (fflushが実行されます)，その後で再びファイルの終端を検索すると，元のファイルの終端がレコード境

界上になかった場合，元のファイルの終端と新しいファイルの終端の間に，0 が埋め込まれた「穴」が作成されています。

1.7.2.3 例 — ストリーム・モードとレコード・モードの違い

例 1–1 は，ストリーム・モードとレコード・モードのアクセスの相違点を示しています。

例 1–1 ストリーム・モードとレコード・モードのアクセスの相違点

```
/*      CHAP_1_STREAM_RECORD.C                      */

/* This program demonstrates the difference between  */
/* record mode and stream mode input/output.         */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void process_records(const char *fspec, FILE * fp);

main()
{
    FILE *fp;

    fp = fopen("example-fixed.dat", "w", "rfm=fix", "mrs=40", "rat=none");
    if (fp == NULL) {
        perror("example-fixed");
        exit(EXIT_FAILURE);
    }
    printf("Record mode\n");
    process_records("example-fixed.dat", fp);
    fclose(fp);

    printf("\nStream mode\n");
    fp = fopen("example-streamlf.dat", "w");
    if (fp == NULL) {
        perror("example-streamlf");
        exit(EXIT_FAILURE);
    }
    process_records("example-streamlf.dat", fp);
    fclose(fp);
}

void process_records(const char *fspec, FILE * fp)
{
    int i,
        sts;

    char buffer[40];
```

(次ページに続く)

例 1–1 (続き) ストリーム・モードとレコード・モードのアクセスの相違点

```
    /* Write records of all 1's, all 2's and all 3's */
    for (i = 0; i < 3; i++) {
        memset(buffer, '1' + i, 40);
        sts = fwrite(buffer, 40, 1, fp);
        if (sts != 1) {
            perror("fwrite");
            exit(EXIT_FAILURE);
        }
    }

    /* Rewind the file and write 10 characters of A's, then 10 B's, */
    /* then 10 C's.                                                 */
    /*                                                              */
    /* For stream mode, each fwrite call outputs 10 characters      */
    /* and advances the file position 10 characters                 */
    /* characters.                                                  */
    /*                                                              */
    /* For record mode, each fwrite merges the 10 characters  into  */
    /* the existing 40-character record, updates the record and     */
    /* advances the file position 40 characters to the next record. */
    rewind(fp);
    for (i = 0; i < 3; i++) {
        memset(buffer, 'A' + i, 10);
        sts = fwrite(buffer, 10, 1, fp);
        if (sts != 1) {
            perror("fwrite2");
            exit(EXIT_FAILURE);
        }
    }

    /* Now reopen the file and output the records. */

    fclose(fp);
    fp = fopen(fspec, "r");
    for (i = 0; i < 3; i++) {
        sts = fread(buffer, 40, 1, fp);
        if (sts != 1)
            perror("fread");
            printf("%.40s\n", buffer);
    }

    return;
}
```

このプログラムを実行すると，次の出力が生成されます。

```
Record Mode
AAAAAAAAAA1111111111111111111111111111
BBBBBBBBBB2222222222222222222222222222
CCCCCCCCCC3333333333333333333333333333
```

```
Stream mode
AAAAAAAAAABBBBBBBBBBCCCCCCCCCC1111111111
22222222222222222222222222222222222222222
33333333333333333333333333333333333333333
```

1.8 特定の移植性に関する問題

複数のシステム間でソース・プログラムを移植する予定がある場合，HP C RTL を使用するための最後の準備作業の 1 つとして，HP C RTL と， C 言語の他の実装のランタイム・ライブラリの相違点を認識することが必要です。ここでは， OpenVMS システムとの間でプログラムを移植するときに発生する可能性のある問題の一部について説明します。移植性はHP C RTL の実装に密接に関係していますが，ここでは，他のHP C for OpenVMS 構造体の移植性についても説明します。

HP C RTL では，ANSI Cで定義されているライブラリ関数をはじめ，一般に使用されている多くの API や，若干の OpenVMS 拡張機能も提供されます。特定の標準のうち，HP C RTL で実装されている部分については，第 1.4 節を参照してください。可能な限り機能面で完全な移植性を維持するようになっています。HP C RTL で提供される多くの標準 I/O および UNIX I/O の関数およびマクロは，他の実装の関数やマクロに機能的に対応します。

RTL 関数およびマクロの説明では，ここに示した問題の他に，ここに示していない問題についても詳しく説明しています。

次の一覧は， C プログラムを OpenVMS 環境に移植するときに考慮しなければならない問題点を示しています。

- HP C for OpenVMS Systems はグローバル・シンボル`end`，`edata`，`etext`を実装していません。
- OpenVMS システムと UNIX システムが仮想メモリをレイアウトする方法は異なっています。一部の UNIX システムでは， 0 からブレーク・アドレスまでの間のアドレス空間はユーザ・プログラムでアクセスできます。 OpenVMS システムでは，メモリの最初のページにはアクセスできません。

  たとえば，プログラムで OpenVMS システムのロケーション 0 を参照しようとすると，ハードウェア・エラー (ACCVIO) が返され，プログラムは異常終了します。 OpenVMS システムでは，ヌル・ポインタによって指されているロケーションへの参照など，不正なポインタ参照を検知するために，アドレス空間の最初のページが確保されています。この理由から，一部の UNIX システムで動作する既存のプログラムは，エラーになる可能性があり，必要に応じて変更する必要があります (しかし，この点に関して Tru64 UNIX システムと OpenVMS システムは互換性があります)。

- 一部のCプログラムでは，#includeファイルにすべての外部宣言が指定されることがあります。その後，初期化が必要な特定の宣言は関連モジュールで再び宣言されます。この方法でプログラミングした場合，HP Cコンパイラは同じコンパイルで2回以上宣言された変数があることに関する警告メッセージを出力します。この警告を回避するための1つの方法として，#includeファイルで再宣言されるシンボルをextern変数に設定することができます。
- OpenVMS VAXシステムとOpenVMS Integrityシステムで，HP Cはasm呼び出しをサポートしていません。 OpenVMS Alphaシステムでは，これらの呼び出しはサポートされます。組み込み関数の詳細については，『HP C User's Guide for OpenVMS Systems』を参照してください。
- 一部のCプログラムでは，カウント付き文字列関数strcmpnおよびstrcpynが呼び出されます。これらの名前はHP C for OpenVMS Systemsでは使用されません。その代わり，strcmpnおよびstrcpynの名前を，それに対応するANSI準拠の名前strncmpおよびstrncpyに拡張するマクロを定義することができます。
- HP C for OpenVMSコンパイラでは，次の初期化形式はサポートされません。

  ```
  int  foo  123;
  ```

  この初期化形式を使用するプログラムは変更する必要があります。
- HP C for OpenVMS Systemsでは，__vax，__alpha，__ia64 __32BITS，__vms，__vaxc，__VMS_VER，__DECC_VER，__D_FLOAT，__G_FLOAT，__IEEE_FLOAT，__X_FLOATなどの複数のコンパイル時マクロをあらかじめ定義しています。これらの定義済みマクロは，他のマシンやオペレーティング・システムと互換性を維持しなければならないプログラムにとって便利です。詳細については，『HP C User's Guide for OpenVMS Systems』の定義済みマクロの章を参照してください。
- ANSI C言語では，宣言で変数のメモリ順序が保証されません。次の例を参照してください。

  ```
  int  a, b, c;
  ```
- 要求される外部リンクのタイプに応じて，プログラム内のextern変数は，OpenVMSシステムでHP Cを使用する場合， UNIXシステムの場合と異なる方法で取り扱われることがあります。詳細については，『HP C User's Guide for OpenVMS Systems』を参照してください。
- ドル記号($)は，HP C for OpenVMSの識別子で使用できる文字であり， 1文字目として使用できます。
- ANSI C言語では，関数パラメータ・リストや他の多くの式で式の評価の順序を定義していません。各Cコンパイラで式を評価する方法は，その式が副作用を持つ場合にだけ重要です。次の例について考えてみましょう。

```
a[i]  =  i++;

x = func_y() + func_z();

f(p++, p++)
```

HP Cでも他のC コンパイラでも，このような式がすべてのC コンパイラで同じ順序で評価されるという保証はありません。

- int型のHP C変数のサイズは， OpenVMS システムでは32 ビットです。他のマシン用に作成され，int型の変数のサイズが異なるサイズであると仮定されているプログラムは変更する必要があります。long型の変数は，int型の変数と同じサイズ(32 ビット) です。
- C 言語では，コンパイラが設計されたマシンに依存する構造体アラインメントを定義しています。 OpenVMS Alpha システムでは，#pragma nomember_alignmentが指定されている場合を除き，HP Cは構造体メンバを自然境界に揃えます。他の実装では，構造体メンバを異なる方法でアラインメントすることがあります。
- HP Cでの構造体メンバへの参照はあいまいにすることができません。詳細については，『HP C Language Reference Manual』を参照してください。
- レジスタは変数が使用される頻度に応じて割り当てられますが，registerキーワードは，特定の変数をレジスタに格納するかどうかに関して強力なヒントをコンパイラに与えます。可能な場合は，変数はレジスタに格納されます。ストレージ・クラスがautoまたはregisterのスカラ変数は，変数のアドレスがアンパサンド演算子(&)で参照されることがなく，構造体やユニオンのメンバでない限り，レジスタに割り当てることができます。

1.8.1 リエントラント

HP C RTL では，リエントラントのサポートが向上し，強化されました。次の種類のリエントラントがサポートされます。

- AST リエントラントでは，_BBSSI 組み込み関数を使用して， RTL コードのクリティカル・セクションの周囲で単純なロックを実行しますが，ロックされたコード領域で非同期システム・トラップ (AST) も必要になることがあります。この種のロックは， AST コードにHP C RTL I/O ルーチンへの呼び出しが含まれているときに使用しなければなりません。

  AST リエントラントを指定しないと， I/O ルーチンが異常終了することがあり，errnoが EALREADY に設定されます。
- MULTITHREAD リエントラントは， DECthreads ライブラリを使用するプログラムなど，スレッド・プログラムで使用するように設計されています。このリエントラントは DECthreads ロックを使用し， AST を無効にしません。この形式のリエントラントを使用するには，システムで DECthreads が使用可能でなければなりません。

- TOLERANT リエントラントでは，_BBSSI 組み込み関数を使用して， RTL コードのクリティカル・セクションの周囲で単純なロックを実行しますが， AST は禁止されません。この種のロックは， AST が使用され，ただちに配布しなければならないときに使用する必要があります。
- NONE はHP C RTL で最適な性能を提供しますが， RTL コードのクリティカル・セクションの周囲で絶対にロックを行いません。実行スレッドがHP C RTL を呼び出す AST によって割り込まれる可能性がない場合は，シングル・スレッド環境でのみ使用するようにしなければなりません。

デフォルトのリエントラント・タイプは TOLERANT です。

リエントラント・タイプを設定するには，/REENTRANCY コマンド・ライン修飾子を指定してコンパイルするか，decc$set_reentrancy関数を呼び出します。この関数は非 AST レベルから排他的に呼び出さなければなりません。

複数のスレッドまたは AST を使用するアプリケーションをプログラミングする場合は，次の 3 つのクラスの関数について考慮してください。

- 内部データのない関数
- スレッドだけで有効な内部データのある関数
- プロセス単位の内部データがある関数

ほとんどの関数は，まったく内部データのない関数です。このような関数の場合，同期化が必要になるのは，パラメータがマルチスレッドのアプリケーションで使用されるか，または AST コンテキストと非 AST コンテキストの両方でパラメータが使用される場合だけです。たとえば，strcat関数は一般にはスレッド・セーフですが，次の例は安全でない使い方を示しています。

```
extern char buffer[100];
void routine1(char *data) {
    strcat( buffer, data );
}
```

routine1が複数のスレッドで並列に実行されるか，またはroutine1がそのルーチンを呼び出す AST ルーチンによって割り込まれると，strcat呼び出しの結果は予測不能になります。

2 番目のクラスの関数は，スレッドだけで有効な静的データを含む関数です。通常，これらの関数は，アプリケーションが文字列の格納領域を解放することを許可されていない状況で，文字列を返すライブラリ内のルーチンです。これらのルーチンはスレッド・セーフですが， AST リエントラントではありません。つまり，並列に呼び出しても安全ですが，各スレッドはそれぞれ独自のデータのコピーを保有します。同じルーチンが非 AST コンテキストで実行される可能性がある場合は，これらのルーチンを AST ルーチンから呼び出すことはできません。このクラスのルーチンは次のとおりです。

```
asctime         stat
ctermid         strerror
ctime           strtok
cuserid         VAXC$ESTABLISH
gmtime          the errno variable
localtime       wcstok
perror
```

使用している TCP/IP 製品がスレッド・セーフの場合，すべてのソケット関数もこのリストに含まれます。

3 番目のクラスの関数は，プロセス単位のデータに影響する関数です。これらの関数はスレッド・セーフではなく， AST リエントラントでもありません。たとえば，sigsetmaskはプロセス単位のシグナル・マスクを確立します。次のようなルーチンについて考えてみましょう。

```
void update_data
base()
{
    int old_mask;

    old_mask = sigsetmask( 1 << (SIGINT - 1));
        /* Do work here that should not be aborted. */
    sigsetmask( old_mask );
}
```

update_databaseが複数のスレッドで並列して呼び出された場合，スレッド 2 が強制終了されない作業をまだ実行している間に，スレッド 1 は SIGINT のブロックを解除する可能性があります。

このクラスのルーチンは次のとおりです。

- すべてのsignalルーチン
- すべてのexecルーチン
- exit，_exit，nice，system，wait，getitimer，setitimer，setlocaleルーチン

注意

一般に， UTC **ベースの時刻関数はメモリ内のタイム・ゾーン情報に影響を与える可能性があり，これはプロセス単位のデータです。しかし，アプリケーションが実行されている間，システム・タイム・ゾーンが変化せず (これは一般的な場合です)，タイム・ゾーン・ファイルのキャッシュが許可されている場合 (これはデフォルトです)，時刻関数**asctime_r，ctime_r，gmtime_r，localtime_r**の_r バリアントはスレッド・セーフであり，かつ** AST **リエントラントです。**

しかし，アプリケーションの実行中にシステム・タイム・ゾーンが変化する可能性がある場合や，タイム・ゾーン・ファイルのキャッシュが許可されていない場合は， UTC **ベースの時刻関数のバリアントはどちらも** 3 **番目のクラスの関数に属し，スレッド・セーフでも** AST **リエントラントでもありません。**

次に示す一部の関数は，リエントラントであるかどうかに関係なく，本質的にスレッド・セーフではありません。

```
execl       exit
execle      _exit
execlp      nice
execv       system
execve      vfork
execvp
```

1.8.2 マルチスレッドの制限事項

同じアプリケーション内でマルチスレッド・プログラミング・モデルと OpenVMS AST プログラミング・モデルを混在させることは推奨できません。アプリケーションで，どのスレッドが AST によって割り込まれるかを制御することはできません。この結果， AST ルーチンからも必要とされるリソースをスレッドが保有している場合，リソースのデッドロックが発生します。次の関数はミューテクスを使用します。リソース・デッドロックの発生を回避するには，マルチスレッド・アプリケーションで AST 関数からこれらを呼び出さないようにしなければなりません。

- すべての I/O 関数
- すべてのソケット関数
- すべてのシグナル関数
- vfork，exec，wait，system
- catgets
- set_new_handler (C++ のみ)
- getenv
- randとsrand
- exitと_exit
- clock
- nice
- times
- ctime，localtime，asctime，mktime

1.9 64 ビット・ポインタのサポート *(Integrity，Alpha)*

このセクションの説明は， OpenVMS Alpha Version 7.0 以降で 64 ビット仮想メモリ・アドレッシングを使用する必要のあるアプリケーション開発者を対象にしています。

OpenVMS Alpha の 64 ビット仮想アドレッシングのサポートでは，Alpha アーキテクチャで定義されている 64 ビット仮想アドレス空間が OpenVMS オペレーティング・システムとそのユーザの両方から使用できるようになっています。また，従来の 32 ビットの制限を超えて動的にマッピングされたデータにアクセスするために，プロセス単位の仮想アドレッシングも可能です。

OpenVMS Alpha Version 7.0 以降のシステムのHP Cランタイム・ライブラリでは，64 ビット・ポインタをサポートするために次の機能が提供されます。

- 既存のプログラムとの間で，バイナリ・レベルおよびソース・レベルの互換性が保証されます。
- 64 ビットのサポート機能を活用するように変更されていないアプリケーションには影響を与えません。
- 64 ビット・メモリを割り振るメモリ割り当てルーチンの機能が拡張されています。
- 64 ビット・ポインタに対応するために，関数パラメータのサイズが拡大されました。
- 呼び出し元が使用するポインタ・サイズに関する情報が必要な関数は，二重に実装されています。
- 適切な実装を呼び出すことができるように，DEC C Version 5.2 以降のコンパイラで新しい情報が提供されるようになりました。
- ポインタ・サイズが混在するアプリケーションで，32 ビット形式と 64 ビット形式の関数を明示的に呼び出すことができる機能が提供されます。
- 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使用するために，1 つの共用可能イメージが提供されます。

1.9.1 HP C ランタイム・ライブラリの使用

OpenVMS Alpha Version 7.0 以降のシステムでは，HP Cランタイム・ライブラリは 64 ビット・ポインタを生成したり，受け付けることができます。64 ビット・ポインタで使用される第 2 のインタフェースを必要とする関数は，対応する 32 ビットの関数と同じオブジェクト・ライブラリおよび共用可能イメージに存在します。新しいオブジェクト・ライブラリや共用可能イメージが提供されるわけではありません。64 ビット・ポインタを使用する場合，リンク・コマンドやリンク・オプション・ファイルを変更する必要はありません。

HP Cの 64 ビット環境では，アプリケーションは 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・サイズの操作方法の詳細については，『HP C User's Guide for OpenVMS Systems』の/POINTER_SIZE 修飾子と，`#pragma pointer_size`および`#pragma required_pointer_size`プリプロセッサ・ディレクティブを参照してください。

/POINTER_SIZE 修飾子には， 32 または 64 の値を指定する必要があります。この値は，コンパイル・ユニットの内部でデフォルト・ポインタ・サイズとして使用されます。 32 ビット・ポインタを使用して 1 組のモジュールをコンパイルし， 64 ビット・ポインタを使用して別の 1 組のモジュールをコンパイルすることができます。これらの 2 組のモジュールを相互に呼び出す場合は，注意を払う必要があります。

/POINTER_SIZE 修飾子の使用は，HP C RTL ヘッダ・ファイルの処理にも影響を与えます。 32 ビット版と 64 ビット版の両方がある関数の場合，/POINTER_SIZE を指定すると，修飾子に指定された実際の値とは無関係に，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，修飾子に指定された値によって，そのコンパイル・ユニットで呼び出すデフォルトの実装が決定されます。

#pragma pointer_sizeおよび#pragma required_pointer_sizeプリプロセッサ・ディレクティブを使用すると，コンパイル・ユニットの内部で有効なポインタ・サイズを変更することができます。デフォルトのポインタ・サイズを 32 ビット・ポインタに設定した後，モジュール内で特定のポインタを 64 ビット・ポインタとして宣言することができます。また， 64 ビット・メモリ領域からメモリを取得するには，mallocの_malloc64という形式を特別に呼び出す必要もあります。

1.9.2 メモリへの 64 ビット・ポインタの取得

HP C RTL には，新たに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。これらの各関数では，アプリケーションはポインタによって示されるメモリを所有し，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のとおりです。

```
malloc
calloc
realloc
strdup
```

これらの各関数には， 32 ビット版と 64 ビット版があります。/POINTER_SIZE 修飾子を使用すると，次の関数も呼び出すことができます。

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

/POINTER_SIZE=32 を指定した場合，すべてのmallocの呼び出しはデフォルトで_malloc32に設定されます。

/POINTER_SIZE=64 を指定した場合，すべてのmallocの呼び出しはデフォルトで_malloc64に設定されます。

アプリケーションで 32 ビットのメモリ割り当てルーチンを呼び出す場合も，64 ビットのメモリ割り当てルーチンを呼び出す場合も，free関数は 1 つしかありません。この関数はどちらのポインタ・サイズも受け付けます。

64 ビット・メモリを指すポインタを返すのは，メモリ割り当て関数だけであるということに注意しなければなりません。呼び出し元のアプリケーションに返されるすべてのHP C RTL 構造体ポインタ (FILE，WINDOW，DIR など) は常に 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元アプリケーションがこれらの構造体ポインタをアプリケーションの内部で渡すことができます。

1.9.3 HP C ヘッダ・ファイル

OpenVMS に付属のヘッダ・ファイルでは，64 ビット・ポインタがサポートされます。シグネチャにポインタが含まれる各関数プロトタイプは，受け付けるポインタのサイズを示すように作成されています。

32 ビット・ポインタは，引数として 32 ビットまたは 64 ビットのポインタを受け付ける関数に対して，引数として渡すことができます。

しかし，32 ビット・ポインタを受け付ける関数に対する引数として，64 ビット・ポインタを渡すことはできません。このような処理はコンパイラで診断され，MAYLOSEDATA メッセージが出力されます。診断メッセージ IMPLICITFUNC は，コンパイラがその関数の呼び出しでポインタ・サイズの追加確認を実行できないことを示します。この関数がHP C RTL 関数の場合は，その関数を定義しているヘッダ・ファイルの名前については，『HP C ランタイム・ライブラリ・リファレンス・マニュアル (下巻)』「リファンレンス・セクション」を参照してください。

ポインタ・サイズに関する次のコンパイラ診断情報は役立ちます。

- %CC-IMPLICITFUNC

  指定された関数を使用する前に，関数プロトタイプを見つけることができませんでした。コンパイラおよび実行時システムは，プロトタイプの定義をもとに，不正なポインタ・サイズの使用を検出します。適切なヘッダ・ファイルを取り込むことができないと，不正な結果が発生したり，ポインタが切り捨てられる可能性があります。

- %CC-MAYLOSEDATA

  この操作を実行するには，切り捨てが必要です。この操作では，指定されたコンテキストで 64 ビット・ポインタをサポートしていない関数に 64 ビット・ポインタを渡す可能性があります。また，32 ビット・ポインタに戻り値を格納しようとする呼び出し元アプリケーションに，関数が 64 ビット・ポインタを返している可能性もあります。

- %CC-MAYHIDELOSS

このメッセージ (有効に設定されている場合) は，キャスト操作のために出力されていない実際の MAYLOSEDATA メッセージを確認するのに役立ちます。この警告を有効にするには，修飾子/WARNINGS=ENABLE=MAYHIDELOSS を指定してコンパイルします。

1.9.4 影響を受ける関数

HP C RTL は，32 ビット・ポインタのみ，64 ビット・ポインタのみ，またはその両方の組み合わせを使用するアプリケーションに対応します。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。必要なソース・コードの変更の量は，アプリケーション自体，他のランタイム・ライブラリの呼び出し，使用するポインタ・サイズの組み合わせに応じて異なります。

64 ビット・ポインタのサポートに関して，HP C RTL 関数は次の 4 種類に分類できます。

- ポインタ・サイズの選択の影響を受けない関数
- どちらのポインタ・サイズも受け付けるように拡張された関数
- 32 ビット版と 64 ビット版がある関数
- 32 ビット・ポインタだけを受け付ける関数

アプリケーション開発者の立場から考えると，最初の 2 種類の関数は単一ポインタ・モードまたは複合ポインタ・モードで最も簡単に使用できます。

3 番目の関数は，1 種類のポインタだけを使用してコンパイルする場合は変更が不要ですが，複合ポインタ・モードで使用する場合はソース・コードの変更が必要です。

4 番目の関数は，64 ビット・ポインタを使用しなければならないときに注意する必要があります。

1.9.4.1 ポインタ・サイズの影響を受けない関数

プロトタイプにポインタ関連パラメータや戻り値が含まれていない場合は，ポインタ・サイズの選択は関数に影響を与えません。このような関数の例として，算術演算関数があります。

このカテゴリに分類される関数で，プロトタイプにポインタを含む一部の関数もポインタ・サイズの影響を受けません。たとえば，strerrorには，次のプロトタイプがあります。

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

この関数は文字列を指すポインタを返しますが，この文字列はHP C RTL によって割り振られます。この結果，32 ビット・アプリケーションと 64 ビット・アプリケーションの両方をサポートするために，これらのポインタ・タイプは 32 ビット・ポインタに収まるように保証されます。

1.9.4.2 両方のポインタ・サイズを受け付ける関数

Alpha アーキテクチャでは 64 ビット・ポインタがサポートされます。 OpenVMS Alpha の呼び出し標準規約では，すべての引数は実際に 64 ビットの値として渡されることが指定されています。 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すべての 32 ビット・アドレスは，この 64 ビット・パラメータになるように符号拡張されていました。呼び出される関数は 32 ビット・アドレスとしてパラメータを宣言し，コンパイラはこれらのパラメータを操作するために 32 ビットの命令 (LDL など) を生成していました。

HP C RTL の多くの関数は，完全な 64 ビット・アドレスを受け付けることができるように拡張されています。たとえば，strlenについて考えてみましょう。

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

この関数のポインタは文字列ポインタだけです。呼び出し元が 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・アドレスを操作します。呼び出し元が 64 ビット・アドレスを渡した場合は，関数はそのアドレスを直接操作します。

HP C RTL は，このカテゴリに分類される関数に対して，今後もエントリ・ポイントを 1 つだけ使用します。この種の関数に対して 4 種類のポインタ・サイズ・オプションを追加するためにソース・コードを変更する必要はありません。 OpenVMS のドキュメントでは，このような関数を「64 ビットと親和性がある関数」と呼んでいます。

1.9.4.3 2 つの実装のある関数

多くの理由から， 1 つの関数に対して 32 ビット・ポインタを取り扱う実装と 64 ビット・ポインタを取り扱う実装を用意しなければならないことがあります。たとえば，次のような場合が考えられます。

- 戻り値のポインタ・サイズが引数のいずれかのポインタ・サイズと同じサイズである場合。たとえば，引数が 32 ビットの場合は，戻り値は 32 ビット，引数が 64 ビットの場合は，戻り値は 64 ビットになります。
- 引数のいずれかが，ポインタ・サイズに敏感に反応するオブジェクトを指すポインタである場合。指し示すバイト数を知るためには，関数はコードが 32 ビット・ポインタ・サイズ・モードでコンパイルされたのか， 64 ビット・ポインタ・サイズ・モードでコンパイルされたのかを知る必要があります。
- 関数が動的に割り当てられたメモリのアドレスを返す場合。 32 ビット・ポインタに対してコンパイルされた場合は，メモリは 32 ビット空間に割り当てられ， 64 ビット・ポインタに対して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考えると，これらの各関数に対して 3 つの関数プロトタイプがあります。<string.h>ヘッダ・ファイルには，戻り値が関数呼び出しの最初の引数として使用されたポインタ・サイズに依存する多くの関数が含まれています。たとえば，memset関数について考えてみましょう。ヘッダ・ファイルはこの関数に対して 3 つのエントリ・ポイントを定義しています。

```
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，この関数を使用したときにアプリケーションが現在呼び出している関数です。/POINTER_SIZE=32 を使用してコンパイルした場合は，コンパイラはmemsetに対する呼び出しを_memset32に変更し，/POINTER_SIZE=64 を指定してコンパイルした場合は，_memset64に変更します。

このデフォルトの動作は，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変更できます。この機能は，/POINTER_SIZE 修飾子で指定したデフォルト・ポインタ・サイズとは無関係に，複合ポインタ・サイズを使用するアプリケーションに対応します。

/POINTER_SIZE 修飾子を**指定せずに**アプリケーションをコンパイルした場合は，32 ビット固有の関数プロトタイプも， 64 ビット固有の関数プロトタイプも**定義されません**。この場合，コンパイラは 2 種類の実装のあるすべての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。

表 1–5 は， 64 ビット・ポインタ・サイズをサポートするために 2 種類の実装を用意しているHP C RTL 関数を示しています。/POINTER_SIZE 修飾子を使用してコンパイルした場合，変更されていない関数名の呼び出しは，修飾子に指定したポインタ・サイズに対応する関数インタフェースに変更されます。

表 1–5　2 種類の実装が用意されている関数

basename	bsearch	calloc	catgets
ctermid	cuserid	dirname	fgetname
fgets	fgetws	gcvt	getcwd
getname	getpwent	getpwnam	getpwnam_r
getpwuid	getpwuid_r	gets	index
longname	malloc	mbsrtowcs	memccpy
memchr	memcpy	memmove	memset
mktemp	mmap	qsort	readv
realloc	rindex	strcat	strchr
strcpy	strdup	strncat	strncpy
strpbrk	strptime	strrchr	strsep
strstr	strtod	strtok	strtok_r
strtol	strtoll	strtoq	strtoul
strtoull	strtouq	tmpnam	wcscat

(次ページに続く)

表 1–5 (続き) 2 種類の実装が用意されている関数

wcschr	wcscpy	wcsncat	wcsncpy
wcspbrk	wcsrchr	wcsrtombs	wcsstr
wcstok	wcstol	wcstoul	wcswcs
wmemchr	wmemcpy	wmemmove	wmemset
writev	glob	globfree	

表 1–6 に，64 ビット・ポインタ・サイズをサポートするために 2 種類の実装が用意されている TCP/IP ソケット・ルーチンを示します。

表 1–6 2 種類の実装が用意されているソケット・ルーチン

freeaddrinfo	getaddrinfo
recvmsg	sendmsg

1.9.4.4 64 KB を超えるソケットの転送

OpenVMS Version 8.3 以降，次のソケット・ルーチンで 64 KB を超えるソケットの転送をサポートしています。

send	recv	read
sendto	recvfrom	write
sendmsg	recvmsg	

1.9.4.5 64 ビット構造体を明示的に使用する必要がある関数

いくつかの関数では，/POINTER_SIZE=LONG でコンパイルする際に，明示的に 64 ビット構造体を使用する必要があります。最近 64 ビット・サポートが追加された関数では，不注意により 32 ビット版の構造体と 64 ビット版の構造体が混在した場合に，予期しない実行時エラーが起こるのを防ぐためにこのようにする必要があります。

次のような関数について考えて見ましょう。

getaddrinfo	getpwnam
freeaddrinfo	getpwnam_r
getpwuid	getpwent
sendmsg	getpwent_r
recvmsg	

これらの関数は，以前はたとえ/POINTER_SIZE=LONG を指定してコンパイルしても 32 ビットのみサポートしていました。/POINTER_SIZE=LONG でコンパイルした場合もこれらの関数で以前の 32 ビット・ポインタ・サポートの動作を保つようにするため，これらの 7 つの関数は，前の項で説明した 32 ビットと 64 ビットのサポートのための通常の規則には従いません。

これらの関数の次のような変形とそれらが対応する構造体が， 64 ビット・サポートのために C RTL に追加されています。

Function	Structure
__getaddrinfo32	__addrinfo32
__getaddrinfo64	__addrinfo64
__freeaddrinfo32	__addrinfo32
__freeaddrinfo64	__addrinfo64
__recvmsg32	__msghdr32
__recvmsg64	__msghdr64
__sendmsg32	__msghdr32
__sendmsg64	__msghdr64
__32_getpwnam	__passwd32
__64_getpwnam	__passwd64
_getpwnam_r32	__passwd32
_getpwnam_r64	__passwd64
__32_getpwuid	__passwd32
__64_getpwuid	__passwd64
_getpwuid_r32	__passwd32
_getpwuid_r64	__passwd64
__32_getpwent	__passwd32
__64_getpwent	__passwd64

これらの関数の標準バージョンをコンパイルする場合は，次のような動作が発生します。

- /POINTER_SIZE=32 を指定すると，コンパイラは， 32 ビット・バージョンの関数呼び出しに変換します。たとえば，getaddrinfoは__getaddrinfo32に変換されます。
- /POINTER_SIZE=64 を指定すると，コンパイラは， 64 ビット・バージョンの関数呼び出しに変換します。たとえば，getaddrinfoは__getaddrinfo64に変換されます。
- /POINTER_SIZE 修飾子を指定しなかった場合， 32 ビット固有関数プロトタイプも 64 ビット固有関数プロトタイプも定義されません。

これらの関数では，対応する構造体に関して同様の変換は行われません。この動作は，これらの構造体が OpenVMS Version 7.3-2 までは/POINTER_SIZE=LONG でコンパイルした場合も 32 ビット・バージョンとしてのみ存在していたために必要となります。構造体のサイズを変更すると，予測できない実行時エラーが発生することになります。

これらの関数の標準バージョンを使用するプログラムを 64 ビット・サポートのためにコンパイルする場合，関連する構造体の 64 ビット固有定義を使用する必要があります。標準の関数名および標準の構造体定義を使用するプログラムを/POINTER_SIZE=64 を指定してコンパイルすると，コンパイラ PTRMISMATCH 警告メッセージが発生します。

たとえば次のプログラムは，addrinfo構造体の標準の定義とともにgetaddrinfoおよびfreeaddrinfo ルーチンを使用しています。このプログラムをコンパイルすると，次のような警告メッセージが表示されます。

```
$ type test.c
#include <netdb.h>

int main ()
{
    struct addrinfo *ai;

    getaddrinfo ("althea", 0, 0, &ai);
    freeaddrinfo (ai);
    return 0;
}

$ cc /pointer_size=64 TEST.C

    getaddrinfo ("althea", 0, 0, &ai);
....^
%CC-W-PTRMISMATCH, In this statement, the referenced type of the pointer value
"&ai" is "long pointer to struct addrinfo", which is not compatible with "long
pointer to struct __addrinfo64".
at line number 7 in file TEST.C;1

    freeaddrinfo (ai);
....^
%CC-W-PTRMISMATCH, In this statement, the referenced type of the pointer value
"ai" is "struct addrinfo", which is not compatible with "struct __addrinfo64".
at line number 8 in file TEST.C;1
$
```

64 ビット用にコンパイルする場合は， 64 ビット固有バージョンの構造体を使用する必要があります。前の例のai構造体の宣言を次のように変更します。

```
struct __addrinfo64 *ai;
```

あるいは， 32 ビットと 64 ビットで柔軟にコンパイルできるようにするために，次のようにai構造体を宣言します。

```
#if __INITIAL_POINTER_SIZE == 64
    struct __addrinfo64 *ai;
#else
    struct __addrinfo32 *ai;
#endif
```

1.9.4.6 32 ビット・ポインタに制限される関数

HP C RTL の中の若干の関数は 64 ビット・ポインタをサポートしません。これらの関数に 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 警告を生成します。/POINTER_SIZE=64 を指定してコンパイルしたアプリケーションでは， 64 ビット・ポインタをこれらの関数に渡さないように変更する必要があります。

表 1–7 は， 32 ビット・ポインタに制限される関数を示しています。HP C RTL では，これらの関数に対して 64 ビットのポインタはサポートされません。これらの関数では 32 ビット・ポインタだけを使用するように注意しなければなりません。

表 1–7 32 ビット・ポインタに制限される関数

atexit	getopt	putenv
execv	iconv	setbuf
execve	initstate	setstate
execvp	ioctl	setvbuf

表 1–8 は，関数呼び出しの処理の一部としてユーザ指定関数に対するコールバックを作成する関数を示しています。コールバック・プロシージャには 64 ビット・ポインタは渡されません。

表 1–8 32 ビット・ポインタのみを渡すコールバック

decc$from_vms	decc$to_vms
ftw	tputs

1.9.5 ヘッダ・ファイルの読み込み

ここでは，HP C RTL ヘッダ・ファイルで使用されるポインタ・サイズの操作について説明します。これらのヘッダ・ファイルの読み込みについてより理解を深め，各自のヘッダ・ファイルを変更するのに役立つように，次の例を使って説明します。

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HP Cコンパイラで

は，__INITIAL_POINTER_SIZE マクロがあらかじめ定義されています。HP C RTL ヘッダ・ファイルは，マクロが定義されていない場合，暗黙の値として 0 を使用するという ANSI の規則に従います。

/POINTER_SIZE 修飾子が使用された場合は，マクロは 32 または 64 として定義されます。修飾子が使用されない場合は，マクロは 0 として定義されます。**1** に指定されている文は，「ユーザが/POINTER_SIZE=32 または/POINTER_SIZE=64 をコマンド・ラインに指定した場合」と考えることができます。

OpenVMS の多くのバージョンでは， C コンパイラがサポートされます。**2** に示した行は，コンパイラの対象が 64 ビット・ポインタをサポートしないプラットフォームの場合，エラー・メッセージを生成します。

ヘッダ・ファイルでは，ヘッダ・ファイルが取り込まれたときに有効な実際のポインタ・サイズ・コンテキストに関して，何も仮定することができません。さらに，HP Cコンパイラでは，__INITIAL_POINTER_SIZE マクロとポインタ・サイズを変更するための機能だけが提供され，現在のポインタ・サイズを判断するための方法は提供されません。

ポインタ・サイズに依存するすべてのヘッダ・ファイルは，ポインタ・サイズ・コンテキストの保存**3** ， 初期化**4** ， 変更**5** ， 復元**6** を行う責任があります。

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
:
```

一部の関数プロトタイプでは，64 ビット・ポインタ・サイズ・コンテキストで 32 ビット・ポインタを参照しなければなりません。また，32 ビット・ポインタ・サイズ・コンテキストで64 ビット・ポインタを参照しなければならない関数プロトタイプもあります。

HP Cは，typedefが作成されるときに，typedefで使用されるポインタ・サイズをバインドします。このヘッダ・ファイルを/POINTER_SIZE 修飾子なしで，または/POINTER_SIZE=SHORT を指定してコンパイルしたと仮定すると，__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を宣言する前に，ポインタ・コンテキストは 32 ビット・ポインタから 64 ビット・ポインタに変更されます**2** 。

32 ビット・ポインタに制限される関数は，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** に配置されます。他のすべての関数は，ヘッダ・ファイルの 64 ビット・コンテキスト・セクション**3** に配置されます。

ポインタ・サイズの影響を受けない関数 **(4** と**5)** は 64 ビット・セクションに配置されます。 32 ビットのアドレス戻り値以外はポインタ・サイズの影響を受けない関数**5** も 64 ビット・セクションに配置され，前に説明したように， 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
:
```

この例では， 32 ビット版と 64 ビット版の両方が用意されている関数の宣言を示しています。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ます。

関数に対する通常のインタフェース**2** は，/POINTER_SIZE 修飾子に指定したポインタ・サイズを使用して宣言されます。ヘッダ・ファイルは 64 ビット・ポインタ・コンテキストに配置され，**1** に示した文が指定されているため，**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** は 32 ビット・ポインタ・サイズ・コンテキストで宣言され， 64 ビット固有のインタフェース**4** は 64 ビット・ポインタ・サイズ・コンテキストで宣言されます。

2
入出力について

HP C Run-Time Library (RTL) には， 3 種類の入出力 (I/O) があります。それは UNIX I/O，標準 I/O，端末 I/O です。表 2–1 は，HP C RTL のすべての I/O 関数とマクロを示しています。各関数とマクロの詳細については，『HP C ランタイム・ライブラリ・リファレンス・マニュアル (下巻)』「リファレンス・セクション」を参照してください。

表 2–1 I/O 関数とマクロ

関数またはマクロ	説明
UNIX I/O— **ファイルのオープンとクローズ**	
close	ファイル記述子に関連付けられているファイルをクローズする。
creat	ファイルを新規作成する。
dup, dup2	open，creat，pipeから返されたファイル記述子によって指定されるファイルに新しい記述子を割り当てる。
open	ファイルをオープンし，ファイルの位置をファイルの先頭に設定する。
UNIX I/O— **ファイルからの読み込み**	
read	ファイルからバイトを読み込み，バッファに格納する。
UNIX I/O— **ファイルへの書き込み**	
write	指定されたバイト数をバッファからファイルに書き込む。
UNIX I/O— **ファイル内の位置設定**	
lseek	ストリーム・ファイルを任意のバイト位置に設定し，新しい位置をintとして返す。

(次ページに続く)

表 2–1 (続き) I/O 関数とマクロ

関数またはマクロ	説明
UNIX I/O— **追加の** X/Open I/O **関数とマクロ**	
fstat, stat	ファイル記述子またはファイル指定に関する情報にアクセスする。
flockfile, ftrylockfile, funlockfile	ファイル・ポインタ・ロック関数。
fsync	指定されたファイルに保存するためにバッファに格納されている情報をディスクに書き込む。
getname	ファイル記述子に関連付けられているファイル指定を返す。
isapipe	ファイル記述子がパイプに関連付けられている場合は 1，関連付けられていない場合は 0 を返す。
isatty	指定されたファイル記述子が端末に関連付けられている場合は 1，関連付けられていない場合は 0 を返す。
lwait	保留中の非同期 I/O の終了を待つ。
ttyname	ファイル記述子 0 (デフォルトの入力デバイス) に関連付けられている端末デバイスの名前 (ヌル区切り) を指すポインタを返す。
標準 I/O— **ファイルのオープンとクローズ**	
fclose	ファイル制御ブロックに関連付けられているバッファの内容を書き込み，ファイル・ポインタに関連付けられているファイル制御ブロックおよびバッファを解放することにより，ファイルをクローズする。
fdopen	open，creat，dup，dup2，pipe関数から返されたファイル記述子にファイル・ポインタを関連付ける。
fopen	FILE 構造体のアドレスを返すことにより，ファイルをオープンする。 (ファイルをオープンしてファイル・ポインタを返す。)
freopen	ファイル指定によって名前が指定されるファイルを，ファイル・ポインタによってアドレスが指定されるオープンされているファイルに置き換える。 (ファイルをオープンして指定されたファイル・ポインタに結びつける。)
標準 I/O— **ファイルからの読み込み**	
fgetc, getc, fgetwc, getw, getwc	指定されたファイルから文字を 1 文字読み込む。
fgets, fgetws	指定されたファイルから 1 行を読み込み，文字列を引数に格納する。
fread	指定された数の項目をファイルから読み込む。
fscanf, fwscanf, vfscanf, vfwscanf	指定されたファイルから書式に従って読み込む。
sscanf, swscanf, vsscanf, vswscanf	メモリ内の文字列から書式に従って読み込む。
ungetc, ungetwc	文字を入力ストリームに戻し，挿入した文字の前にあるストリームはそのまま残す。

(次ページに続く)

表 2–1 (続き) I/O 関数とマクロ

関数またはマクロ	説明
標準 I/O— ファイルへの書き込み	
fprintf, fwprintf, vfprintf, vfwprintf	指定されたファイルに書式に従って出力する。
fputc, putc, putw, putwc, fputwc	指定されたファイルに文字を書き込む。
fputs, fputws	文字列をファイルに書き込む (ヌル文字は含まない)。
fwrite	指定された数の項目をファイルに書き込む。
sprintf, swprintf, vsprintf, vswprintf	メモリ内の文字列に書式設定された出力を実行する。
標準 I/O— ファイル内の位置設定	
fflush	指定されたファイルに保存するためにバッファに格納されている情報を RMS に送信する。
fgetpos	ストリームのファイル位置指示子の現在の値を格納する。
fsetpos	ポインタが指しているオブジェクトの値に従って，ストリームのファイル位置指示子を設定する。
fseek, fseeko	ファイルの位置をファイル内の指定されたバイト・オフセットに設定する。
ftell, ftello	指定されたストリーム・ファイルに対する現在のバイト・オフセットを返す。
rewind	ファイルの位置を先頭に設定する。
標準 I/O— 追加の標準 I/O 関数とマクロ	
access	ファイルを調べて，指定されたアクセス・モードが許可されているかどうかを確認する。
clearerr	ファイルに対するエラー指示子およびファイルの終端 (EOF) 指示子をリセットする。
feof	ファイルを調べて，ファイルの終端 (EOF) に到達したかどうかを確認する。
ferror	ファイルへの書き込みまたは読み込みでエラーが発生した場合， 0 以外の整数を返す。
fgetname	ファイル・ポインタに関連付けられているファイル指定を返す。
fileno	指定されたファイルを識別する整数のファイル記述子を返す。
ftruncate	ファイルを指定された位置で切り捨てる。
fwait	保留中の非同期 I/O の終了を待つ。
fwide	ストリームの単位を設定する。
mktemp	テンプレートから固有の名前を作成する。
remove, delete	ファイルを削除する。
rename	既存のファイルに新しい名前を付ける。
setbuf, setvbuf	バッファを入力ファイルまたは出力ファイルに関連付ける。

(次ページに続く)

表 2–1 (続き) I/O 関数とマクロ

関数またはマクロ	説明
標準 I/O— 追加の標準 I/O 関数とマクロ	
tmpfile	一時ファイルを作成して更新のためにオープンする。
tmpnam	他の関数呼び出しでファイル名引数の代わりに使用できる文字列を作成する。
端末 I/O— ファイルからの読み込み	
getchar, getwchar	標準入力 (stdin) から 1 文字を読み込む。
gets	標準入力 (stdin) から 1 行を読み込む。
scanf, wscanf, vscanf, vwscanf	標準入力から書式設定された入力を実行する。
端末 I/O— ファイルへの書き込み	
printf, wprintf, vprintf, vwprintf	標準出力 (stdout) に書式設定された出力を実行する。
putchar, putwchar	標準出力に 1 文字を書き込み，その文字を返す。
puts	標準出力に文字列と改行文字を書き込む。

2.1 RTL ルーチンからの RMS の使用

HP C RTL の I/O 関数とマクロを使用してファイルを作成する場合，次の属性も含めて，多くの RMS ファイル属性の値を指定できます。

- 割り当てサイズ
- ブロック・サイズ
- デフォルトのファイル拡張
- デフォルトのファイル名
- ファイル・アクセス・コンテキスト・オプション
- ファイル処理オプション
- ファイル共用オプション
- マルチブロック・カウント
- マルチバッファ・カウント
- 最大レコード・サイズ
- レコード属性
- レコード・フォーマット
- レコード処理オプション

これらの値の詳細については，「リファレンス・セクション」のcreat関数の説明を参照してください。

これらの値は，open，fopen，freopenなどの関数でも設定できます。

RMS の詳細については，『HP C User's Guide for OpenVMS Systems』を参照してください。

2.2 UNIX I/O と標準 I/O

UNIX I/O 関数は UNIX システム・サービスであり，現在はISO POSIX-1 (ISO Portable Operating System Interface) によって標準化されています。

UNIX I/O 関数では，ファイル記述子を使用してファイルにアクセスします。**ファイル記述子**とは，ファイルを識別する整数です。ファイル記述子は次のように宣言されます。ここで，file_descはファイル記述子の名前です。

```
int  file_desc;
```

creatなどの UNIX I/O 関数は，ファイル記述子をファイルに関連付けます。次の例について考えてみましょう。

```
file_desc1 = creat("INFILE.DAT", 0, "rat=cr", "rfm=var");
```

この文は，ファイル・アクセス・モード 0，キャリッジ・リターン制御，可変長レコードで INFILE.DAT というファイルを作成し，変数file_desc1をファイルに関連付けます。読み込みや書き込みなどの操作のためにファイルにアクセスするときは，ファイル記述子を使用してファイルを参照します。次の例を参照してください。

```
write(file_desc1, buffer, sizeof(buffer));
```

この文はバッファの内容を INFILE.DAT に書き込みます。

標準 I/O 関数やマクロの代わりに， UNIX I/O 関数やマクロを使用しなければならない状況があります。この 2 種類の I/O の詳細についてと，各 I/O で RMS ファイル・フォーマットがどのように操作されるかについては，第 1 章を参照してください。

標準 I/O 関数は ANSI C 標準で指定されています。

標準 I/O 関数では， UNIX I/O の機能にバッファリング機能が追加され，ファイルにアクセスするためにファイル・ポインタが使用されます。**ファイル・ポインタ**とは，FILE *型のオブジェクトです。これは，次に示すように，<stdio.h>ヘッダ・ファイルに定義されているtypedefです。

```
typedef  struct  _iobuf  *FILE;
```

_iobuf識別子も<stdio.h>に定義されています。

ファイル・ポインタを宣言するには，次のように指定します。

```
FILE  *file_ptr;
```

既存のファイルの作成やオープンには，標準 I/O のfopen関数を使用します。次の例を参照してください。

```
#include <stdio.h>
main()
{
   FILE  *outfile;
   outfile  =  fopen("DISKFILE.DAT", "w+");
      .
      .
      .
}
```

この例では，ファイル DISKFILE.DAT が書き込み更新アクセスのためにオープンされます。

HP C RTL では，ファイル記述子とファイル・ポインタの間の変換のために，次の関数が用意されています。

- fileno— 指定されたファイル・ポインタに関連付けられているファイル記述子を返します。
- fdopen—open，creat，dup，dup2，pipe関数から返されたファイル記述子にファイル・ポインタを関連付けます。

2.3 ワイド文字 I/O 関数とバイト I/O 関数

ワイド文字 I/O 関数は，大部分のバイト I/O 関数と同様の動作を実行します。ただし，基本単位としてワイド文字を取り扱う点がバイト I/O 関数と異なります。

しかし，外部表現 (ファイル内の表現) はワイド文字ではなく，マルチバイト文字シーケンスです。ワイド文字で書式設定された入出力関数の場合，次のことに注意してください。

- ワイド文字で書式設定された**入力**関数 (たとえばfwscanfなど) は，指定されたディレクティブとは無関係に，常にファイルからマルチバイト文字シーケンスを読み込み，このシーケンスをワイド文字シーケンスに変換してから，その後の処理を実行します。
- ワイド文字で書式設定された**出力**関数 (たとえばfwprintfなど) は，最初にワイド文字型の引数をマルチバイト文字シーケンスに変換した後，オペレーティング・システムの出力プリミティブを呼び出すことにより，ワイド文字を出力ファイルに書き込みます。

バイト I/O 関数は状態依存エンコーディングを取り扱うことができません。ワイド文字 I/O 関数はこのような操作が可能です。ワイド文字 I/O 関数は，各ワイド文字ストリームをmbstate_t型の変換状態オブジェクトに関連付けることにより，この操作を実行します。

ワイド文字 I/O 関数は次のとおりです。

```
fgetwc      fputwc      fwscanf     fwprintf     ungetwc
fgetws      fputws      wscanf      wprintf
getwc       putwc                   vfwprintf
getwchar    putwchar                vwprintf
```

バイト I/O 関数は次のとおりです。

```
fgetc       fputc       fscanf      fprintf       ungetc
fgets       fputs       scanf       printf        fread
getc        putc                    vfprinf       fwrite
gets        puts                    vprintf
getchar     putchar
```

ワイド文字入力関数は，fgetwc関数を連続して呼び出すことによって書き込まれる場合，ストリームからマルチバイト文字を読み込み，これらの文字をワイド文字に変換します。各変換は，ストリーム独自のmbstate_tオブジェクトによって記述される変換状態を指定してmbrtowc関数を呼び出したかのように実行されます。

ワイド文字出力関数は，fputwc関数を連続して呼び出すことにより書き込んだかのように，ワイド文字をマルチバイト文字に変換し，これらの文字をストリームに書き込みます。各変換は， I/O ストリームの独自のmbstate_tオブジェクトによって記述される変換状態を指定してwcrtomb関数を呼び出したかのように実行されます。

ワイド文字 I/O 関数で不正なマルチバイト文字が検出されると，errnoが EILSEQ に設定されます。

2.4 変換指定

複数の標準 I/O 関数 (端末 I/O 関数を含む) では， I/O のデータの書式を指定するために変換指定が使用されます。これらの関数は，書式設定された入力関数と書式設定された出力関数です。次の例について考えてみましょう。

```
int     x = 5.0;
FILE    *outfile;
   .
   .
   .
fprintf(outfile, "The answer is %d.\n", x);
```

識別子 outfile に関連付けられたファイルに書き込まれる文字列内の変換指定%dは，変数xの 10 進数に置き換えられます。

各変換指定はパーセント記号(%)から始まり，**変換指定子**で終了します。これは，実行する変換の種類を指定する文字です。パーセント記号と変換指定子の間に，必要に応じて任意の文字を指定することができます。

ワイド文字で書式設定された I/O 関数の場合，変換指定はワイド文字の文字列です。対応するバイト I/O 関数では，変換指定はバイト列です。

これらの任意に指定できる文字と変換指定子については， 2.4.1 項と 2.4.2 項を参照してください。

2.4.1 入力情報の変換

情報を入力する場合の書式指定文字列には，次の 3 種類の項目を含むことができます。

- 空白文字 (スペース，タブ，改行文字)。入力フィールドの省略可能な空白文字に対応します。
- 一般の文字 (%以外)。入力の中で次の空白以外の文字に対応しなければなりません。
- 変換指定。入力フィールドの文字の変換を管理し，対応する入力ポインタによって示されるオブジェクトへの代入も管理します。

各入力ポインタは，対応する変換指定と同じ型のオブジェクトを示すアドレス式です。変換指定は書式指定文字列の一部になります。示されるオブジェクトは，入力値を受け取るターゲットです。変換指定と同じ数だけ入力ポインタが必要であり，アドレスが指定されるオブジェクトは変換指定と同じ型でなければなりません。

変換指定は次の文字で構成され，順序はここに示したとおりになります。

- パーセント記号(%)または%n$というシーケンス (nは整数)。

  %n$というシーケンスは，変換がn番目の入力ポインタに適用されることを示します。ただし，nは[1, NL_ARGMAX]の範囲の 10 進整数です (<limits.h>ヘッダ・ファイルを参照)。たとえば，%5$から始まる変換指定は，変換が書式指定の後の 5 番目の入力ポインタに適用されることを示します。%$というシーケンスは無効です。

  変換指定の先頭に%n$というシーケンスが指定されていない場合は，変換指定は左から右に順に入力ポインタに対応付けられます。書式指定では，変換指定を 1 種類だけ (%または%n$) 使用するようにしなければなりません。
- 1 文字以上の省略可能な文字 (表 2–2 を参照)。
- 変換指定子 (表 2–3 を参照)。

表 2–2 は，パーセント記号(%) (または%n$というシーケンス) と変換指定子の間に指定できる文字を示しています。これらの文字は省略可能ですが，指定する場合は，表 2–2 に示した順に指定する必要があります。

表 2–2 % (または%n$) と入力変換指定子の間に指定できる省略可能な文字

文字	意味
*	代入禁止文字
フィールド幅	0 以外の 10 進整数であり，最大フィールド幅を指定する。 ワイド文字入力関数の場合は，フィールド幅はワイド文字の文字数で表現される。 バイト入力関数の場合は，ディレクティブが次のいずれかである場合を除き，フィールド幅はバイト数で表現される。 `%lc, %ls, %C, %S, %[` これらの場合，フィールド幅はマルチバイト文字単位で表現される。 /L_DOUBLE=64 でコンパイルしたプログラムの場合 (デフォルトの/L_DOUBLE=128 ではなくて)，最大フィールド幅は 1024。
h，l， L (または ll)	変換指定子 d， i， n のいずれかの前に指定する。対応する引数が`int`を指すポインタではなく，`short int`を指すポインタの場合は， h を指定する。`long int`を指すポインタの場合は， l (小文字のエル) を指定する。 OpenVMS Alpha システムの場合のみ，`__int64`を指すポインタの場合は， L または ll (2 つの小文字のエル) を指定する。 変換指定子 o， u，または x の前に指定する。対応する引数が`unsigned int`を指すポインタではなく，`unsigned short int`を指すポインタの場合は， h を指定する。`unsigned long int`を指すポインタの場合は， l を指定する。 OpenVMS Alpha システムの場合のみ，`unsigned __int64`を指すポインタの場合は， L または ll を指定する。 変換指定子 c， s，[の前に指定する。対応する引数が`wchar_t`を指すポインタの場合は， l (小文字のエル) を指定する。 最後に，変換指定子 e， f， g の前に指定する。対応する引数が`float`を指すポインタではなく，`double`を指すポインタの場合は， l (小文字のエル) を指定する。`long double`を指すポインタの場合は L を指定する。 h， l， L， ll のいずれかが他の変換指定子とともに指定されている場合は，その動作は未定義になる。

表 2–3 は，書式設定された入力の変換指定子を示しています。

表 2–3 書式設定された入力の変換指定子

指定子	入力の種類[1]	説明
d		base引数の値を 10 に設定したstrtol関数のサブジェクト・シーケンスに対して期待される書式と同じ書式の 10 進整数が入力に期待される。対応する引数はintを指すポインタでなければならない。
i		入力文字の先頭文字によって型が決定される整数が期待される。先頭が 0 の場合は 8 進数，先頭が 0X または 0x の場合は 16 進数，他のすべての形式の場合は 10 進数であると解釈される。対応する引数はintを指すポインタでなければならない。
o		入力に 8 進整数 (先頭の 0 はあってもなくてもかまわない) が期待される。対応する引数はintを指すポインタでなければならない。
u		base引数の値が 10 のstrtoul関数のサブジェクト・シーケンスに対して期待される書式と同じ書式の 10 進整数が入力に期待される。
x		入力に 16 進整数 (先頭の 0x はあってもなくてもかまわない) が期待される。対応する引数はunsigned intを指すポインタでなければならない。
c	バイト	入力に 1 バイトが期待される。対応する引数はcharを指すポインタでなければならない。 c 変換指定子の前にフィールド幅が指定されている場合は，フィールド幅によって指定される文字数が読み込まれる。この場合，対応する引数はchar型の配列を指すポインタでなければならない。 この変換指定子の前に省略可能な文字 l (小文字のエル) が指定されている場合は，ワイド文字コードに変換されるマルチバイト文字が入力に期待される。 対応する引数はwchar_t型を指すポインタでなければならない。 c 変換指定子の前にフィールド幅も指定されている場合は，フィールド幅によって指定された文字数が読み込まれる。この場合，対応する引数はwchar_tの配列を指すポインタでなければならない。
	ワイド文字	省略可能なフィールド幅に指定された文字数が期待される。指定されていない場合は 1 になる。 c 指定子の前に l (小文字のエル) が指定されていない場合は，対応する引数はcharの配列を指すポインタでなければならない。 c 指定子の前に l (小文字のエル) が指定されている場合は，対応する引数はwchar_tの配列を指すポインタでなければならない。
C	バイト	ワイド文字コードに変換されるマルチバイト文字が入力に期待される。対応する引数はwchar_t型を指すポインタでなければならない。 C 変換指定子の前にフィールド幅も指定されている場合は，フィールド幅によって指定される文字数が読み込まれる。その場合，対応する引数はwchar_tの配列を指すポインタでなければならない。
	ワイド文字	省略可能なフィールド幅に指定された文字数のシーケンスが期待される。指定されていない場合は 1 になる。対応する引数はwchar_tの配列を指すポインタでなければならない。

[1] **バイト**または**ワイド文字**。ただし，どちらも示されていない場合は，指定子の説明は両方に適用される。

(次ページに続く)

表 2–3 (続き) 書式設定された入力の変換指定子

指定子	入力の種類[1]	説明
s	バイト	入力にバイト・シーケンスが期待される。対応する引数は，バイト・シーケンスと，自動的に付加されるヌル区切り文字 (\ 0) を格納できるだけの十分な大きさの文字配列を指すポインタでなければならない。入力フィールドはスペース，タブ，改行文字で区切られる。 この変換指定子の前に省略可能な文字 l (小文字のエル) が指定されている場合は，ワイド文字コードに変換されるマルチバイト文字シーケンスが入力に期待される。対応する引数は，文字シーケンスの他に，自動的に付加される区切り文字のヌル・ワイド文字コードを格納できるだけの十分な大きさのワイド文字 (wchar_t型) の配列を指すポインタでなければならない。入力フィールドはスペース，タブ，改行文字で区切られる。
	ワイド文字	入力に空白文字以外の文字シーケンスが期待される (概念上)。 s 指定子の前に l (小文字のエル) が指定されていない場合は，対応する引数は，シーケンスの他に，自動的に末尾に付加されるヌル・バイトを格納できるだけの十分な大きさのcharの配列を指すポインタでなければならない。 s 指定子の前に l (小文字のエル) が指定されている場合は，対応する引数はシーケンスの他に，自動的に末尾に付加されるヌル・ワイド文字を格納できるだけの十分な大きさのwchar_tの配列を指すポインタでなければならない。
S	バイト	ワイド文字コードに変換されるマルチバイト文字シーケンスが入力に期待される。対応する引数は，シーケンスの他に，自動的に末尾に付加されるヌル・ワイド文字コードを格納できるだけの十分な大きさのワイド文字 (wchar_t型) の配列を指すポインタでなければならない。
	ワイド文字	空白文字以外の文字シーケンスが入力に期待される。対応する引数は，シーケンスの他に，自動的に末尾に付加されるヌル・ワイド文字を格納できるだけの十分な大きさのwchar_tの配列を指すポインタでなければらない。
e, f, g		入力に浮動小数点数値が期待される。対応する引数は，floatを指すポインタでなければならない。浮動小数点数値の入力形式は次のとおりである。[±]nnn[radix][ddd][{E | e}[±]nn]. n と d は 10 進数である (フィールド幅によって示される桁数から符号と英字の E を差し引いた桁数)。基数を示す文字は現在のロケールで定義される。

[1] **バイト**または**ワイド文字**。ただし，どちらも示されていない場合は，指定子の説明は両方に適用される。

(**次ページに続く**)

表 2–3 (続き) 書式設定された入力の変換指定子

指定子	入力の種類[1]	説明
[. . .]		空白文字によって区切られていない空でない文字シーケンスが期待される。角括弧は，入力シーケンスに期待される文字セット (scanset) を囲む。入力シーケンス内の文字のうち， scanset に含まれる文字に対応しない文字が検出されると，文字シーケンスは終了する。 角括弧で囲まれたすべての文字で scanset が構成される。ただし，左角括弧の直後の文字がサーカンフレックス (^) の場合は例外である。この場合は， scanset に含まれる文字は，サーカンフレックスと右角括弧で囲まれる文字以外のすべての文字になる。サーカンフレックスと右角括弧で**囲まれた**文字が検出されると，入力文字シーケンスは終了する。 変換指定子が[ ]または[^]で始まる場合は，右角括弧文字は scanset に含まれ，次の右角括弧文字が指定を終了する右角括弧文字になる。それ以外の場合は，最初の右角括弧文字が検出されたときに，指定は終了する。
	バイト	[指定子の前に l (小文字のエル) が指定されていない場合は，scanset に含まれる文字はシングル・バイト文字だけでなければならない。この場合，対応する引数は，シーケンスの他に，自動的に末尾に付加されるヌル・バイトを格納できるだけの十分な大きさの`char`の配列を指すポインタでなければならない。 [指定子の前に l (小文字のエル) が指定されている場合は，入力シーケンス内の文字はマルチバイト文字であると解釈され，その後の処理のためにワイド文字シーケンスに変換される。文字の範囲が scanset に指定されている場合は，処理は現在のプログラムのロケールの LC_COLLATE カテゴリに従って実行される。この場合，対応する引数は，シーケンスの他に，自動的に末尾に付加されるヌル・ワイド文字を格納できるだけの十分な大きさの`wchar_t`の配列を指すポインタでなければならない。
	ワイド文字	[変換指定子の前に l (小文字のエル) が指定されていない場合は，処理は%l[指定子のバイト入力の場合と同じである。ただし，対応する引数は，マルチバイト・シーケンスの他に，自動的に末尾に付加されるヌル・バイトを格納できるだけの十分な大きさの`char`の配列を指すポインタでなければならない。 [変換指定子の前に l (小文字のエル) が指定されている場合は，処理は前に説明した場合と同じであるが，対応する引数は，ワイド文字シーケンスの他に，自動的に末尾に付加されるヌル・ワイド文字を格納できるだけの十分な大きさの`wchar_t`の配列を指すポインタでなければならない。
p		`void`を指すポインタが引数として期待される。入力値は 16 進数として解釈される。
n		入力は使用されない。対応する引数は整数を指すポインタである。整数は，書式設定された入力関数に対するこの呼び出しで，これまで入力ストリームから読み込まれた文字数として代入される。%n ディレクティブの実行は，書式設定された入力関数の実行が終了したときに返される代入カウントを増分しない。
%		1 つのパーセント記号に対応する。変換や代入は実行されない。完全な変換指定は%%である。

[1] **バイト**または**ワイド文字**。ただし，どちらも示されていない場合は，指定子の説明は両方に適用される。

注意事項

- 角括弧([ ])変換指定子を使用して，入力フィールドの区切り文字を変更することができます。変更しなかった場合は，入力フィールドは空白文字以外の文字列として定義されます。入力フィールドは次の空白文字が検出されるまで，またはフィールド幅が指定されているときは，そのフィールド幅に到達するまでであると解釈されます。改行文字は空白文字であるため，関数は行やレコード境界を超えて読み込みます。
- 入力変換関数のいずれかを呼び出すと，前の呼び出しで最後に処理された文字の直後から検索が再開されます。
- 代入禁止文字(*)が書式指定に指定されている場合は，代入は実行されません。対応する入力フィールドが解釈された後，スキップされます。
- HP Cでは値による呼び出しだけが認められるため，引数はポインタまたは他のアドレス値に評価される式でなければなりません。数値を 10 進数として読み込み，その値を n に代入するには，次の形式を使用する必要があります。

  ```
  scanf("%d", &n)
  ```

  次の形式は使用できません。

  ```
  scanf("%d", n)
  ```

- 書式指定の空白は，入力フィールドの省略可能な空白に対応します。次の書式指定について考えてみましょう。

  ```
  field = %x
  ```

  この書式指定は次の形式に対応します。

  ```
  field = 5218
  field=5218
  field= 5218
  field =5218
  ```

  次の例には対応しません。

  ```
  fiel d=5218
  ```

2.4.2 出力情報の変換

情報を出力する場合の書式指定文字列には，次の文字を指定できます。

- 出力にコピーされる一般文字。
- 変換指定。これらの変換指定を指定することにより，対応する出力ソースは特定の書式で文字列に変換されます。変換指定は左から右への順に出力ソースに対応付けられます。

変換指定は次の文字で構成され，順序はここに示したとおりになります。

- パーセント記号(%)または%n$というシーケンス。

  %n$というシーケンスは，変換がn番目の出力ソースに適用されることを示します。ただし，nは[1, NL_ARGMAX]の範囲の 10 進整数です (<limits.h>ヘッダ・ファイルを参照)。たとえば，%5$から始まる変換指定は，変換が書式指定の後の 5 番目の出力ソースに適用されることを示します。

  変換指定の先頭に%n$というシーケンスが指定されていない場合は，変換指定は左から右への順に出力ソースに対応します。書式指定では，変換指定を 1 種類だけ (%または%n$) 使用するようにしなければなりません。

- 1 文字以上の省略可能な文字 (表 2–4 を参照)。
- 変換指定子 (表 2–5 を参照) で変換指定は終了します。

変換指定の例については，第 2.6 節のサンプル・プログラムを参照してください。

表 2–4 は，パーセント記号(%) (または%n$) というシーケンスと変換指定子の間に指定できる文字を示しています。これらの文字は省略可能ですが，指定する場合は，表 2–4 に示した順に指定する必要があります。

表 2–4 % (または%n$) と出力変換指定子の間に指定できる省略可能な文字

文字	意味	
フラグ	次のフラグ文字を単独または任意の順序で組み合わせて，変換指定を変更することができる。	
	' (一重引用符)	数値変換で 3 桁ごとの区切り文字を付けて書式設定することを要求する。基数を表す文字の左側の数字にだけ桁区切り文字が付けられる。桁区切り文字として使用する文字と区切り文字の位置は，プログラムの現在のロケールで定義される。
	– (ハイフン)	変換した出力ソースをそのフィールド内で左桁揃えする。
	+	符号付き変換で明示的な符号を付けることを要求する。このフラグを指定しなかった場合，符号付き変換で負の値が変換された場合にだけ，結果の先頭に符号が付加される。
	space	符号付き変換の最初の文字が符号でない場合や，変換した結果，文字が生成されない場合は，符号付き変換の結果の先頭にスペースを付加する。spaceと + フラグをどちらも指定した場合は，spaceフラグは無視される。

(次ページに続く)

表 2–4 (続き) % (または%n$) と出力変換指定子の間に指定できる省略可能な文字

文字	意味	
	#	代替変換書式を要求する。指定した変換に応じて，異なる動作が実行される。 o (8 進数) 変換の場合は， 1 桁目が 0 になるように精度が上げられる。 x (または X) 変換の場合は， 0 以外の結果の先頭に 0x (または 0X) が付加される。 e， E， f， g， G 変換の場合は，結果に小数点が含まれる。整数値の場合も末尾に小数点が付加される。 g および G 変換の場合は，後続の 0 は切り捨てられない。 他の変換の場合は，#の効果は未定義である。
	0	d， i， o， u， x， X， e， E， f， g， G 変換の場合，フィールド幅になるように埋め込むパッド文字として，スペースではなく 0 を使用する。 0 と–フラグをどちらも指定した場合は，0 フラグは無視される。 d， i， o， u， x， X 変換で，精度が指定されている場合は， 0 フラグは無視される。他の変換の場合は， 0 フラグの動作は未定義である。
フィールド幅	最小フィールド幅は 10 進整数定数または出力ソースによって指定できる。出力ソースを指定するには，アスタリスク(*)または*n$というシーケンスを使用する。ただし，nは，書式指定の後のn番目の出力ソースを参照する。 最小フィールド幅は，書式ディレクティブの他のすべてのコンポーネントに従って変換を実行した後で考慮される。このコンポーネントは，次に示すように変換結果のパッド文字埋め込み処理に影響する。 変換の結果が最小フィールド幅より長い場合は，結果がそのまま書き込まれる。 変換の結果が最小幅より短い場合は，フィールド幅になるようにパッド文字が埋め込まれる。デフォルトではパッド文字はスペースである。 0 フラグが指定された場合は，パッド文字は 0 である。この場合，幅が 8 進数であることを意味するわけではない。デフォルト設定では，パッド文字は左側に埋め込まれ，マイナス記号が指定された場合は右側に埋め込まれる。 ワイド文字出力関数の場合は，フィールド幅はワイド文字の文字数で指定される。バイト出力関数の場合は，バイト数で指定される。 /L_DOUBLE=64 でコンパイルしたプログラムの場合 (デフォルトの/L_DOUBLE=128 ではなくて)，最大フィールド幅は 1024。	
. (ピリオド)	フィールド幅と精度を区切る。	

(次ページに続く)

表 2–4 (続き) % (または%n$) と出力変換指定子の間に指定できる省略可能な文字

文字	意味
精度	精度は次のいずれかを定義する。 • d, i, o, u, x, X 変換の場合は表示される最小桁数 • e, E, f 変換の場合は小数点以下の桁数 • g および G 変換の場合は最大有効桁数 • s または S 変換の場合は文字列から書き込まれる最大文字数 精度が他の変換指定子と組み合わせて指定された場合は，動作は未定義である。 精度は 10 進整数定数または出力ソースによって指定できる。出力ソースを指定するには，アスタリスク(*)または*n$というシーケンスを使用する。ただし，nは，書式指定の後のn番目の出力ソースを参照する。 ピリオドだけが指定されている場合は，精度は 0 であると解釈される。
h, l, L (または ll)	h は，後続の d, i, o, u, x, X 変換指定子がshort intまたはunsigned short int引数に適用されることを指定する。また，h は後続の n 変換指定子がshort int引数を指すポインタに適用されることも指定する。 l (小文字のエル) は，後続の d, i, o, u, x, X 変換指定子がlong intまたはunsigned long int引数に適用されることを指定する。l は，後続の n 変換指定子がlong int引数を指すポインタに適用されることも指定する。 OpenVMS Alpha システムおよび OpenVMS Integrity システムでは，L または ll (2 つの小文字のエル) は，後続の d, i, o, u, x, X 変換指定子が__int64またはunsigned __int64引数に適用されることを指定する。*(Integrity，Alpha)* L は，後続の e, E, f, g, G 変換指定子がlong double引数に適用されることを指定する。 l は，後続の c または s 変換指定子がwchar_t引数に適用されることを指定する。 h, l, L のいずれかが他の変換指定子と組み合わせて指定されている場合は，動作は未定義である。 OpenVMS Alpha システムでは，HP C intの値はlongの値と等価である。

表 2–5 は，書式設定された出力の変換指定子を示しています。

表 2–5 書式設定された出力の変換指定子

指定子	出力の種類[1]	説明
d, i		int引数を符号付き 10 進形式に変換する。
o		unsigned int引数を符号なし 8 進形式に変換する。
u		unsigned int引数を符号なし 10 進形式に変換する (数値が 0 ～ 4,294,967,295 の範囲の場合)。

[1] **バイト**または**ワイド文字**。ただし，どちらも示されていない場合は，指定子の説明は両方に適用される。

(**次ページに続く**)

表 2–5 (続き) 書式設定された出力の変換指定子

指定子	出力の種類[1]	説明
x, X		unsigned int引数を符号なし 16 進形式 (先頭の 0x は付いている場合も付いていない場合もある) に変換する。X変換の場合は英字abcdefが使用され，X変換の場合は英字ABCDEFが使用される。
f		floatまたはdouble引数を[–]mmm.nnnnnn. という形式に変換する。次のように， n の数は精度指定の値に等しい。 • 精度が指定されていない場合，デフォルトは 6 である。 • 精度が 0 で，#フラグが指定されている場合は，小数点は出力されるが，小数点以下の n は出力されない。 • 精度が 0 で，#フラグが指定されていない場合は，小数点も出力されない。 • 小数点が出力される場合は，その前に少なくとも 1 桁が出力される。 値は適切な桁数に四捨五入される。
e, E		floatまたはdouble引数を[–]m.nnnnnnE±xx という形式に変換する。 n の数は精度によって指定される。精度が指定されていない場合は，デフォルトは 6 である。精度が 0 として指定され，#フラグが指定されている場合は，小数点は出力されるが，小数点以下の n は出力されない。精度が 0 で，#フラグが指定されていない場合は，小数点も出力されない。 e 変換の場合は 'e' が出力される。 E 変換の場合は 'E' が出力される。指数には常に少なくとも 2 桁が含まれる。値が 0 の場合は，指数は 0 である。
g, G		floatまたはdouble引数を f または e という形式に変換する (または G 変換指定子が使用されている場合は E)。有効桁数は精度によって指定される。精度が 0 の場合は，有効桁数は 1 であると解釈される。使用される書式は引数の値に応じて異なる。変換の結果，指数が–4より小さい場合や，精度以上になる場合にだけ，書式 e (または E) が使用される。それ以外の場合は，書式 f が使用される。結果の小数点以下の部分で後続の 0 は切り捨てられる。小数点以下に有効桁がある場合にだけ，小数点が出力される。
c	バイト	int引数をunsigned charに変換し，結果のバイトを書き込む。 この変換指定子の前に省略可能な文字 l (小文字のエル) が指定されている場合は，この指定子はwchar_t引数を，文字を表すバイト配列に変換し，結果の文字を出力する。フィールド幅が指定されていて，結果の文字がフィールド幅より少ないバイト数の場合は，パッド文字としてスペースが埋め込まれる。精度が指定されている場合は，動作は未定義である。

[1] **バイト**または**ワイド文字**。ただし，どちらも示されていない場合は，指定子の説明は両方に適用される。

(**次ページに続く**)

表 2–5 (続き) 書式設定された出力の変換指定子

指定子	出力の種類[1]	説明
	ワイド文字	c 指定子の前に l (小文字のエル) が指定されていない場合は，int引数は，btowcを呼び出したかのようにワイド文字に変換され，結果の文字が出力される。 c 指定子の前に l (小文字のエル) が指定されている場合は，この指定子はwchar_t引数を，文字を表すバイト配列に変換し，結果の文字を出力する。フィールド幅が指定されていて，結果の文字がフィールド幅より少ない文字数の場合は，指定された幅になるようにスペース文字が埋め込まれる。精度が指定されている場合は，動作は未定義である。
C	バイト	wchar_t引数を，文字を表すバイト配列に変換し，結果の文字を出力する。フィールド幅が指定されていて，結果の文字がフィールド幅より少ないバイト数の場合は，指定された幅になるようにスペース文字が埋め込まれる。精度が指定されている場合は，動作は未定義である。
	ワイド文字	wchar_t引数を，文字を表すバイト配列に変換し，結果の文字を出力する。フィールド幅が指定されていて，結果の文字がフィールド幅より少ない文字数の場合は，指定された幅になるようにスペース文字が埋め込まれる。精度が指定されている場合は，動作は未定義である。
s	バイト	char型の文字配列を指すポインタである引数が必要である。この引数は，ヌル文字が検出されるか，精度指定によって示される文字数になるまで，文字を出力するために使用される。精度指定が 0 の場合や，指定されていない場合は，ヌル文字までのすべての文字が出力される。 この変換指定子の前に省略可能な文字 l(小文字のエル) が指定されている場合は，この指定子はワイド文字コードの配列をマルチバイト文字に変換し，マルチバイト文字を出力する。wchar_t型のワイド文字の配列を指すポインタが引数として必要である。ヌル・ワイド文字が検出されるか，精度指定によって示されるバイト数になるまで，文字が出力される。精度指定が省略されている場合や，変換されたバイト配列のサイズより大きい場合は，ワイド文字の配列の末尾にヌル・ワイド文字が必要である。
	ワイド文字	s 指定子の前に l (小文字のエル) が指定されていない場合は，この指定子は各マルチバイト文字に対してmbrtowcを呼び出したかのように，マルチバイト文字の配列を変換し，ヌル・ワイド文字が検出されるか，または精度指定によって示されるワイド文字の文字数になるまで，結果の文字を出力する。精度指定が省略されている場合や，変換された文字配列のサイズより大きい場合は，変換された配列の末尾にヌル・ワイド文字が必要である。 この変換指定子の前に l が指定されている場合は，引数はwchar_t型の配列を指すポインタである。ヌル・ワイド文字が検出されるか，精度指定によって示されるワイド文字の文字数になるまで，この配列から文字が出力される。精度指定が省略されている場合や，配列のサイズより大きい場合は，配列の末尾にヌル・ワイド文字が必要である。

[1] **バイト**または**ワイド文字**。ただし，どちらも示されていない場合は，指定子の説明は両方に適用される。

(次ページに続く)

表 2–5 (続き) 書式設定された出力の変換指定子

指定子	出力の種類[1]	説明
S	バイト	ワイド文字コードの配列をマルチバイト文字に変換し，マルチバイト文字を出力する。wchar_t型のワイド文字の配列を指すポインタが引数として必要である。ヌル・ワイド文字が検出されるか，または精度指定によって示されるバイト数になるまで，文字が出力される。精度指定が省略されている場合や，変換されたバイト配列のサイズより大きい場合は，ワイド文字の配列の末尾にヌル・ワイド文字が必要である。
	ワイド文字	引数はwchar_t型の配列を指すポインタである。ヌル・ワイド文字が検出されるか，精度指定によって示されるワイド文字の文字数になるまで，この配列から文字が出力される。精度指定が省略されている場合や，配列のサイズより大きい場合は，配列の末尾にヌル・ワイド文字が必要である。
p		voidを指すポインタが引数として必要である。ポインタの値が 16 進数として出力される。
n		整数を指すポインタが引数として必要である。書式設定された出力関数に対するこの呼び出しでこれまで出力ストリームに書き込まれた文字数が整数に代入される。引数は変換されない。
%		パーセント記号を出力する。変換は実行されない。完全な変換指定は%%である。

[1] **バイト**または**ワイド文字**。ただし，どちらも示されていない場合は，指定子の説明は両方に適用される。

2.5 端末 I/O

HP Cでは，通常は端末 (対話型ジョブの場合) またはバッチ・ストリーム (バッチ・ジョブの場合) に関連付けられている論理デバイスとの間で I/O を実行するために，3 つのファイル・ポインタが定義されています。OpenVMS 環境では， 3 つのパーマネント・プロセス・ファイル SYS$INPUT， SYS$OUTPUT， SYS$ERROR は，対話型ジョブの場合もバッチ・ジョブの場合も同じ機能を実行します。端末 I/O は，端末 I/O とバッチ・ストリーム I/O の両方を示します。ファイル・ポインタstdin，stdout，stderrは，#includeプリプロセッサ・ディレクティブを使用して<stdio.h>ヘッダ・ファイルを取り込むときに定義されます。

stdinファイル・ポインタは，入力を実行するために端末に関連付けられます。このファイルは SYS$INPUT に対応します。stdoutファイル・ポインタは，出力を実行するために端末に関連付けられます。このファイルは SYS$OUTPUT に対応します。stderrファイル・ポインタは，実行時エラーを報告するために端末に関連付けられます。このファイルは SYS$ERROR に対応します。

端末を参照する 3 つのファイル記述子があります。ファイル記述子 0 は SYS$INPUT に対応し， 1 は SYS$OUTPUT に対応し， 2 は SYS$ERROR に対応します。

端末で I/O を実行する場合，標準 I/O 関数とマクロを使用でき (引数としてポインタstdin，stdout，stderrを指定します)， UNIX I/O 関数を使用することもでき (引数として対応するファイル記述子を指定します)，端末 I/O 関数とマクロを指定することもできます。機能的にいずれかの I/O が他の種類の I/O より優れているというわけではありません。ただし，端末 I/O 関数の場合は引数がないため，必要なキーストロークを削減できます。

2.6 プログラムの例

ここでは，アプリケーションで I/O 関数を使用する方法を示すために，いくつかのサンプル・プログラムを示します。

例 2–1 はprintf関数を示しています。

例 2–1 変換指定の出力

```
/*      CHAP_2_OUT_CONV.C                                   */

/* This program uses the printf function to print the      */
/* various conversion specifications and their effect      */
/* on the output.                                          */

/* Include the proper header files in case printf has      */
/* to return EOF.                                          */

#include <stdlib.h>
#include <stdio.h>
#include <wchar.h>

#define WIDE_STR_SIZE 20

main()
{
    double val = 123345.5;
    char c = 'C';
    int i = -1500000000;
    char *s = "thomasina";
    wchar_t wc;
    wchar_t ws[WIDE_STR_SIZE];

    /* Produce a wide character and a wide character string */

    if (mbtowc(&wc, "W", 1) == -1) {
        perror("mbtowc");
        exit(EXIT_FAILURE);
    }

    if (mbstowcs(ws, "THOMASINA", WIDE_STR_SIZE) == -1) {
        perror("mbstowcs");
        exit(EXIT_FAILURE);
    }
```

(次ページに続く)

例 2–1 (続き) 変換指定の出力

```
    /* Print the specification code, a colon, two tabs, and the  */
    /* formatted output value delimited by the angle bracket     */
    /* characters (<>).                                          */

    printf("%%9.4f:\t\t<%9.4f>\n", val);
    printf("%%9f:\t\t<%9f>\n", val);
    printf("%%9.0f:\t\t<%9.0f>\n", val);
    printf("%%-9.0f:\t\t<%-9.0f>\n\n", val);

    printf("%%11.6e:\t\t<%11.6e>\n", val);
    printf("%%11e:\t\t<%11e>\n", val);
    printf("%%11.0e:\t\t<%11.0e>\n", val);
    printf("%%-11.0e:\t\t<%-11.0e>\n\n", val);

    printf("%%11g:\t\t<%11g>\n", val);
    printf("%%9g:\t\t<%9g>\n\n", val);

    printf("%%d:\t\t<%d>\n", c);
    printf("%%c:\t\t<%c>\n", c);
    printf("%%o:\t\t<%o>\n", c);
    printf("%%x:\t\t<%x>\n\n", c);

    printf("%%d:\t\t<%d>\n", i);
    printf("%%u:\t\t<%u>\n", i);
    printf("%%x:\t\t<%x>\n\n", i);

    printf("%%s:\t\t<%s>\n", s);
    printf("%%-9.6s:\t\t<%-9.6s>\n", s);
    printf("%%-*.*s:\t\t<%-*.*s>\n", 9, 5, s);
    printf("%%6.0s:\t\t<%6.0s>\n\n", s);
    printf("%%C:\t\t<%C>\n", wc);
    printf("%%S:\t\t<%S>\n", ws);
    printf("%%-9.6S:\t\t<%-9.6S>\n", ws);
    printf("%%-*.*S:\t\t<%-*.*S>\n", 9, 5, ws);
    printf("%%6.0S:\t\t<%6.0S>\n\n", ws);
}
```

例 2–1 を実行すると，次の出力が生成されます。

```
$ RUN  EXAMPLE
%9.4f:          <123345.5000>
%9f:            <123345.500000>
%9.0f:          <   123346>
%-9.0f:         <123346   >

%11.6e:         <1.233455e+05>
%11e:           <1.233455e+05>
%11.0e:         <      1e+05>
%-11.0e:        <1e+05      >

%11g:           <     123346>
%9g:            <   123346>
```

```
%d:             <67>
%c:             <C>
%o:             <103>
%x:             <43>

%d:             <-1500000000>
%u:             <2794967296>
%x:             <a697d100>

%s:             <thomasina>
%-9.6s:         <thomas   >
%-*.*s:         <thoma    >
%6.0s:          <      >

%C:             <W>
%S:             <THOMASINA>
%-9.6S:         <THOMAS   >
%-*.*S:         <THOMA    >
%6.0S:          <      >
$
```

例 2–2 は，fopen，ftell，sprintf，fputs，fseek，fgets，fclose関数の使い方を示しています。

例 2–2 標準 I/O 関数の使用

```
/*      CHAP_2_STDIO.C  */

/* This program establishes a file pointer, writes lines from   */
/* a buffer to the file, moves the file pointer to the second   */
/* record, copies the record to the buffer, and then prints    */
/* the buffer to the screen.                                    */

#include <stdio.h>
#include <stdlib.h>

main()
{
    char buffer[32];
    int i,
        pos;
    FILE *fptr;

    /*  Set file pointer.           */
    fptr = fopen("data.dat", "w+");
    if (fptr == NULL) {
        perror("fopen");
        exit(EXIT_FAILURE);
    }
```

(次ページに続く)

例 2–2 (続き) 標準 I/O 関数の使用

```
    for (i = 1; i < 5; i++) {
        if (i == 2)     /*  Get position of record 2. */
            pos = ftell(fptr);
        /*  Print a line to the buffer.  */
        sprintf(buffer, "test data line %d\n", i);
        /*  Print buffer to the record.  */
        fputs(buffer, fptr);
    }

    /*  Go to record number 2.       */
    if (fseek(fptr, pos, 0) < 0) {
        perror("fseek");        /*  Exit on fseek error. */
        exit(EXIT_FAILURE);
    }

    /*  Read record 2 in the buffer. */
    if (fgets(buffer, 32, fptr) == NULL) {
        perror("fgets");        /*  Exit on fgets error.  */
        exit(EXIT_FAILURE);
    }
    /*  Print the buffer.       */
    printf("Data in record 2 is: %s", buffer);
    fclose(fptr);       /*  Close the file.     */
}
```

例 2–2 を実行すると，次の結果が生成されます。

```
$ RUN  EXAMPLE
Data in record 2 is: test data line 2
```

例 2–2 から DATA.DAT への出力は次のようになります。

```
test data line 1
test data line 2
test data line 3
test data line 4
```

例 2–3 ワイド文字 I/O 関数の使用

```
/*      CHAP_2_WC_IO.C                                          */

/* This program establishes a file pointer, writes lines from   */
/* a buffer to the file using wide-character codes, moves the   */
/* file pointer to the second record, copies the record to the  */
/* wide-character buffer, and then prints the buffer to the     */
/* screen.                                                      */

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

(次ページに続く)

例 2–3 (続き) ワイド文字 I/O 関数の使用

```
main()
{
    char flat_buffer[32];
    wchar_t wide_buffer[32];
    wchar_t format[32];
    int i,
        pos;
    FILE *fptr;

    /*  Set file pointer.           */
    fptr = fopen("data.dat", "w+");
    if (fptr == NULL) {
        perror("fopen");
        exit(EXIT_FAILURE);
    }

    for (i = 1; i < 5; i++) {
        if (i == 2)     /*  Get position of record 2. */
            pos = ftell(fptr);
        /*  Print a line to the buffer.  */
        sprintf(flat_buffer, "test data line %d\n", i);
        if (mbstowcs(wide_buffer, flat_buffer, 32) == -1) {
            perror("mbstowcs");
            exit(EXIT_FAILURE);
        }

        /*  Print buffer to the record.  */
        fputws(wide_buffer, fptr);
    }

    /*  Go to record number 2. */
    if (fseek(fptr, pos, 0) < 0) {
        perror("fseek");        /*  Exit on fseek error. */
        exit(EXIT_FAILURE);
    }

    /*  Put record 2 in the buffer.  */
    if (fgetws(wide_buffer, 32, fptr) == NULL) {
        perror("fgetws");       /*  Exit on fgets error. */
        exit(EXIT_FAILURE);
    }
    /*  Print the buffer. */
    printf("Data in record 2 is: %S", wide_buffer);
    fclose(fptr);               /*  Close the file.  */
}
```

例 2–3 を実行すると，次の結果が生成されます。

```
$ RUN  EXAMPLE
Data in record 2 is: test data line 2
```

例 2–3 から DATA.DAT への出力は次のようになります。

```
test data line 1
test data line 2
test data line 3
test data line 4
```

例 2–4 は，1 つのファイルにアクセスするための，ファイル・ポインタとファイル記述子の両方の使い方を示しています。

例 2–4 ファイル記述子とファイル・ポインタを使用した I/O

```
/*      CHAP_2_FILE_DIS_AND_POINTER.C                          */

/* The following example creates a file with variable-length    */
/* records (rfm=var) and the carriage-return attribute (rat=cr).*/
/*                                                              */
/* The program uses creat to create and open the file, and      */
/* fdopen to associate the file descriptor with a file pointer. */
/* After using the fdopen function, the file must be referenced */
/* using the Standard I/O functions.                            */

#include <unixio.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define ERROR 0
#define ERROR1 -1
#define BUFFSIZE 132

main()
{
    char buffer[BUFFSIZE];
    int fildes;
    FILE *fp;

    if ((fildes = creat("data.dat", 0, "rat=cr",
                        "rfm=var")) == ERROR1) {
        perror("DATA.DAT: creat() failed\n");
        exit(EXIT_FAILURE);
    }

    if ((fp = fdopen(fildes, "w")) == NULL) {
        perror("DATA.DAT: fdopen() failed\n");
        exit(EXIT_FAILURE);
    }
    while (fgets(buffer, BUFFSIZE, stdin) != NULL)
        if (fwrite(buffer, strlen(buffer), 1, fp) == ERROR) {
            perror("DATA.DAT: fwrite() failed\n");
            exit(EXIT_FAILURE);
        }
```

(次ページに続く)

例 2–4 (続き) ファイル記述子とファイル・ポインタを使用した I/O

```
    if (fclose(fp) == EOF) {
        perror("DATA.DAT: fclose() failed\n");
        exit(EXIT_FAILURE);
    }
}
```

3

文字，文字列，引数リスト関数

表 3–1 は，HP C Run-Time Library (RTL) で提供される文字，文字列，および引数リスト関数を示しています。この章ではこれらの関数について説明しますが，各関数の詳細については，『HP C ランタイム・ライブラリ・リファレンス・マニュアル (下巻)』「リファレンス・セクション」を参照してください。

表 3–1 文字，文字列，引数リスト関数

関数	説明
文字の分類	
isalnum, iswalnum	引数が現在のロケールで英数字のいずれかである場合は，0 以外の整数を返す。
isalpha, iswalpha	引数が現在のロケールで英字のいずれかである場合は，0 以外の整数を返す。
isascii	引数が ASCII 文字である場合は，0 以外の整数を返す。
iscntrl, iswcntrl	引数が現在のロケールで制御文字である場合は，0 以外の整数を返す。
isdigit, iswdigit	引数が現在のロケールで数字である場合は，0 以外の整数を返す。
isgraph, iswgraph	引数が現在のロケールでグラフィック文字である場合は，0 以外の整数を返す。
islower, iswlower	引数が現在のロケールで小文字である場合は，0 以外の整数を返す。
isprint, iswprint	引数が現在のロケールでプリント文字である場合は，0 以外の整数を返す。
ispunct, iswpunct	引数が現在のロケールで句読点文字である場合は，0 以外の整数を返す。
isspace, iswspace	引数が現在のロケールで空白文字である場合は，0 以外の整数を返す。
isupper, iswupper	引数が現在のロケールで大文字である場合は，0 以外の整数を返す。
iswctype	指定されたプロパティを引数が持つ場合，0 以外の整数を返す。
isxdigit, iswxdigit	引数が 16 進数の (0 ～ 9，A ～ F，a ～ f) である場合は，0 以外の整数を返す。

(次ページに続く)

表 3–1 (続き) 文字，文字列，引数リスト関数

関数	説明
文字変換	
btowc	1 バイトのマルチバイト文字を初期シフト状態でワイド文字に変換する。
ecvt, fcvt, gcvt	引数を ASCII 数字列に変換し，末尾にヌル文字を付加し，文字列のアドレスを返す。
index, rindex	文字列から文字を検索する。
mblen, mbrlen	マルチバイト文字に含まれるバイト数を判断する。
mbsinit	mbstate_tオブジェクトが初期変換状態を記述するかどうかを判断する。
mbstowcs	マルチバイト文字シーケンスを対応するコード・シーケンスに変換する。
toascii	8 ビットの ASCII 文字である引数を 7 ビットの ASCII 文字に変換する。
tolower, _tolower, towlower	大文字の引数を小文字に変換する。
toupper, _toupper, towupper	小文字の引数を大文字に変換する。
towctrans	指定されたマッピング記述子に従って， 1 文字のワイド文字を別の文字に変換する。
wcstombs	マルチバイト文字に対応するワイド文字コード・シーケンスをマルチバイト文字シーケンスに変換する。
wctob	ワイド文字が 1 バイトのマルチバイト文字に対応するかどうか判断し，そのマルチバイト文字表現を返す。
wctomb	ワイド文字をマルチバイト文字表現に変換する。
wctrans	指定されたプロパティに対応するマッピングの記述を返す。この記述は後でtowctransの呼び出しで使用できる。
wctype	現在のロケールで定義されている正しい文字クラスを，wctype_t型のオブジェクトに変換する。
文字列の操作	
atof	指定された文字列を倍精度数値に変換する。
atoi, atol	ASCII 文字列を適切な数値に変換する。
atoll, atoq *(Integrity， Alpha)*	ASCII 文字列を__int64に変換する。
basename	パス名の最後のコンポーネントを返す。
dirname	ファイル・パス名の親ディレクトリ名を報告する。
strcat, strncat, wcscat, wcsncat	1 つの文字列を別の文字列の末尾に追加する。
strchr, strrchr, wcschr, wcsrchr	ヌル区切り文字列の中で指定された文字の最初または最後の発生箇所のアドレスを返す。

(次ページに続く)

表 3–1 (続き) 文字，文字列，引数リスト関数

関数	説明
文字列の操作	
strcmp, strncmp, strcoll, wcscmp, wcsncmp, wcscoll	2 つの文字列を比較し，最初の文字列の各文字の値が 2 番目の文字列の値より小さいか，等しいか，大きいかを示す。結果に応じて負の整数，0，正の整数のいずれかを返す。
strcpy, strncpy, wcscpy, wcsncpy	1 つの文字列全体またはその一部を別の文字列にコピーする。
strxfrm, wcsxfrm	strcmpまたはwcscmp関数を使用して比較できるように，マルチバイト文字列を別の文字列に変換する。
strcspn, wcscspn	指定された文字セットに含まれる文字を文字列から検索する。
strlen, wcslen	文字列の長さを返す。返される長さには，文字列の末尾のヌル文字(\0)は含まれない。
strpbrk, wcspbrk	指定された文字セットに含まれる文字の発生箇所を文字列から検索する。
strspn, wcsspn	指定された文字セットに含まれない文字の発生箇所を文字列から検索する。
strstr, wcswcs	指定された文字セットに含まれる文字の最初の発生箇所を文字列から検索する。
strtod, wcstod	指定された文字列を倍精度の数値に変換する。
strtok, wcstok	テキスト・トークンを指定された文字列に配置する。
strtol, wcstol	文字列の最初の部分を符号付きロング整数に変換する。
strtoll, strtoq *(Integrity，Alpha)*	文字列の最初の部分を符号付き__int64に変換する。
strtoul, wcstoul	文字列の最初の部分を符号なしロング整数に変換する。
strtoull, strtouq *(Integrity，Alpha)*	文字列を指すポインタによって示される文字列の最初の部分を符号なし__int64に変換する。
文字列の処理 — バイナリ・データへのアクセス	
bcmp	バイト列を比較する。
bcopy	バイト列をコピーする。
bzero	ヌル文字をバイト列にコピーする。
memchr, wmemchr	検索するオブジェクトの最初の長さの範囲内から，指定されたバイトの最初の発生箇所を検索する。
memcmp, wmemcmp	2 つのオブジェクトをバイト単位で比較する。
memcpy, memmove, wmemcpy, wmemmove	指定されたバイト数を 1 つのオブジェクトから別のオブジェクトにコピーする。
memset, wmemset	指定されたオブジェクト内の指定されたバイト数を指定された値に設定する。
引数リストの処理 — 可変長引数リストへのアクセス	
va_arg	引数リストの次のアイテムを返す。
va_count	引数リストでクォドワード*(Alpha only)*の数を返す。

(次ページに続く)

表 3–1 (続き) 文字，文字列，引数リスト関数

関数	説明
引数リストの処理 — 可変長引数リストへのアクセス	
va_end	va_startセッションを終了する。
va_start, va_start_1	変数を引数リストの先頭に初期化する。
vfprintf, vprintf, vsprintf	引数リストをもとに書式設定された出力を印刷する。

3.1 文字分類関数

文字分類関数は，論理操作を実行する対象となる 1 つの引数を受け付けます。引数はどのような値でも構いません。 ASCII 文字である必要はありません。isascii関数は，引数が ASCII 文字 (8 進数の 0 〜 177) であるかどうかを判断します。他の関数は，引数がグラフィック文字や数字など，特定の種類の ASCII 文字であるかどうかを確認します。isw*関数はワイド文字を判定します。文字分類情報はプログラムの現在のロケールの LC_CTYPE カテゴリにあります。

どの関数の場合も，戻り値が正の値の場合は TRUE を示します。戻り値が 0 の場合は FALSE を示します。

この後の表では，文字分類関数を簡単に参照できるように，表 3–2 に示す番号が各関数に割り当てられています。

表 3–2 文字分類関数

関数番号	関数	関数番号	関数
1	isalnum	7	islower
2	isalpha	8	isprint
3	isascii	9	ispunct
4	iscntrl	10	isspace
5	isdigit	11	isupper
6	isgraph	12	isxdigit

表 3–3 は，各 ASCII 文字に対して TRUE という戻り値を返す関数の番号 (表 3–2 で割り当てられた番号) を示しています。数値コードは，各 ASCII 文字の 8 進コードを表しています。

表 3–3 ASCII 文字と文字分類関数

ASCII 値	関数番号	ASCII 値	関数番号
NUL 00	3,4	@ 100	3,6,8,9
SOH 01	3,4	A 101	1,2,3,6,8,11,12
STX 02	3,4	B 102	1,2,3,6,8,11,12
ETX 03	3,4	C 103	1,2,3,6,8,11,12
EOT 04	3,4	D 104	1,2,3,6,8,11,12
ENQ 05	3,4	E 105	1,2,3,6,8,11,12
ACK 06	3,4	F 106	1,2,3,6,8,11,12
BEL 07	3,4	G 107	1,2,3,6,8,11
BS 10	3,4	H 110	1,2,3,6,8,11
HT 11	3,4,10	I 111	1,2,3,6,8,11
LF 12	3,4,10	J 112	1,2,3,6,8,11
VT 13	3,4,10	K 113	1,2,3,6,8,11
FF 14	3,4,10	L 114	1,2,3,6,8,11
CR 15	3,4,10	M 115	1,2,3,6,8,11
SO 16	3,4	N 116	1,2,3,6,8,11
SI 17	3,4	O 117	1,2,3,6,8,11
DLE 20	3,4	P 120	1,2,3,6,8,11
DC1 21	3,4	Q 121	1,2,3,6,8,11
DC2 22	3,4	R 122	1,2,3,6,8,11
DC3 23	3,4	S 123	1,2,3,6,8,11
DC4 24	3,4	T 124	1,2,3,6,8,11
NAK 25	3,4	U 125	1,2,3,6,8,11
SYN 26	3,4	V 126	1,2,3,6,8,11
ETB 27	3,4	W 127	1,2,3,6,8,11
CAN 30	3,4	X 130	1,2,3,6,8,11
EM 31	3,4	Y 131	1,2,3,6,8,11
SUB 32	3,4	Z 132	1,2,3,6,8,11
ESC 33	3,4	[133	3,6,8,9
FS 34	3,4	\ 134	3,6,8,9
GS 35	3,4	] 135	3,6,8,9
RS 36	3,4	^ 136	3,6,8,9
US 37	3,4	– 137	3,6,8,9
SP 40	3,8,10	?` 140	3,6,8,9
! 41	3,6,8,9	a 141	1,2,3,6,7,8,12

(次ページに続く)

表 3–3 (続き) ASCII 文字と文字分類関数

ASCII 値	関数番号	ASCII 値	関数番号
" 42	3,6,8,9	b 142	1,2,3,6,7,8,12
# 43	3,6,8,9	c 143	1,2,3,6,7,8,12
$ 44	3,6,8,9	d 144	1,2,3,6,7,8,12
% 45	3,6,8,9	e 145	1,2,3,6,7,8,12
& 46	3,6,8,9	f 146	1,2,3,6,7,8,12
′ 47	3,6,8,9	g 147	1,2,3,6,7,8
(50	3,6,8,9	h 150	1,2,3,6,7,8
) 51	3,6,8,9	i 151	1,2,3,6,7,8
* 52	3,6,8,9	j 152	1,2,3,6,7,8
+ 53	3,6,8,9	k 153	1,2,3,6,7,8
’ 54	3,6,8,9	l 154	1,2,3,6,7,8
- 55	3,6,8,9	m 155	1,2,3,6,7,8
?. 56	3,6,8,9	n 156	1,2,3,6,7,8
/ 57	3,6,8,9	o 157	1,2,3,6,7,8
0 60	1,3,5,6,8,12	p 160	1,2,3,6,7,8
1 61	1,3,5,6,8,12	q 161	1,2,3,6,7,8
2 62	1,3,5,6,8,12	r 162	1,2,3,6,7,8
3 63	1,3,5,6,8,12	s 163	1,2,3,6,7,8
4 64	1,3,5,6,8,12	t 164	1,2,3,6,7,8
5 65	1,3,5,6,8,12	u 165	1,2,3,6,7,8
6 66	1,3,5,6,8,12	v 166	1,2,3,6,7,8
7 67	1,3,5,6,8,12	w 167	1,2,3,6,7,8
8 70	1,3,5,6,8,12	x 170	1,2,3,5,6,8
9 71	1,3,5,6,8,12	y 171	1,2,3,5,6,8
: 72	3,6,8,9	z 172	1,2,3,5,6,8
; 73	3,6,8,9	{ 173	3,6,8,9
< 74	3,6,8,9	| 174	3,6,8,9
= 75	3,6,8,9	} 175	3,6,8,9
> 76	3,6,8,9	?~ 176	3,6,8,9
? 77	3,6,8,9	DEL 177	3,4

例 3–1 は文字分類関数の使い方を示しています。

例 3–1 文字分類関数

```
/*       CHAP_3_CHARCLASS.C                                    */

/* This example uses the isalpha, isdigit, and isspace         */
/* functions to count the number of occurrences of letters,    */
/* digits, and white-space characters entered through the      */
/* standard input (stdin).                                     */

#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>

main()
{
    char c;
    int i = 0,
        j = 0,
        k = 0;

    while ((c = getchar()) != EOF) {
        if (isalpha(c))
            i++;
        if (isdigit(c))
            j++;
        if (isspace(c))
            k++;
    }

    printf("Number of letters: %d\n", i);
    printf("Number of digits:  %d\n", j);
    printf("Number of spaces:  %d\n", k);
}
```

次の例は，例 3–1 へのサンプル入力とサンプル出力を示しています。

```
$ RUN  EXAMPLE1
I saw 35 people riding bicycles on Main Street. Return
Ctrl/Z
Number of letters: 36
Number of digits:  2
Number of spaces:  8
$
```

3.2 文字変換関数

文字変換関数は，ある種類の文字を別の種類に変換します。次の関数があります。

```
ecvt         _tolower
fcvt         toupper
gcvt         _toupper
mbtowc       towctrans
mbrtowc      wctrans
mbsrtowcs    wcrtomb
toascii      wcsrtombs
tolower
```

これらの各関数の詳細については，『HP C ランタイム・ライブラリ・リファレンス・マニュアル (下巻)』「リファレンス・セクション」を参照してください。

例 3–2 は，ecvt関数の使い方を示しています。

例 3–2 倍精度値から ASCII 文字列への変換

```
/*     CHAP_3_CHARCONV.C                                        */

/* This program uses the ecvt function to convert a double     */
/* value to a string. The program then prints the string.      */

#include <stdio.h>
#include <stdlib.h>
#include <unixlib.h>
#include <string.h>

main()
{
    double val;         /* Value to be converted */

    int sign,           /* Variables for sign   */
        point;          /* and decimal place    */

    /*  Array for the converted string  */
    static char string[20];

    val = -3.1297830e-10;

    printf("original value: %e\n", val);
    if (sign)
        printf("value is negative\n");
    else
        printf("value is positive\n");
    printf("decimal point at %d\n", point);
}
```

例 3–2 からの出力は次のようになります。

```
$ RUN  EXAMPLE2
original value: -3.129783e-10
converted string: 31298
value is negative
decimal point at -9
$
```

例 3–3 はtoupper関数とtolower関数の使い方を示しています。

例 3–3 大文字と小文字の変更

```
/*     CHAP_3_CONV_UPPERLOWER.C                                  */

/* This program uses the functions toupper and tolower to       */
/* convert uppercase to lowercase and lowercase to uppercase    */
/* using input from the standard input (stdin).                 */

#include <ctype.h>
#include <stdio.h>      /*  To use EOF identifier */
#include <stdlib.h>

main()
{
    char c,
         ch;

    while ((c = getchar()) != EOF) {
        if (c >= 'A' && c <= 'Z')
            ch = tolower(c);
        else
            ch = toupper(c);
        putchar(ch);
    }
}
```

次の例は例 3–3 へのサンプル入力とサンプル出力を示しています。

```
$ RUN  EXAMPLE3
LET'S GO TO THE welcome INN. Ctrl/Z
let's go to the WELCOME inn.
$
```

3.3 文字列および引数リスト関数

HP C RTL には，文字列を操作する関数グループがあります。これらの関数には，文字列を連結する関数，文字列から特定の文字を検索する関数，2 つの文字列が等しいかどうか判断する関数など，その他の比較を実行する関数があります。

HP C RTL には，バイナリ・データが格納されているバッファをコピーするための関数も用意されています。

<varargs.h>および<stdarg.h>ヘッダ・ファイルに定義および宣言されている関数は，可変長引数リストにアクセスするために使用できます。<stdarg.h>関数は ANSI C 標準で定義されており，<varargs.h>に定義されている関数より高い移植性を備えています。

たとえばprintfやexeclなどの RTL 関数では，可変長引数リストを使用します。可変長引数リストを使用するユーザ定義関数で<varargs.h>や<stdarg.h>を使用しない場合は，マシン間で引数の受け渡し規則が異なるため，移植することができません。

<stdarg.h>ヘッダ・ファイルには，*va_alist*およびva_dclが含まれていません。次の構文の例は，<stdarg.h>を使用する場合の構文を示しています。

```
function_name(int arg1, ...)
{

    va_list ap;
    ...
```

<varargs.h>を使用する場合は，次のようになります。

- 識別子*va_alist*は関数定義に含まれるパラメータです。
- va_dclはパラメータ*va_alist*を宣言します。これはセミコロン(;)で終了しない宣言です。
- va_list型は，リスト内を移動するために使用される変数の宣言で使用されます。<varargs.h>を使用する場合は，va_list型の変数を少なくとも 1 つ宣言する必要があります。

これらの名前および宣言の構文は次のとおりです。

```
function_name(int arg1, . . . )
{

    va_list ap;

        .
        .
        .
```

3.4 プログラムの例

例 3–4 は，strcat関数とstrncat関数の使い方を示しています。

例 3–4 2 つの文字列の結合

```
/*        CHAP_3_CONCAT.C                                        */

/*  This example uses strcat and strncat to concatenate two      */
/*  strings.                                                     */

#include <stdio.h>
#include <string.h>

main()
{
    static char string1[80] = "Concatenates ";
    static char string2[] = "two strings ";
    static char string3[] = "up to a maximum of characters.";
    static char string4[] = "imum number of characters";

    printf("strcat:\t%s\n", strcat(string1, string2));
    printf("strncat ( 0):\t%s\n", strncat(string1, string3, 0));
    printf("strncat (11):\t%s\n", strncat(string1, string3, 11));
    printf("strncat (40):\t%s\n", strncat(string1, string4, 40));
}
```

例 3–4 は次の出力を生成します。

```
$ RUN  EXAMPLE1
strcat: Concatenates two strings
strncat ( 0): Concatenates two strings
strncat (11): Concatenates two strings up to a max
strncat (40): Concatenates two strings up to a maximum number of characters.
$
```

例 3–5 はstrcspn関数の使い方を示しています。

例 3–5 strcspn 関数に対する 4 つの引数

```
/*        CHAP_3_STRCSPN.C                                       */

/*  This example shows how strcspn interprets four               */
/*  different kinds of arguments.                                */

#include <stdio.h>

main()
{

    printf("strcspn with null charset: %d\n",
            strcspn("abcdef", ""));

    printf("strcspn with null string: %d\n",
            strcspn("", "abcdef"));
```

(次ページに続く)

例 3–5（続き） strcspn 関数に対する 4 つの引数

```
    printf("strcspn(\"xabc\", \"abc\"): %d\n",
           strcspn("xabc", "abc"));

    printf("strcspn(\"abc\", \"def\"): %d\n",
            strcspn("abc", "def"));
}
```

例 3–5 を呼び出すと，strcspn.outファイルに次のサンプル出力が生成されます。

```
$ RUN  EXAMPLE2

strcspn with null charset:  6
strcspn with null string:  0
strcspn("xabc","abc"):  1
strcspn("abc","def"):  3
```

例 3–6 は，<stdarg.h>関数および定義の使い方を示しています。

例 3–6 <stdarg.h>関数と定義の使用

```
/*        CHAP_3_STDARG.C                                        */

/* This routine accepts a variable number of string arguments,  */
/* preceded by a count of the number of such strings. It        */
/* allocates enough space in which to concatenate all of the    */
/* strings, concatenates them together, and returns the address */
/* of the new string. It returns NULL if there are no string    */
/* arguments, or if they are all null strings.                  */

#include <stdarg.h>       /* Include appropriate header files   */
#include <stdlib.h>       /* for the "example" call in main.    */
#include <string.h>
#include <stdio.h>

/* NSTRINGS is the maximum number of string arguments accepted  */
/* (arbitrary).                                                 */

#define NSTRINGS 10

char *concatenate(int n,...)
{
    va_list ap;         /* Declare the argument pointer. */

    char *list[NSTRINGS],
        *string;
    int index = 0,
        size = 0;

    /* Check that the number of arguments is within range.   */
```

（次ページに続く）

例 3–6 (続き) <stdarg.h>関数と定義の使用

```
    if (n <= 0)
        return NULL;
    if (n > NSTRINGS)
        n = NSTRINGS;

    va_start(ap, n);    /* Initialize the argument pointer.  */

    do {
        /* Extract the next argument and save it. */

        list[index] = va_arg(ap, char *);

        size += strlen(list[index]);
    } while (++index < n);

    va_end(ap); /* Terminate use of ap. */

    if (size == 0)
        return NULL;

    string = malloc(size + 1);
    string[0] = '\0';

    /* Append each argument to the end of the growing result    */
    /*  string.                                                 */

    for (index = 0; index < n; ++index)
        strcat(string, list[index]);

    return string;
}

/* An example of calling this routine is */

main() {
    char *ret_string ;

    ret_string = concatenate(7, "This ", "message ", "is ",
                               "built with ", "a", " variable arg",
                               " list.") ;

    puts(ret_string) ;
}
```

例 3–6 を呼び出すと，次の出力が生成されます。

```
This message is built with a variable arg list.
```

4

エラー処理とシグナル処理

表 4–1 は，HP C Run-Time Library (RTL) で提供されるすべてのエラー処理関数とシグナル処理関数を示しています。各関数の詳細については，『HP C ランタイム・ライブラリ・リファレンス・マニュアル (下巻)』「リファレンス・セクション」を参照してください。

表 4–1 エラー処理関数とシグナル処理関数

関数	説明
abort	プログラムの実行を終了するシグナル SIGABRT を送出する。
assert	診断情報をプログラムに渡す。
atexit	プログラムの終了時に呼び出す関数を登録する。
exit, _exit	現在のプログラムを終了する。
perror	現在のerrnoの値を記述する短いエラー・メッセージをstderrに出力する。
strerror	errnoのエラー・コードをエラー・メッセージ文字列に変換する。
alarm	引数によって示される秒数が経過した後，シグナル SIGALARM を起動プロセスに送信する。
gsignal	指定されたソフトウェア・シグナルを生成する。
kill	process IDによって指定されるプロセスに SIGKILL シグナルを送信する。
longjmp	正常終了せずに，制御をネストした一連の関数呼び出しから定義済みポイントに返す。
pause	シグナルを受信するまでプロセスを待機させる。
raise	指定されたシグナルを生成する。
setjmp	正常終了せずに，ネストした一連の関数呼び出しから後で制御を転送するためにコンテキストを設定する。
sigaction	シグナルの配布時に実行する動作を指定する。
sigaddset	指定された個々のシグナルを追加する。
sigblock	引数に指定されたシグナルを，配布がブロックされている現在のシグナル・セットに追加する。
sigdelset	指定された個々のシグナルを削除する。
sigemptyset	すべてのシグナルを除外するようにシグナル・セットを初期化する。
sigfillset	すべてのシグナルを含むようにシグナル・セットを初期化する。
sighold	指定されたシグナルを，呼び出し元プロセスのシグナル・マスクに追加する。
sigignore	指定されたシグナルの処理方法として， SIG_IGN を設定する。

(次ページに続く)

表 4–1 (続き) エラー処理関数とシグナル処理関数

関数	説明
sigismember	指定されたシグナルがシグナル・セットのメンバであるかどうか判定する。
siglongjmp	シグナルを処理して非ローカル・ジャンプを実行する。
sigmask	指定されたシグナル番号に対してマスクを作成する。
signal	シグナルを検出するか，または無視する。
sigpause	指定されたシグナル・セットをブロックし，ブロックされていないシグナルを待機する。
sigpending	保留状態のシグナルを調べる。
sigprocmask	現在のシグナル・マスクを設定する。
sigrelse	指定されたシグナルを，呼び出し元プロセスのシグナル・マスクから削除する。
sigsetjmp	非ローカル・ジャンプのジャンプ・ポイントを設定する。
sigsetmask	配布がブロックされるシグナルを設定する。
sigsuspend	ブロックされているシグナル・セットを個別に変更し，シグナルを待機する。
sigtimedwait	呼び出し元スレッドを一時停止し，シグナル通知が到着するのを待機する。
sigvec	特定のシグナルのハンドラを永久的に割り当てる。
sigwait	呼び出し元スレッドを一時停止し，シグナル通知が到着するのを待機する。
sigwaitinfo	呼び出し元スレッドを一時停止し，シグナル通知が到着するのを待機する。
ssignal	特定のシグナルが発生したときに実行する動作を指定することができる。
VAXC$ESTABLISH	HP C RTL の例外処理と互換性のある方法でアプリケーション例外ハンドラを設定する。

4.1 エラー処理

HP C RTL 関数の呼び出しでエラーが発生した場合，その関数は異常終了状態を返します。多くの RTL ルーチンは，外部変数errnoを，障害の理由を示す値に設定します。エラーが発生した場合は，戻り値を常に確認する必要があります。

<errno.h>ヘッダ・ファイルにはerrnoが宣言され，可能なエラー・コードがシンボルで定義されています。<errno.h>ヘッダ・ファイルをプログラムに取り込むことにより，HP C RTL 関数呼び出しの後，特定のエラー・コードを確認することができます。

プログラムの起動時に，errnoの値は 0 です。errnoの値は，多くのHP C RTL 関数によって 0 以外の値に設定される可能性があります。errnoの値がHP C RTL 関数によって 0 にリセットされることはないため，この値を正しく使用するには，HP C RTL 関数を呼び出し，異常終了状態が返された後で使用するようにしなければなりませ

ん。表 4–2 は，HP C RTL がerrnoに代入する可能性のあるシンボル値を示しています。

表 4–2 エラー・コードのシンボル値

シンボル定数	説明
E2BIG	引数リストが長すぎる。
EACCES	アクセス許可が拒否された。
EADDRINUSE	指定されたアドレスはすでに使用されている。
EADDRNOTAVAIL	要求されたアドレスを割り当てることができない。
EAFNOSUPPORT	サポートされていないアドレス・ファミリ。
EAGAIN	これ以上プロセスは存在しない。
EALIGN	アラインメント・エラー。
EALREADY	操作はすでに実行中である。
EBADF	ファイル番号が不正である。
EBADCAT	メッセージ・カタログの形式が不正である。
EBADMSG	壊れたメッセージが検出された。
EBUSY	ビジー状態のデバイスをマウントした。
ECANCELED	操作がキャンセルされた。
ECHILD	子は存在しない。
ECONNABORTED	ソフトウェアで接続アボートが発生した。
ECONNREFUSED	接続が拒否された。
ECONNRESET	接続相手から接続がリセットされた。
EDEADLK	リソース・デッドロックが回避された。
EDESTADDRREQ	宛先アドレスが必要である。
EDOM	算術演算引数。
EDQUOT	ディスク・クォータ超過。
EEXIST	ファイルはすでに存在する。
EFAIL	操作を開始できない。
EFAULT	アドレスが不正である。
EFBIG	ファイルが大きすぎる。
EFTYPE	ファイル・タイプに対して不適切な操作が実行された。
EHOSTDOWN	ホストがダウンしている。
EHOSTUNREACH	ホストにルーティングできない。
EIDRM	識別子が削除された。
EILSEQ	バイト・シーケンスが不正である。
EINPROGRESS	操作が現在実行中である。
EINPROG	非同期操作が現在実行中である。
EINTR	システム呼び出しが中断された。
EINVAL	引数が不正である。

(次ページに続く)

表 4–2 (続き) エラー・コードのシンボル値

シンボル定数	説明
EIO	I/O エラー。
EISCONN	ソケットはすでに接続されている。
EISDIR	ディレクトリである。
ELOOP	シンボリック・リンクのレベルが多すぎる。
EMFILE	オープンされているファイルの数が多すぎる。
EMLINK	リンクの数が多すぎる。
EMSGSIZE	メッセージが長すぎる。
ENAMETOOLONG	ファイル名が長すぎる。
ENETDOWN	ネットワークがダウンしている。
ENETRESET	リセット時にネットワークで接続が切断された。
ENETUNREACH	ネットワークに到達できない。
ENFILE	ファイル・テーブル・オーバフロー。
ENOBUFS	使用可能なバッファ領域が残っていない。
ENODEV	このようなデバイスは存在しない。
ENOENT	このようなファイルまたはディレクトリは存在しない。
ENOEXEC	Exec フォーマット・エラー。
ENOLCK	使用可能なロックが存在しない。
ENOMEM	十分なコアがない。
ENOMSG	適切なタイプのメッセージがない。
ENOPROTOOPT	プロトコルは使用できない。
ENOSPC	デバイスに領域が残されていない。
ENOSYS	この関数は実装されていない。
ENOTBLK	ブロック・デバイスが必要である。
ENOTCONN	ソケットが接続されていない。
ENOTDIR	ディレクトリでない。
ENOTEMPTY	ディレクトリが空でない。
ENOTSOCK	ソケット以外のものに対してソケット操作を実行した。
ENOTSUP	この関数は実装されていない。
ENOTTY	タイプライタでない。
ENWAIT	待機中のプロセスは存在しない。
ENXIO	このようなデバイスまたはアドレスは存在しない。
EOPNOTSUPP	ソケットでこの操作はサポートされていない。
EPERM	オーナでない。
EPFNOSUPPORT	サポートされていないプロトコル・ファミリ。
EPIPE	パイプが壊れている。
EPROCLIM	プロセスの数が多すぎる。
EPROTONOSUPPORT	プロトコルがサポートされていない。

(次ページに続く)

表 4–2 (続き) エラー・コードのシンボル値

シンボル定数	説明
EPROTOTYPE	ソケットに対してプロトコルのタイプが不正である。
ERANGE	結果が大きすぎる。
EREMOTE	パス内のリモートのレベルが多すぎる。
EROFS	読み込み専用ファイル・システム。
ESHUTDOWN	ソケット・シャットダウンの後で送信することはできない。
ESOCKTNOSUPPORT	サポートされていないソケット・タイプ。
ESPIPE	不正なシーク。
ESRCH	このようなプロセスは存在しない。
ESTALE	無効な NFS ファイル・ハンドル。
ETIMEDOUT	接続時間切れ。
ETOOMANYREFS	参照の数が多すぎる，分割できない。
ETXTBSY	テキスト・ファイルが使用中である。
EUSERS	ユーザの数が多すぎる。
EVMSERR	OpenVMS 固有の変換不可能なエラー・コード。
EWOULDBLOCK	I/O 操作はチャネルをブロックする。
EXDEV	クロスデバイス・リンク。

perror関数またはstrerror関数を使用すると，エラー・コードを， UNIX システムで使用されているメッセージに類似したメッセージに変換できます。errnoが EVMSERR に設定されている場合は，perrorはエラー・コードを変換することができず，次のメッセージを印刷し，その後，値に対応する OpenVMS エラー・メッセージを印刷します。

```
%s:nontranslatable vms error code: xxxxxx vms message:
```

このメッセージで，%s はperrorに指定する文字列です。 xxxxxx は OpenVMS 条件値です。

errnoが EVMSERR に設定されている場合は，<errno.h>ヘッダ・ファイルに宣言されているvaxc$errno変数で OpenVMS 条件値を使用できます。vaxc$errno変数は，errnoが EVMSERR に設定されている場合にだけ，有効な値を持つことが保証されます。errnoが EVMSERR 以外の値に設定されている場合は，vaxc$errnoの値は未定義です。

エラー・コードを変換するための別の方法については，『HP C ランタイム・ライブラリ・リファレンス・マニュアル (下巻)』「リファレンス・セクション」のstrerror関数を参照してください。

4.2 シグナル処理

シグナルとは，ユーザ・プロセスの通常の実行に対するソフトウェア割り込みです。シグナルは次に示すイベントなど，さまざまなイベントの結果として発生します。

- 端末での Ctrl/C の入力
- 特定のプログラミング・エラー
- gsignal関数またはraise関数の呼び出し
- ウェイクアップ動作

4.2.1 OpenVMS と UNIX の用語の比較

OpenVMS システムと UNIX システムはどちらも，シグナル処理機能を備えています。 2 つのシステムでこれらの機能は異なる動作をしますが，使用する用語は類似しています。HP C RTL では，プログラミングの際にどちらのシグナル処理機能も使用できます。シグナル処理ルーチンについて説明する前に，一部の用語について定義しておく必要があります。

UNIX では，ソフトウェア割り込みのことを**シグナル**と呼びます。 UNIX システムでシグナルを処理するために呼び出すルーチンを**シグナル・ハンドラ**と呼びます。

OpenVMS システムでは，ソフトウェア割り込みのことを**シグナル**，**条件**，**例外**などと呼びます。 OpenVMS システムでソフトウェア割り込みを処理するために呼び出すルーチンのことを**シグナル・ハンドラ**，**条件ハンドラ**，**例外ハンドラ**と呼びます。

混乱を避けるために，本書では，**シグナル**および**シグナル・ハンドラ**という用語は，UNIX の割り込みおよび割り込み処理ルーチンのことを参照し，**例外**および**例外ハンドラ**という用語は， OpenVMS の割り込みおよび割り込み処理ルーチンのことを参照することにします。

4.2.2 UNIX のシグナルとHP C RTL

シグナルは<signal.h>ヘッダ・ファイルに定義されているニーモニックによって表現されます。表 4–3 は，サポートされるシグナル・ニーモニックと， OpenVMS オペレーティング・システムで各シグナルを生成するイベントを示しています。

表 4–3 HP C RTL シグナル

名前	説明	シグナルを生成するイベント
SIGABRT[1]	強制終了	abort()
SIGALRM	アラーム・クロック	タイマ AST，alarmルーチン
SIGBUS	バス・エラー	アクセス違反または変更モード・ユーザ
SIGCHLD	子プロセスの停止	子プロセスの終了または停止
SIGEMT	EMT 命令	互換性モード・トラップまたはカスタマに予約されている op コード
SIGFPE	浮動小数点例外	浮動小数点オーバフロー/アンダフロー
SIGHUP	ハングアップ	データ・セット・ハングアップ
SIGILL[1]	不正な命令	不正な命令，予約オペランド，予約アドレス・モード
SIGINT[4]	割り込み	OpenVMS Ctrl/C 割り込み
SIGIOT[1]	IOT 命令	カスタマに予約されている op コード
SIGKILL[2], [3]	削除	外部シグナルのみ
SIGQUIT[5]	中止	実装されていない
SIGPIPE	破壊されたパイプ	リーダのないパイプへの書き込み
SIGSEGV	セグメント違反	長さ違反または変更モード・ユーザ
SIGSYS	システム呼び出しエラー	システム呼び出しに対する不正な引数
SIGTERM	ソフトウェアの終了	外部シグナルのみ
SIGTRAP[1]	トレース・トラップ	TBIT トレース・トラップまたはブレークポイント・フォルト命令
SIGUSR1	ユーザ定義シグナル	シグナルを送信するための明示的なプログラム呼び出し
SIGUSR2	ユーザ定義シグナル	シグナルを送信するための明示的なプログラム呼び出し
SIGWINCH[6]	ウィンドウ・サイズの変更	シグナルを送信するための明示的なプログラム呼び出し

[1]検出したときにリセットできない。

[2]検出または無視できない。

[3]ブロックできない。

[4]SIGINT を設定すると，Ctrl/Y 割り込みの種類に影響を与える可能性がある。たとえば，SIGINT をブロックまたは無視するという呼び出し元の要求に応答して，HP C RTL は Ctrl/Y 割り込みを無効にする。

[5]SIGQUIT に対する「実装されていない」という表現は，Ctrl/Y 割り込みも含めて，SIGQUIT シグナルを起動する外部イベントが存在しないため，SIGQUIT に対して設定されているシグナル・ハンドラが起動されることを意味する。このシグナルは，raiseなど，適切なHP C RTL 関数によってのみ生成することができる。

[6]OpenVMS Version 7.3 以降のバージョンでサポートされる。

デフォルト設定では，シグナル (SIGCHLD を除く) が発生すると，プロセスは終了します。しかし，次のいずれかの関数を使用してシグナルを無視するように設定することができます。

```
sigaction
signal
sigvec
```

```
ssignal
```

次のいずれかの関数を使用して，シグナルをブロックすることもできます。

```
sigblock
sigsetmask
sigprocmask
sigsuspend
sigpause
```

表 4–3 は，無視またはブロックできないシグナルを示しています。

次のいずれかの関数を使用して，シグナルを検出および処理するためにシグナル・ハンドラを設定することもできます。

```
sigaction
signal
sigvec
ssignal
```

表 4–3 に示した場合を除き，各シグナルはリセットできます。シグナル・ハンドラ関数がsignalまたはssignalを呼び出して，シグナルを検出するように再設定した場合，シグナルはリセットされます。例 4–1 は，シグナル・ハンドラの設定方法とシグナルのリセット方法を示しています。

シグナル・ハンドラの呼び出しインタフェースは次のとおりです。

```
void handler (int sigint);
```

ただし，sigintは，このハンドラを呼び出す原因になるシグナルのシグナル番号です。

sigvecによってインストールされたシグナル・ハンドラは，変更されるまでインストールされたままになります。

signalまたはsignalによってインストールされたシグナル・ハンドラは，シグナルが生成されるまでインストールされたままになります。

複数のシグナルに対して 1 つのシグナル・ハンドラをインストールすることができます。このことを制御するには， SA_RESETHAND フラグを使用してsigactionルーチンを呼び出します。

4.2.3 シグナル処理の概念

イベントによってシグナルが最初に発生した場合，プロセスに対してシグナルが**生成される** (またはプロセスにシグナルが送信される) と言うことができます。このようなイベントとしては，ハードウェア障害の検出，タイマの満了，端末での動作，killの

起動などがあります。場合によっては，同じイベントで複数のプロセスに対してシグナルが生成されることもあります。

各シグナルに対して各プロセスが実行する動作は，システムで定義されています。プロセスとシグナルに対して適切な動作が実行されたときに，シグナルはプロセスに**配布**されると言うこともできます。

シグナルが生成されてから配布されるまでの間，シグナルは**保留状態**であると言うことができます。通常，アプリケーションでこの間隔を検出することはできません。しかし，シグナルがプロセスに配布されないように**ブロック**することはできます。

- ブロックされているシグナルに対して割り当てられている動作がシグナルを無視するという動作以外の場合に，シグナルがプロセスに対して生成されると，シグナルのブロックが解除されるか，そのシグナルに割り当てられている動作がシグナルを無視するという動作に設定されるまで，シグナルは保留状態になります。
- ブロックされているシグナルに割り当てられている動作がシグナルを無視するという動作の場合に，シグナルがプロセスに対して生成されると，そのシグナルが生成と同時にただちに破棄されるか，保留状態で残されるかは未定義です。

各プロセスには**シグナル・マスク**があり，現在配布されないようにブロックされているシグナルのセットが定義されます。プロセスのシグナル・マスクはその親のシグナル・マスクから初期化されます。sigaction，sigprocmask，sigsuspend関数はシグナル・マスクの操作を制御します。

シグナルに対する応答としてどの動作を実行するかの判断は，シグナルの配布時に行われるため，シグナルが生成された後で変更することができます。この判断は，シグナルがもともと生成された手段とは無関係です。保留状態のシグナルが後で再び生成された場合，そのシグナルが 2 回以上配布されるかどうかは実装に応じて異なります。HP C RTL ではシグナルは 1 回だけ配布されます。同時に保留状態になっている複数のシグナルがプロセスに配布される順序は不定です。

4.2.4 シグナル・アクション

このセクションの説明は，sigaction，signal，sigvec，ssignal関数に適用されます。

シグナルに割り当てることができるアクションは次の 3 種類です。

SIG_DFL
SIG_IGN
関数を指すポインタ

最初は，mainルーチンのエントリの前で，すべてのシグナルが SIG_DFL または SIG_IGN に設定されます (exec関数を参照)。これらの値によって設定される動作は次のとおりです。

SIG_DFL — シグナル固有のデフォルト動作。

- 本書で定義しているシグナルのデフォルト動作は，<signal.h>に指定されています。
- デフォルト動作がプロセスの停止である場合，そのプロセスの実行が一時停止されます。プロセスが停止すると，親プロセスで SA_NOCLDSTOP フラグが設定されていない限り， SIGCHLD シグナルが親プロセスに対して生成されます。プロセスが停止されている間，そのプロセスに送信される追加シグナルは，プロセスが続行されるまで配布されません。ただし， SIGKILL は例外で，このシグナルは常にシグナルを受け取ったプロセスを終了します。孤立しているプロセス・グループのメンバであるプロセスは， SIGSTOP, SIGTTIN， SIGTTOU シグナルに対する応答として停止することができません。これらのシグナルのいずれかが配布されることによってこのようなプロセスが停止される場合，シグナルは破棄されます。
- 保留状態のシグナルに対して，シグナルの動作が SIG_DFL に設定されていて，そのデフォルト動作がシグナルの無視である場合 (たとえば SIGCHLD), シグナルがブロックされているかどうかとは無関係に，保留状態のシグナルは破棄されます。

SIG_IGN — シグナルを無視します。

- シグナルの配布はプロセスに影響を与えません。killまたはraise関数によって生成されたものではない SIGFPE， SIGILL， SIGSEGV シグナルを無視した後，プロセスの動作は未定義です。
- SIGKILL または SIGSTOP シグナルの動作を SIG_IGN に設定することはできません。
- 保留状態のシグナルに対してシグナルの動作を SIG_IGN に設定すると，保留状態のシグナルは，ブロックされているかどうかとは無関係に破棄されます。
- プロセスで SIGCHLD シグナルの動作が SIG_IGN に設定されると，動作は不定になります。

関数に対するポインタ — シグナルを検出します。

- シグナルの配布時に，シグナルを受け取るプロセスが指定されたアドレスでシグナル検出関数を実行します。シグナル検出関数から戻った後，シグナルを受信したプロセスは割り込まれた位置から実行を再開します。
- シグナル検出関数は次のように指定します。

  ```
  void func(int signo);
  ```

  ただし，funcは指定されたシグナル検出関数であり，signoは配布するシグナルのシグナル番号です。

- killまたはraiseによって生成されたものではない SIGFPE， SIGKILL，SIGSEGV シグナルに対するシグナル検出関数から正常に戻った後のプロセスの動作は未定義です。
- SIGKILL および SIGSTOP シグナルをプロセスで検出することはできません。
- 終了した子プロセスがあり，その子プロセスを待機していないときに，SIGCHLD シグナルに対してシグナル検出関数をプロセスで設定した場合，SIGCHLD シグナルがその子プロセスを示すように生成されるかどうかは不定です。

4.2.5 シグナル処理と OpenVMS の例外処理

ここでは，HP C RTL のシグナル処理が OpenVMS の例外処理でどのように実装され，どのように相互に影響するかについて説明します。このセクションの説明を読むと，HP C RTL のシグナル処理と競合しない OpenVMS 例外ハンドラを作成することができます。 OpenVMS の例外処理の詳細については，『OpenVMS Procedure Calling and Condition Handling Standard』を参照してください。

HP C RTL では， OpenVMS の例外によってシグナルを実装しています。gsignalまたはraiseを呼び出すと，シグナル番号が特定の OpenVMS 例外に変換され，LIB$SIGNAL の呼び出しで使用されます。ユーザ・エラーによって発生した OpenVMS 例外を検出し，それを対応する UNIX シグナルに変換するには，この機能が必要です。たとえば， NULL ポインタへの書き込みによって発生する ACCVIO は， SIGBUS または SIGSEGV シグナルに変換されます。

表 4–4，および 4–5 は，HP C RTL のシグナル名，対応する Open VMS Alpha，および Integrity の例外，シグナルを生成するイベント，gsignal関数およびraise関数で使用するための省略可能なシグナル・コードを示しています。

signalまたはsigvecによって設定したシグナル・ハンドラを呼び出すために，HP C RTL は，プログラムの main ルーチンに OpenVMS 例外ハンドラを配置することにより，シグナルに対応する OpenVMS 例外をインターセプトします。プログラムにmain関数がある場合は，この例外ハンドラが自動的に設定されます。main関数がない場合や， main 関数がHP C以外の言語で作成されている場合は，このハンドラを設定するためにVAXC$CRTL_INITルーチンを呼び出す必要があります。

HP C RTL では， OpenVMS 例外を使用してsetjmp関数とlongjmp関数を実装しています。longjmp関数を呼び出すと， C$_LONGJMP OpenVMS 例外が通知されます。 C$_LONGJMP 例外がユーザ例外ハンドラによって妨害されるのを防止するには，LIB$ESTABLISHを呼び出す代わりに，VAXC$ESTABLISHルーチンを使用してユーザ OpenVMS 例外ハンドラを設定します。 C$_LONGJMP ニーモニックは<errnodef.h>ヘッダ・ファイルに定義されています。

C プログラムで OpenVMS 例外ハンドラと UNIX シグナルを使用する場合は，表 4–4 *(Alpha only)*に示した OpenVMS 例外を受け付けて再通知するように， OpenVMS 例外ハンドラを準備する必要があり，さらにHP C RTL の将来のバージョンで導入される可能性のある C$機能例外や C$_LONGJMP 例外も受け付けるようにしなければなりません。これは， UNIX シグナルがコンテキストでグローバルであるのに対し，OpenVMS 例外がスタック・フレーム・ベースだからです。

したがって， OpenVMS 例外ハンドラは， main ルーチンのHP C RTL 例外ハンドラより前に， UNIX シグナルに対応する例外を常に受け取ります。 OpenVMS 例外を再通知することにより，HP C RTL 例外ハンドラは例外を受け取ることができます。 OpenVMS 例外をインターセプトすることも可能ですが，その場合は，対応する UNIX シグナルを禁止します。

表 4–4 HP C RTL のシグナルと対応する OpenVMS Alpha 例外*(Alpha only)*

名前	OpenVMS 例外	シグナルを生成するイベント	コード
SIGABRT	SS$_OPCCUS	abort関数	–
SIGALRM	SS$_ASTFLT	alarm関数	–
SIGBUS	SS$_ACCVIO	アクセス違反	–
SIGBUS	SS$_CMODUSER	変更モード・ユーザ	–
SIGCHLD	C$_SIGCHLD	子プロセスの停止	–
SIGEMT	SS$_COMPAT	互換性モード・トラップ	–
SIGFPE	SS$_DECDIV	10 進数除算トラップ	FPE_DECDIV_TRAP
SIGFPE	SS$_DECINV	10 進数不正オペランド・トラップ	FPE_DECINV_TRAP
SIGFPE	SS$_DECOVF	10 進オーバフロー・トラップ	FPE_DECOVF_TRAP
SIGFPE	SS$_HPARITH	0 による浮動小数点/ 10 進数除算	FPE_FLTDIV_TRAP
SIGFPE	SS$_HPARITH	浮動小数点オーバフロー・トラップ	FPE_FLTOVF_TRAP
SIGFPE	SS$_HPARITH	浮動小数点アンダフロー・トラップ	FPE_FLTUND_TRAP
SIGFPE	SS$_HPARITH	整数オーバフロー	FPE_INTOVF_TRAP
SIGFPE	SS$_HPARITH	不正なオペランド	FPE_INVOPR_TRAP
SIGFPE	SS$_HPARITH	不正確な結果	FPE_INXRES_TRAP
SIGFPE	SS$_INTDIV	0 による整数除算	FPE_INTDIV_TRAP
SIGFPE	SS$_SUBRNG	範囲外の添字	FPE_SUBRNG_TRAP
SIGFPE	SS$_SUBRNG1	範囲外の添字 1	FPE_SUBRNG1_TRAP
SIGFPE	SS$_SUBRNG2	範囲外の添字 2	FPE_SUBRNG2_TRAP
SIGFPE	SS$_SUBRNG3	範囲外の添字 3	FPE_SUBRNG3_TRAP
SIGFPE	SS$_SUBRNG4	範囲外の添字 4	FPE_SUBRNG4_TRAP
SIGFPE	SS$_SUBRNG5	範囲外の添字 5	FPE_SUBRNG5_TRAP
SIGFPE	SS$_SUBRNG6	範囲外の添字 6	FPE_SUBRNG6_TRAP
SIGFPE	SS$_SUBRNG7	範囲外の添字 7	FPE_SUBRNG7_TRAP
SIGHUP	SS$_HANGUP	データ・セット・ハングアップ	–

(次ページに続く)

表 4–4 (続き) HP C RTL のシグナルと対応する OpenVMS Alpha 例外*(Alpha only)*

名前	OpenVMS 例外	シグナルを生成するイベント	コード
SIGILL	SS$_OPCDEC	予約命令	ILL_PRIVIN_FAULT
SIGILL	SS$_ROPRAND	予約オペランド	ILL_RESOP_FAULT
SIGINT	SS$_CONTROLC	OpenVMS Ctrl/C 割り込み	–
SIGIOT	SS$_OPCCUS	カスタマ予約 op コード	–
SIGKILL	SS$_ABORT	外部シグナルのみ	–
SIGQUIT	SS$_CONTROLY	raise関数	–
SIGPIPE	SS$_NOMBX	メールボックスなし	–
SIGPIPE	C$_SIGPIPE	壊れたパイプ	–
SIGSEGV	SS$_ACCVIO	長さ違反	–
SIGSEGV	SS$_CMODSUPR	変更モード・スーパバイザ	–
SIGSYS	SS$_BADPARAM	システム呼び出しに対する不正な引数	–
SIGTERM	実装されていない	–	–
SIGTRAP	SS$_BREAK	ブレークポイント・フォルト・インストラクション	–
SIGUSR1	C$_SIGUSR1	raise関数	–
SIGUSR2	C$_SIGUSR2	raise関数	–
SIGWINCH[1]	C$_SIGWINCH[2]	raise関数	–

[1]OpenVMS Version 7.3 以降のバージョンでサポートされる。

[2]C$_SIGWINCH が定義されていない場合は SS$_BADWINCNT (OpenVMS Version 7.3 より前のバージョン)。

OpenVMS Alpha のシグナル処理に関する注意*(Alpha only)*

- OpenVMS VAX **システムに存在するシグナルはすべて** OpenVMS Alpha **システムにも存在しますが，対応する** OpenVMS **例外とコードは，多くの場合異なります。これは，** Alpha **プロセッサでは，** 2 **つの新しい** OpenVMS **例外が追加され，複数の例外が無効になったからです。**
- OpenVMS Alpha **システムのすべての浮動小数点例外は，** OpenVMS **例外**SS$_HPARITH (**高性能算術演算トラップ**) **によって通知されます。発生した特定のタイプのトラップは，例外サマリ・ロングワードを使用することによって**HP C RTL **で変換されます。このロングワードは，高性能算術演算トラップが通知されるときに設定されます。**

表 4–5 HP C RTL のシグナルと対応する OpenVMS Integrity 例外 *(Integrity only)*

名前	OpenVMS 例外	シグナルを生成するイベント	コード
SIGABRT	SS$_OPCCUS	abort関数	–
SIGALRM	SS$_ASTFLT	alarm関数	–
SIGBUS	SS$_ACCVIO	アクセス違反	–
SIGBUS	SS$_CMODUSER	変更モード・ユーザ	–
SIGCHLD	C$_SIGCHLD	子プロセスの停止	–
SIGEMT	SS$_COMPAT	互換性モード・トラップ	–
SIGFPE	SS$_DECOVF	10 進オーバフロー・トラップ	FPE_DECOVF_TRAP
SIGFPE	SS$_DECDIV	10 進数除算トラップ	FPE_DECDIV_TRAP
SIGFPE	SS$_DECINV	10 進数不正オペランド・トラップ	FPE_DECINV_TRAP
SIGFPE	SS$_ FLTDENORMAL	デノーマル・オペランド・フォルト	FPE_FLTDENORMAL_FAULT
SIGFPE	SS$_FLTDIV	0 による浮動小数点/10 進数除算	FPE_FLTDIV_TRAP
SIGFPE	SS$_FLTDIV_F	0 による浮動小数点除算フォルト	FPE_FLTDIV_FAULT
SIGFPE	SS$_FLTINE	不正確操作トラップ	FPE_FLTINE_TRAP
SIGFPE	SS$_FLTINV	不正操作トラップ	FPE_FLTINV_TRAP
SIGFPE	SS$_FLTINV_F	不正操作フォルト	FPE_FLTINV_FAULT
SIGFPE	SS$_FLTOVF	浮動小数点オーバフロー・トラップ	FPE_FLTOVF_TRAP
SIGFPE	SS$_FLTUND	浮動小数点アンダフロー・トラップ	FPE_FLTUND_TRAP
SIGFPE	SS$_INTDIV	0 による整数除算	FPE_INTDIV_TRAP
SIGFPE	SS$_INTOVF	整数オーバフロー	FPE_INTOVF_TRAP
SIGFPE	SS$_SUBRNG	添字の範囲	FPE_SUBRNG_TRAP
SIGHUP	SS$_HANGUP	データ・セット・ハングアップ	–
SIGILL	SS$_OPCDEC	予約命令	ILL_PRIVIN_FAULT
SIGILL	SS$_ROPRAND	予約オペランド	ILL_RESOP_FAULT
SIGINT	SS$_CONTROLC	OpenVMS Ctrl/C 割り込み	–
SIGIOT	SS$_OPCCUS	カスタマ予約 op コード	–
SIGKILL	SS$_ABORT	外部シグナルのみ	–
SIGQUIT	SS$_CONTROLY	raise関数	–
SIGPIPE	SS$_NOMBX	メールボックスなし	–
SIGPIPE	C$_SIGPIPE	壊れたパイプ	–
SIGSEGV	SS$_ACCVIO	長さ違反	–
SIGSEGV	SS$_CMODSUPR	変更モード・スーパバイザ	–

(次ページに続く)

表 4–5 (続き) HP C RTL のシグナルと対応する OpenVMS Integrity 例外 *(Integrity only)*

名前	OpenVMS 例外	シグナルを生成するイベント	コード
SIGSYS	SS$_BADPARAM	システム呼び出しに対する不正な引数	–
SIGTERM	実装されていない	–	–
SIGTRAP	SS$_TBIT	TBIT トレース・トラップ	–
SIGTRAP	SS$_BREAK	ブレークポイント・フォルト命令	–
SIGUSR1	C$_SIGUSR1	raise関数	–
SIGUSR2	C$_SIGUSR2	raise関数	–
SIGWINCH	C$_SIGWINCH	raise関数	–

4.3 プログラムの例

例 4–1 は，signal，alarm，pause関数の動作方法を示しています。また，プログラムの終了を防止するために，シグナルを検出するようにシグナル・ハンドラを設定する方法も示しています。

例 4–1 プログラムの一時停止と再開

```
/*    CHAP_4_SUSPEND_RESUME.C                                    */

/* This program shows how to alternately suspend and resume a   */
/* program using the signal, alarm, and pause functions.        */

#define SECONDS 5

#include <stdio.h>
#include <signal.h>

int number_of_alarms = 5;       /*  Set alarm counter.           */

void alarm_action(int);

main()
{
    signal(SIGALRM, alarm_action); /*  Establish a signal handler. */
                                   /*  to catch the SIGALRM signal.*/

    alarm(SECONDS);     /* Set alarm clock for 5 seconds. */

    pause();    /*  Suspend the process until     *
                 *  the signal is received.      */
}

void alarm_action(int x)
{
    printf("\t<%d\007>", number_of_alarms); /*  Print the value of  */
                                           /*  the alarm counter.  */

    signal(SIGALRM, alarm_action);      /*  Reset the signal.       */
```

(次ページに続く)

例 4–1 (続き) プログラムの一時停止と再開

```
    alarm(SECONDS);     /*  Set the alarm clock.      */
    if (--number_of_alarms)     /*  Decrement alarm counter.  */
        pause();
}
```

例 4–1 から次の出力が生成されます。

```
$ RUN  EXAMPLE
       <5>   <4>   <3>   <2>   <1>
```

5
サブプロセス関数

HP C Run-Time Library (RTL) では，HP Cプログラムからサブプロセスを生成するための関数が提供されます。サブプロセスを生成するプロセスを**親**と呼び，生成されるサブプロセスを**子**と呼びます。

親プロセスの内部で子プロセスを生成するには，exec関数 (execl，execle，execv，execve，execlp，execvp) およびvfork関数を使用します。その他にも，親プロセスと子プロセスがプロセス間でデータの読み書きを実行するための関数 (pipe) や 2 つのプロセスの同期をとるための関数 (wait) などもあります。この章では，これらの関数の実装の方法と使用方法について説明します。

親プロセスは子プロセスの内部で，同期的または非同期的にHP Cプログラムを実行できます。同時に実行できる子プロセスの数は，システムの各ユーザに対して設定されている/PRCLM ユーザ登録クォータによって決定されます。サブプロセスの使用に影響を与える可能性のあるその他のクォータとしては，/ENQLM (キュー・エントリ・リミット)，/ASTLM (AST 待ちリミット)，/FILLM (オープン・ファイル・リミット) があります。

この章では，サブプロセス関数について説明します。表 5–1 は，HP C RTL で提供されるすべてのサブプロセス関数を示しています。各関数の詳細については，『HP C ランタイム・ライブラリ・リファレンス・マニュアル (下巻)』「リファレンス・セクション」を参照してください。

表 5–1　サブプロセス関数

関数	説明
子プロセスの生成	
system	コマンド・プロセッサによって実行される文字列をホスト環境に渡す。
vfork	独立した子プロセスを生成する。
exec 関数	
execl，execle，execlp execv，execve，execvp	子プロセスの内部で起動されるイメージの名前を渡す。

(次ページに続く)

表 5–1 (続き) サブプロセス関数

関数	説明
プロセスの同期化	
wait, wait3, wait4, waitpid	値が子から返されるまで，親プロセスを一時停止する。
プロセス間通信	
pipe	親プロセスと子プロセスの間の通信を可能にする。

5.1 HP C での子プロセスの生成

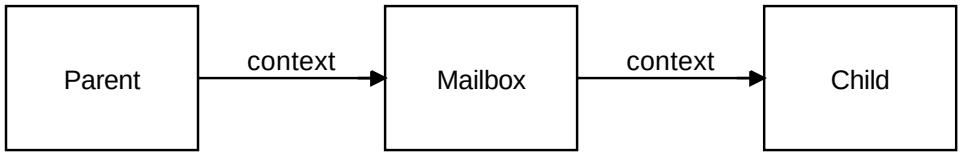
子プロセスは， OpenVMS LIB$SPAWN RTL ルーチンとHP C関数によって生成されます (LIB$SPAWN の詳細については，『VMS Run-Time Library Routines Volume』を参照してください)。 LIB$SPAWN を使用すると，複数レベルの子プロセスを生成できます。親から生成された子がさらに子を生成することができます。生成できるレベルは，この章の冒頭で説明したユーザ登録クォータで認められている上限値までです。

子プロセスは他のHP Cプログラムだけを実行できます。他のネイティブ・モードのOpenVMS 言語は，プロセス間で通信するためにHP Cの機能を共用しません。プロセス間で通信する場合，他の言語は同じ機能を使用しません。親プロセスは， DCL など，弊社がサポートするコマンド言語インタプリタ (CLI) のもとで実行しなければなりません。親を独立プロセスとして実行したり，ユーザ指定 CLI の制御のもとで実行することはできません。

DECC$DETACHED_CHILD_PROCESS 機能論理名を有効にすると，子プロセスが，サブプロセスではなく，デタッチされたプロセスとして生成されます。この機能のサポートには，制限があります。場合によっては，コンソールが親プロセスと，デタッチされたプロセスで共用できず，execが失敗することがあります。

親プロセスと子プロセスは，図 5–1 に示すようにメールボックスを通じて通信します。このメールボックスは，子が実行されるコンテキストを転送します。この**コンテキスト・メールボックス**は，親がオープンしたすべてのファイルの名前やファイル記述子，それらのファイル内での現在の位置など，親から継承する情報を子に渡します。子イメージが終了すると，メールボックスは親によって削除されます。

図 5–1 親プロセスと子プロセスの間の通信リンク



ZK−4002−GE

注意

vfork関数とexec関数によって作成されるメールボックスは一時的なものです。このメールボックスの論理名は VAXC$EXECMBX **であり，**HP C RTL **で使用するために予約されています。**

メールボックスは 512 バイトの最大メッセージ・サイズおよび 512 バイトのバッファ・クォータで作成されます。これらの RTL 関数を使用してメールボックスを作成するには， TMPMBX 特権が必要です。 TMPMBX は DCL コマンド PRINT および SUBMIT で必要とされる特権であるため，システムの大部分のユーザはこの特権を保有しています。保有しているシステム特権を確認するには， SHOW PROCESS /PRIVILEGES コマンドを入力します。

これらのメールボックスの属性を変更することはできません。メールボックスの詳細については，『VMS I/O User's Reference Volume』を参照してください。

5.2 exec 関数

子プロセスでHP Cイメージを実行するために呼び出すことのできるexec関数は 6 つあります。これらの関数では，戻りアドレスを設定するために vforkがあらかじめ呼び出されていることが必要です。exec関数は，親プロセスでvforkが呼び出されていない場合，その関数を呼び出します。

vforkが親で呼び出されると，exec関数は親プロセスに戻ります。vforkがexec関数によって呼び出されると，exec関数はその関数自体に戻り，子プロセスが終了するのを待ち，その後で親プロセスを終了します。exec関数は，親がvforkを呼び出して戻りアドレスを保存しない限り，親プロセスに戻りません。

OpenVMS Version 7.2 で，exec関数は実行可能イメージまたは DCL コマンド・プロシージャを起動するように拡張されました。ファイル拡張がfile_name引数に指定されていない場合は，この関数はまずファイル拡張が.EXE であるファイルを検索し，次にファイル拡張が.COM であるファイルを検索します。同じ名前の実行可能イメージとコマンド・プロシージャの両方が存在する場合は，コマンド・プロシージャを強制的に起動するために，.COM ファイル拡張を指定する必要があります。

DCL コマンド・プロシージャの場合は，exec関数は，P1，P2，... パラメータなど，exec呼び出しに指定した最初の 8 つのarg0，arg1，... 引数をコマンド・プロシージャに渡します。その場合，大文字と小文字の区別は保持されます。

UNIX ベースのシステムと異なり，HP C RTL のexec関数は，指定された実行可能イメージまたはコマンド・プロシージャが存在するかどうか，またこれらを起動および実行できるがどうかを常に判断できるわけではありません。したがって，exec関数は，子プロセスが指定されたファイルを実行できない場合でも，正常終了したように見えることがあります。

親プロセスへ返される子プロセスの状態は，エラーが発生したことを示します。このエラー・コードは，wait関数ファミリのいずれかの関数を使用することにより検索できます。

注意

OpenVMS システムのHP C RTL のvfork関数とexec関数は，UNIX システムの場合と異なる方法で動作します。

- **UNIX システムでは，vforkは子プロセスを生成し，親プロセスを一時停止し，親が停止した場所から子プロセスの実行を開始します。**
- **OpenVMS システムでは，vforkはexec関数で後で使用されるコンテキストを設定しますが，指定されたプログラムを実行するプロセスを開始するのは，vforkではなく，exec関数です。**

プログラマの場合，次の重要な相違点に注意してください。

- **OpenVMS システムでは，vfork関数の呼び出しとexec関数の呼び出しの間のコードは親プロセスで実行されます。**

  UNIX システムでは，このコードは子プロセスで実行されます。
- **OpenVMS システムでは，exec関数が呼び出されたポイントで，子プロセスはオープンされているファイルの記述子などを継承します。**

  UNIX システムでは，この継承は，vforkが呼び出されたポイントで行われます。

5.2.1 exec の処理

exec関数では， LIB$SPAWN ルーチンを使用してサブプロセスを生成し，サブプロセス内で子イメージを起動します。この子プロセスは，定義されているすべての論理名やコマンド・ライン・インタプリタ・シンボルなど，親の環境を継承します。

デフォルトでは，子プロセスは親プロセスのデフォルト (作業) ディレクトリも継承します。ただし，decc$set_child_default_dir関数を使用して，子プロセスの実行開始時のデフォルト・ディレクトリを設定できます。decc$set_child_default_dir関数の

詳細については，『HP C ランタイム・ライブラリ・リファレンス・マニュアル (下巻)』の「リファレンス・セクション」を参照してください。

exec関数は論理名 VAXC$EXECMBX を使用して，親と子の間の通信を行います。この論理名は親イメージのコンテキストの内部に存在しなければなりません。

exec関数は次の情報を子に渡します。

- 親のumaskの値。この値は，新規にファイルを作成するときに，いずれかのアクセスを禁止するかどうかを指定します。umask関数の詳細については，『HP C ランタイム・ライブラリ・リファレンス・マニュアル (下巻)』「リファレンス・セクション」を参照してください。
- 各ファイル記述子に割り当てられているファイル名文字列と，各ファイル内での現在の位置。子プロセスはファイルをオープンし，lseekを呼び出して，親と同じ位置にファイルの現在の位置を設定します。ファイルがレコード・ファイルの場合は，レコード内での親の位置とは無関係に，子プロセスの現在の位置はレコード境界に設定されます。ファイル記述子の詳細については，第 2 章を参照してください。lseek関数の詳細については，『HP C ランタイム・ライブラリ・リファレンス・マニュアル (下巻)』「リファレンス・セクション」を参照してください。

  オープンされているファイルのすべての記述子，ヌル記述子，重複記述子も含めて，親が認識するすべての記述子に関して，この情報が子に送信されます。

  ファイル・ポインタは子に転送されません。親でファイル・ポインタによってオープンされたファイルの場合，対応するファイル記述子だけが子に渡されます。子プロセスが各ファイルのポインタにアクセスする場合は，fdopen関数を呼び出してファイル・ポインタをファイル記述子に関連付ける必要があります。fdopen関数の詳細については，『HP C ランタイム・ライブラリ・リファレンス・マニュアル (下巻)』「リファレンス・セクション」を参照してください。

  DECC$EXEC_FILEATTR_INHERITANCE 機能論理名を使用すると，子プロセスがファイル位置を継承するかどうかと，継承する場合はどのアクセス・モードについて継承するかを制御できます。 DECC$EXEC_FILEATTR_INHERITANCE の詳細は，第 1.5 節を参照してください。
- シグナル・データベース。 SIG_IGN (無視) 動作だけが継承されます。親のシグナル処理ルーチンに子からアクセスすることはできないため，ルーチンとして指定された動作は SIG_DFL (デフォルト) に変更されます。
- 環境および引数ベクタ。

すべての情報が子に転送されると，execの処理は終了します。親プロセス内の制御は，vforkによって保存されたアドレスに返され，子のプロセス ID は親に返されます。

シグナル動作および SIGCHLD シグナルの詳細については，第 4.2.4 項を参照してください。

5.2.2 exec **のエラー条件**

LIB$SPAWN がサブプロセスを生成できない場合は，exec関数は異常終了します。エラーの原因となる可能性のある条件としては，サブプロセス・クォータの超過や，親と子の間でコンテキスト・メールボックスによる通信が不能であることの検出などがあります。一部のクォータは，超過しても LIB$SPAWN がエラーになることはありませんが， LIB$SPAWN が待ち状態になり，その結果，親プロセスがハングする可能性があります。このようなクォータの例としては，オープン・ファイル・リミット・クォータがあります。

オープン・ファイル・リミット・クォータは少なくとも 20 ファイルに設定する必要があり，平均値はプログラムで同時に実行するプロセスの数の 3 倍です。一度に複数のオープン・パイプを使用する場合や，一度に複数のファイルに対して I/O を実行する場合は，このクォータをさらに大きくする必要があります。このクォータを増大する必要があるかどうかについては，システム管理者にお問い合わせください。

exec関数が異常終了した場合，–1 という値が返されます。このような障害が発生した後，親はexit関数または_exit関数を呼び出すことが期待されます。どちらの関数も親のvfork呼び出しに戻り，この関数呼び出しは子のプロセス ID を返します。この場合，exec関数から返される子プロセス ID は 0 より小さくなります。exit関数の詳細については，『HP C ランタイム・ライブラリ・リファレンス・マニュアル (下巻)』「リファレンス・セクション」を参照してください。

5.3 プロセスの同期化

親プロセスが終了すると，子プロセスも終了します。したがって，親プロセスは終了する前に，子プロセスの状態を確認する必要があります。この処理は，HP C RTL 関数waitを使用して行います。

5.4 プロセス間通信

親プロセスと子プロセスが通信するチャネルは**パイプ**と呼ばれます。パイプを作成するには，pipe関数を使用します。

5.5 プログラムの例

例 5–1 は，子プロセスでイメージを実行するための基本手順を示しています。例 5–1 の子プロセスはメッセージを 10 回プリントします。

例 5–1 子プロセスの生成

(次ページに続く)

例 5–1 (続き) 子プロセスの生成

```
/*      chap_5_exec_image.c     */

/* This example creates the child process.  The only    */
/* functionality given to the child is the ability to   */
/* print a message 10 times.                            */

#include <climsgdef.h>  /* CLI status values  */
#include <stdio.h>
#include <perror.h>
#include <processes.h>
#include <stdlib.h>

static const char *child_name = "chap_5_exec_image_child.exe" ;

main()
{
   int status,
       cstatus;

   /* NOTE:                                            */
   /*    Any local automatic variables, even those     */
   /*    having the volatile attribute, may have       */
   /*    indeterminant values if they are modified     */
   /*    between the vfork() call and the matching     */
   /*    exec() call.                                  */

1  if ((status = vfork()) != 0) {
       /* This is either an error or                   */
       /* the "second" vfork return, taking us "back"  */
       /* to parent mode.                              */
3      if (status < 0)
            printf("Parent - Child process failed\n");
       else {
            printf("Parent - Waiting for Child\n");
4           if ((status = wait(&cstatus)) == -1)
               perror("Parent - Wait failed");
5           else if (cstatus == CLI$_IMAGEFNF)
               printf("Parent - Child does not exist\n");
            else
               printf("Parent - Child final status: %d\n", cstatus);
        }
    }
2   else {  /* The FIRST Vfork return is zero, do the exec */
          printf("Parent - Starting Child\n");
          if ((status = execl(child_name, 0)) == -1) {
              perror("Parent - Execl failed");
              exit(EXIT_FAILURE);
         }
    }
}

-------------------------------------------------------
```

(次ページに続く)

例 5–1 (続き) 子プロセスの生成

```
/*      CHAP_5_EXEC_IMAGE_CHILD.C                    */

/* This is the child program that writes a message   */
/* through the parent to "stdout"                    */

#include <stdio.h>

main()
{
    int i;

    for (i = 0; i < 10; i++)
        printf("Child - executing\n");
    return (255) ;    /* Set an unusual success stat */
}
```

例 5–1 の補足説明:

1. vfork関数は，exec呼び出しの戻りアドレスを設定するために呼び出されます。

   vfork関数は通常，if文の式で使用されます。この構造では，1 つの戻り値が 0 で，もう 1 つが 0 以外であるため，vforkの 2 つの戻りアドレスを利用できます。

2. vforkを最初に呼び出すと，戻り値 0 が返され，親はvfork呼び出しに関連付けられたelse句を実行し，その結果，execlが呼び出されます。

3. exec関数が異常終了すると，負の子プロセス ID が返されます。これらの条件に関して戻り値が確認されます。

4. wait関数は，親プロセスと子プロセスの同期をとるために使用されます。

5. exec関数は，子プロセスで起動されるイメージが存在しない場合でも，この時点まで正常終了を示す可能性があるため，親が子プロセスの戻り状態を調べて，定義済み状態 CLI$_IMAGEFNF (ファイルが見つからない) が返されていないかどうか確認します。

例 5–2 では，親は子プロセスに引数を渡します。

例 5–2 子プロセスへの引数の引き渡し

```
/*       CHAP_5_CHILDARG.C                                    */

/* In this example, the arguments are placed in an array, gargv,     */
/* but they can be passed to the child explicitly as a zero-         */
/* terminated series of character strings. The child program in this */
/* example writes the arguments that have been passed it to stdout.  */

#include <climsgdef.h>
#include <stdio.h>
#include <stdlib.h>
#include <perror.h>
#include <processes.h>
```

(次ページに続く)

例 5–2 (続き) 子プロセスへの引数の引き渡し

```
const char *child_name = "chap_5_childarg_child.exe" ;

main()
{
    int status,
        cstatus;
    char *gargv[] =
    {"Child", "ARGC1", "ARGC2", "Parent", 0};

    if ((status = vfork()) != 0) {
        if (status < -1)
            printf("Parent - Child process failed\n");
        else {
            printf("Parent - waiting for Child\n");
            if ((status = wait(&cstatus)) == -1)
                perror("Parent - Wait failed");
            else if (cstatus == CLI$_IMAGEFNF)
                printf("Parent - Child does not exist\n");
            else
                printf("Parent - Child final status: %x\n",
                       cstatus);
        }
    }
    else {
        printf("Parent - Starting Child\n");
        if ((status = execv(child_name, gargv)) == -1) {
            perror("Parent - Exec failed");
            exit(EXIT_FAILURE);
        }
    }
}

------------------------------------------------------
/*       CHAP_5_CHILDARG_CHILD.C                     */

/* This is a child program that echos its arguments   */

#include <stdio.h>

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

    printf("Program name: %s\n", argv[0]);

    for (i = 1; i < argc; i++)
        printf("Argument %d: %s\n", i, argv[i]);
    return(255) ;
}
```

例 5–3 は，wait関数を使用して，同時に実行される複数の子の最終状態を確認する方法を示しています。

例 5–3 子プロセスの状態の確認

```
/*       CHAP_5_CHECK_STAT.C                                    */

/* In this example 5 child processes are started.  The wait()   */
/* function is placed in a separate for loop so that it is      */
/* called once for each child. If wait() were called within     */
/* the first for loop, the parent would wait for one child to   */
/* terminate before executing the next child. If there were     */
/* only one wait request, any child still running when the      */
/* parent exits would terminate prematurely.                    */

#include <climsgdef.h>
#include <stdio.h>
#include <stdlib.h>
#include <perror.h>
#include <processes.h>

const char *child_name = "chap_5_check_stat_child.exe" ;

main()
{
    int status,
        cstatus,
        i;

    for (i = 0; i < 5; i++) {
        if ((status = vfork()) == 0) {
            printf("Parent - Starting Child %d\n", i);
            if ((status = execl(child_name, 0)) == -1) {
                perror("Parent - Exec failed");
                exit(EXIT_FAILURE);
            }
        }
        else if (status < -1)
            printf("Parent - Child process failed\n");
    }

    printf("Parent - Waiting for children\n");

    for (i = 0; i < 5; i++) {
        if ((status = wait(&cstatus)) == -1)
            perror("Parent - Wait failed");
        else if (cstatus == CLI$_IMAGEFNF)
            printf("Parent - Child does not exist\n");
        else
            printf("Parent - Child %X final status: %d\n",
                   status, cstatus);
    }
}
```

例 5–4 は，pipe関数とdup2関数を使用して，特定のファイル記述子を通じて親プロセスと子プロセスの間で通信する方法を示しています。#defineプリプロセッサ・ディレクティブは，プリプロセッサ定数inpipeとoutpipeをファイル記述子 11 と 12 の名前として定義します。

例 5–4 パイプによる通信

```
/*       CHAP_5_PIPE.C                                          */

/* In this example, the parent writes a string to the pipe for  */
/* the child to read.  The child then writes the string back    */
/* to the pipe for the parent to read.  The wait function is    */
/* called before the parent reads the string that the child has */
/* passed back through the pipe.  Otherwise, the reads and      */
/* writes will not be synchronized.                             */

#include <perror.h>
#include <climsgdef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <processes.h>
#include <unixio.h>

#define inpipe 11
#define outpipe 12

const char *child_name = "chap_5_pipe_child.exe" ;

main()
{
    int pipes[2];
    int mode,
        status,
        cstatus,
        len;
    char *outbuf,
        *inbuf;

    if ((outbuf = malloc(512)) == 0) {
        printf("Parent - Outbuf allocation failed\n");
        exit(EXIT_FAILURE);
    }

    if ((inbuf = malloc(512)) == 0) {
        printf("Parent - Inbuf allocation failed\n");
        exit(EXIT_FAILURE);
    }
    if (pipe(pipes) == -1) {
        printf("Parent - Pipe allocation failed\n");
        exit(EXIT_FAILURE);
    }

    dup2(pipes[0], inpipe);
    dup2(pipes[1], outpipe);
    strcpy(outbuf, "This is a test of two-way pipes.\n");

    status = vfork();
```

(次ページに続く)

例 5–4 (続き) パイプによる通信

```
    switch (status) {
    case 0:
        printf("Parent - Starting child\n");
        if ((status = execl(child_name, 0)) == -1) {
            printf("Parent - Exec failed");
            exit(EXIT_FAILURE);
        }
        break;

    case -1:
        printf("Parent - Child process failed\n");
        break;

    default:
        printf("Parent - Writing to child\n");

        if (write(outpipe, outbuf, strlen(outbuf) + 1) == -1) {
            perror("Parent - Write failed");
            exit(EXIT_FAILURE);
        }
        else {
            if ((status = wait(&cstatus)) == -1)
                perror("Parent - Wait failed");

            if (cstatus == CLI$_IMAGEFNF)
                printf("Parent - Child does not exist\n");
            else {
                printf("Parent - Reading from child\n");
                if ((len = read(inpipe, inbuf, 512)) <= 0) {
                    perror("Parent - Read failed");
                    exit(EXIT_FAILURE);
                }
                else {
                    printf("Parent: %s\n", inbuf);
                    printf("Parent - Child final status: %d\n",
                            cstatus);
                }
            }
        }
        break;
    }
}

------------------------------------------------------------------
/*       CHAP_5_PIPE_CHILD.C                                      */

/* This is a child program which reads from a pipe and writes   */
/* the received message back to its parent.                     */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
```

(次ページに続く)

例 5–4 (続き) パイプによる通信

```
#define inpipe 11
#define outpipe 12

main()
{
    char *buffer;
    int len;

    if ((buffer = malloc(512)) == 0) {
        perror("Child - Buffer allocation failed\n");
        exit(EXIT_FAILURE);
    }

    printf("Child - Reading from parent\n");
    if ((len = read(inpipe, buffer, 512)) <= 0) {
        perror("Child - Read failed");
        exit(EXIT_FAILURE);
    }
    else {
        printf("Child: %s\n", buffer);
        printf("Child - Writing to parent\n");
        if (write(outpipe, buffer, strlen(buffer) + 1) == -1) {
            perror("Child - Write failed");
            exit(EXIT_FAILURE);
        }
    }
    exit(EXIT_SUCCESS);
}
```

6

Curses 画面管理関数とマクロ

この章では，HP C for OpenVMS システムで提供される画面管理ルーチンについて説明します。

OpenVMS Curses 画面管理パッケージは，すべての OpenVMS システムでサポートされます。

OpenVMS Alpha システムでは， 2 つの画面管理パッケージがサポートされます。それは OpenVMS Curses と， UNIX との互換性がより高く， BSD (Berkeley Standard Distribution) Curses ソフトウェア[1]をベースにしたパッケージです。詳細については，第 6.1 節を参照してください。

さらに，HP C RTL では， 4.4BSD Berkeley Software Distribution をベースにした Curses パッケージも提供されるようになりました。 4.4BSD Curses パッケージに関する情報は， Kenneth C.R.C. Arnold 著，『Screen Updating and Cursor Movement Optimization: A Library Package』に記載されています。

OpenVMS と BSD ベースの Curses パッケージの関数およびマクロはほぼ同じです。両者の相違点の大部分については，この章で説明しています。それ以外の場合は，この章では 2 つの Curses パッケージを区別せず，単に「Curses」または「Curses 関数とマクロ」と呼んでいます。

6.1 BSD ベースの Curses パッケージの使用 *(Alpha only)*

BSD ベースの Curses の実装を使用するのに必要な<curses.h>ヘッダ・ファイルは，OpenVMS Alpha システムではHP Cコンパイラで提供されます。

OpenVMS Curses 関数はデフォルトの Curses パッケージとして提供されるため，既存のプログラムは BSD ベースの Curses 関数の影響を受けません (この点は， BSD ベースの Curses がデフォルトであったHP Cの以前のバージョンから変更された点です)。

4.4BSD Curses の実装を入手するには，次の修飾子を使用して<curses.h>を取り込むモジュールをコンパイルする必要があります。

```
/DEFINE=_BSD44_CURSES
```

[1] Copyright (c) 1981 Regents of the University of California.
All rights reserved.

BSD ベースの Curses 関数では， Curses 関数によって割り当てられたペーストボードとキーボードを使用する OpenVMS SMG$ルーチンを呼び出すのに必要なサポートは提供されません。したがって， SMG$エントリ・ポイントの呼び出しおよび Curses 関数に依存する Curses プログラムは，今後も OpenVMS の Curses を使用する必要があります。

BSD ベースの Curses は，以前の実装との間で相互運用性がありません。新しい関数と古い関数の呼び出しが混在している場合，画面に不正な出力が表示され， SMG$ルーチンから例外が発生する可能性があります。

6.2 Curses の概要

Curses，つまりHP C Screen Management Package は，端末画面の定義されたセクションの作成と変更，およびカーソルの最適な移動を行うHP C RTL 関数およびマクロで構成されています。画面管理パッケージを使用すると，ユーザにとって親しみやすく，魅力のあるユーザ・インタフェースを開発できます。 Curses は端末に依存しておらず，端末画面の書式設定を簡単に行うことができ，効率のよいカーソルの移動を実現できます。

ほとんどの Curses 関数とマクロは 2 つ 1 組で示されます。最初のルーチンがマクロで， 2 番目が“ウィンドウ”を表す接頭語“w”から始まる関数です。これらの接頭語は角括弧 ([ ]) で囲んで示されます。たとえば，[w]addstrは，addstrマクロとwaddstr関数を示します。マクロのウィンドウは，デフォルトでstdscrに設定され，関数は，引数として指定されたウィンドウを受け付けます。

Curses 関数およびマクロにアクセスするには，<curses.h>ヘッダ・ファイルを取り込みます。

端末に依存しない Screen Management Software は OpenVMS RTL の一部であり，Curses を実装するために使用されます。移植性を確保するために，ほとんどの関数とマクロは他の C の実装と同様の方法で動作するように設計されています。しかし，Curses ルーチンはOpenVMS システムおよびその Screen Management Software に依存しているため，一部の関数とマクロの動作は他の実装の関数およびマクロと少し異なる可能性があります。

他のシステムで提供される関数とマクロの中には，HP C RTL Curses パッケージで提供されないものがあります。

[w]clrattr，[w]insstr，mv[w]insstr，[w]setattrなどの一部の関数は，HP C for OpenVMS Systems 固有であり，移植できません。

表 6–1 は，HP C RTL で提供されるすべての Curses 関数とマクロを示しています。各関数とマクロの詳細については，『HP C ランタイム・ライブラリ・リファレンス・マニュアル (下巻)』「リファレンス・セクション」を参照してください。

表 6–1 Curses 関数とマクロ

関数またはマクロ	説明
[w]addch	ウィンドウの現在のカーソルの位置に文字を追加する。
[w]addstr	ウィンドウの現在のカーソルの位置に文字列を追加する。
box	ウィンドウの周囲に四角形を描く。
[w]clear	指定されたウィンドウの内容を消去し，カーソルの位置を座標 (0,0) にリセットする。
clearok	ウィンドウのクリア・フラグを設定する。
[w]clrattr	ウィンドウの内部でビデオ表示属性を無効にする。
[w]clrtobot	カーソルの現在の位置からウィンドウの最下部まで，ウィンドウの内容を消去する。
[w]clrtoeol	指定されたウィンドウの現在のカーソルの位置から行末まで，ウィンドウの内容を消去する。
[no]crmode	端末を cbreak モードに設定する。または設定を解除する。
[w]delch	指定されたウィンドウで，現在のカーソルの位置から文字を削除する。
[w]deleteln	現在のカーソルの位置から行を削除する。
delwin	指定されたウィンドウをメモリから削除する。
[no]echo	文字が端末画面に表示されるように，または表示されないように，端末を設定する。
endwin	端末画面をクリアし， Curses 構造体に割り当てられている仮想メモリを解放する。
[w]erase	空白をペイントすることにより，ウィンドウを消去する。
[w]getch	端末画面から 1 文字を取得し，指定されたウィンドウに表示する。
[w]getstr	端末画面から文字列を取得し，文字変数に格納し，指定されたウィンドウに表示する。
getyx	ウィンドウで現在のカーソルの位置の座標 (y,x) を変数 y および変数 x に代入する。
[w]inch	ウィンドウを変更せずに，指定されたウィンドウの現在のカーソルの位置にある文字を返す。
initscr	端末タイプ・データとすべての画面関数を初期化する。
[w]insch	指定されたウィンドウの現在のカーソルの位置に文字を挿入する。
[w]insertln	現在カーソルが設定されている行の上に 1 行を挿入する。
[w]insstr	指定されたウィンドウの現在のカーソルの位置に文字列を挿入する。
leaveok	ウィンドウを更新した後，現在の座標にカーソルをそのまま置く。
longname	文字列を十分格納できるだけの大きさの文字名に完全な端末の名前を代入する。
[w]move	指定されたウィンドウの現在のカーソルの位置を変更する。
mv[w]addch	カーソルを移動し，指定されたウィンドウに文字を追加する。
mv[w]addstr	カーソルを移動し，指定されたウィンドウに文字列を追加する。
mvcur	端末のカーソルを移動する。
mv[w]delch	カーソルを移動し，指定されたウィンドウで文字を削除する。

(次ページに続く)

表 6–1 (続き) Curses 関数とマクロ

関数またはマクロ	説明
mv[w]getch	カーソルを移動し，端末画面から文字を取得し，指定されたウィンドウに表示する。
mv[w]getstr	カーソルを移動し，端末画面から文字列を取得し，その文字列を変数に格納し，指定されたウィンドウに表示する。
mv[w]inch	カーソルを移動し，ウィンドウを変更せずに，指定されたウィンドウの文字を返す。
mv[w]insch	カーソルを移動し，指定されたウィンドウに文字を挿入する。
mv[w]insstr	カーソルを移動し，指定されたウィンドウに文字列を挿入する。
mvwin	ウィンドウの開始位置を指定された座標に移動する。
newwin	端末画面の指定された座標から始まる行と列で新しいウィンドウを作成する。
[no]nl	UNIX ソフトウェアとの互換性を維持するためだけに提供され，OpenVMS 環境では機能しない。
overlay	1 つのウィンドウの内容 (別のウィンドウの内容の上に完全に収まるサイズ) を別のウィンドウの内容に上に完全に書き込む。処理は 2 つのウィンドウの開始座標から開始される。
overwrite	1 つのウィンドウの内容を別のウィンドウの上に収まる範囲で書き込む。処理は 2 つのウィンドウの開始座標から開始される。
[w]printw	現在のカーソルの位置からウィンドウに対して printf を実行する。
[no]raw	UNIX ソフトウェアとの互換性を維持するためだけに提供され，OpenVMS 環境では機能しない。
[w]refresh	指定されたウィンドウを端末画面に再表示する。
[w]scanw	ウィンドウに対して scanf を実行する。
scroll	ウィンドウのすべての行を 1 行ずつ上に移動する。
scrollok	指定されたウィンドウのスクロール・フラグを設定する。
[w]setattr	ウィンドウ内でビデオ表示属性を有効にする。
[w]standend	指定されたウィンドウの太字属性を無効にする。
[w]standout	指定されたウィンドウの太字属性を有効にする。
subwin	端末画面で指定された座標から始まる行と列で新しいサブウィンドウを作成する。
touchwin	指定されたウィンドウの最新バージョンを端末画面に表示する。
wrapok	OpenVMS Curses のみ。ウィンドウの右端から次の行の先頭への単語の折り返しを認める。

6.3 Curses の用語

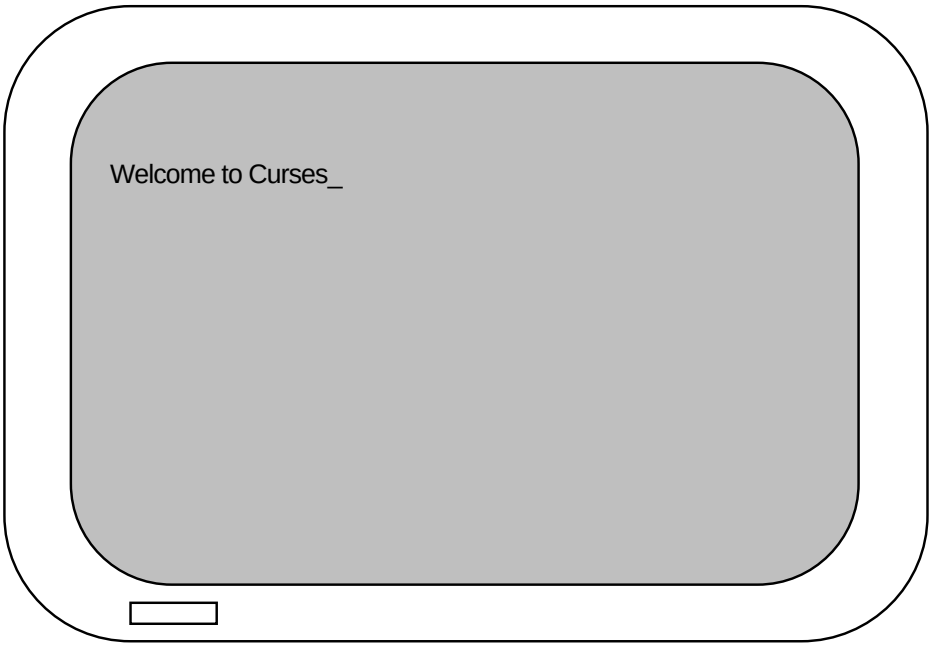
ここでは，Curses の用語について説明し，Curses で端末画面がどのように取り扱われるかを示します。

Curses アプリケーションは重なり合う複数のウィンドウであると考えることができます。ウィンドウの重なり合いのことを**重なり** (occlusion)と呼びます。これらの重なり合うウィンドウの境界を区別するには，四角形のウィンドウに指定の文字で輪郭をつけるか，または反転表示オプションをオンにします (ウィンドウの表示を明るい背景，暗い描画色に設定します)。

6.3.1 定義済みウィンドウ (stdscr と curscr)

端末画面の初期サイズ設定は，Curses で 2 つのウィンドウによってあらかじめ定義されています。これらのウィンドウを`stdscr`と`curscr`と呼びます。`stdscr`ウィンドウはユーザが使用するために定義されています。多くの Curses マクロのデフォルトはこのウィンドウに設定されています。たとえば，`stdscr`の周囲に四角形を描き，画面の左上にカーソルを移動し，文字列を`stdscr`に書き込み，端末画面に`stdscr`を表示すると，表示は図 6–1 に示すようになります。

図 6–1 stdscr ウィンドウの例



ZK−5752−GE

2 番目の定義済みウィンドウであるcurscrは，内部的な Curses の動作のために設計されています。これは，端末画面に現在表示されているもののイメージです。このウィンドウを引数として受け付けるHP C for OpenVMS Curses 関数はclearokだけです。curscrに対して，書き込みや読み込みを行うことはできません。すべての Curses アプリケーションで，stdscrおよびユーザ定義ウィンドウを使用してください。

6.3.2 ユーザ定義ウィンドウ

ユーザは，stdscrの上に自分のウィンドウを重ねることができます。各ウィンドウのサイズと位置は，行数，列数，開始位置によって示されます。

端末画面の行と列によって座標系，つまりウィンドウが作成されるグリッドが形成されます。ウィンドウの開始位置は，ウィンドウの左上の角の (y,x) 座標によって指定します。たとえば，端末画面の座標 (0,0) は画面の左上の角です。

ウィンドウの領域全体が端末画面の境界線の内部になければなりません。ウィンドウのサイズは 1 文字の大きさから端末画面全体の大きさまで，自由に設定できます。また，メモリの範囲内であれば，ウィンドウはいくつでも作成できます。

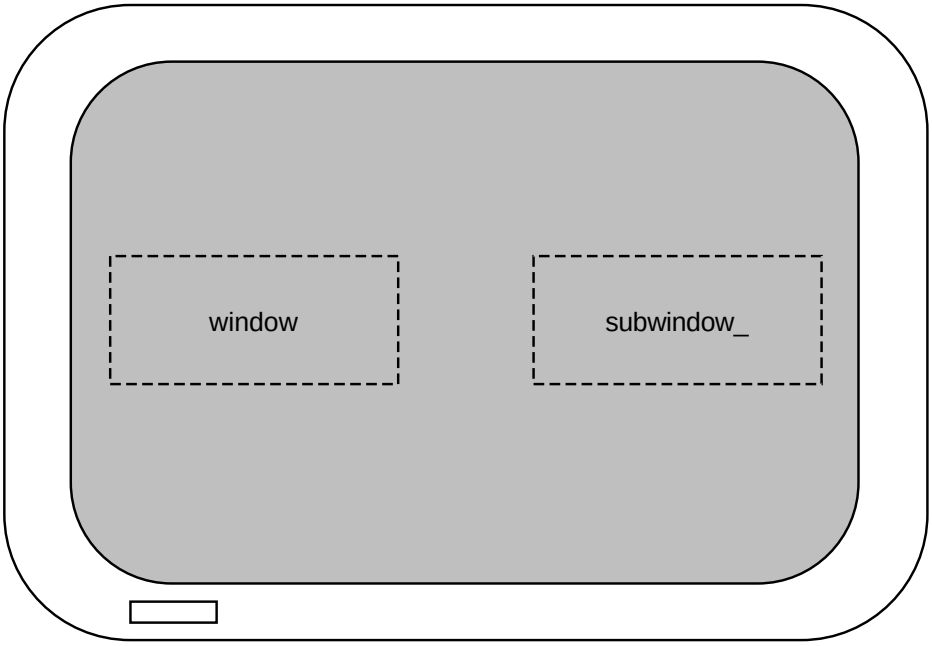
ウィンドウに書き込んだり，ウィンドウから削除しても，ウィンドウを**再表示**するまで，端末画面に変更結果は表示されません。ウィンドウを再表示すると，更新されたウィンドウが端末画面に表示されますが，画面の他の部分はそのまま変更されません。

デフォルト設定では，すべてのユーザ定義ウィンドウがstdscrに重なります。stdscrと重なるだけでなく，互いに重なり合う 2 つ以上のウィンドウを作成することができます。重なり合う一方のウィンドウにデータを書き込んでも，もう一方のウィンドウにデータは書き込まれません。

オーバーラップ・ウィンドウ (**サブウィンドウ**と呼びます) を作成することができます。宣言されるウィンドウには，サブウィンドウの領域全体を収納しなければなりません。データをサブウィンドウに書き込んだり，サブウィンドウがオーバーラップしているウィンドウの部分にデータを書き込むと，どちらのウィンドウにも新しいデータが表示されます。たとえば，サブウィンドウにデータを書き込んだ後，そのサブウィンドウを削除しても，データは下にあるウィンドウにそのまま残されます。

stdscrと重なるウィンドウ，およびstdscrのサブウィンドウを作成すると，端末画面は図 6–2 のようになります。

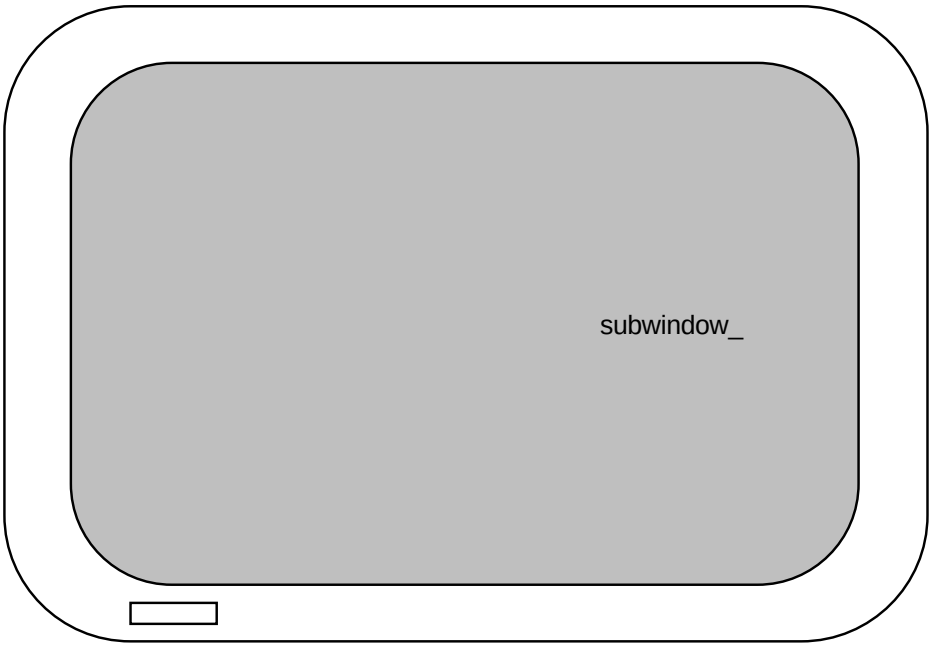
図 6–2 ウィンドウとサブウィンドウの表示



ZK−5754−GE

ユーザ定義ウィンドウとサブウィンドウの両方を削除した後，新しいイメージで端末画面を更新すると，端末画面は図 6–3 のようになります。

図 6–3 端末画面の更新



ZK−5753−GE

ウィンドウに書き込んだ文字列は削除されますが，サブウィンドウに書き込んだ文字列はstdscrの上に残されます。

6.4 Curses の概要

Curses Screen Management 関数およびマクロを使用するときに，端末画面の初期化と復元のために使用しなければならないコマンドがあります。また， Curses が依存する定義済み変数と定数もあります。例 6–1 は Curses を使用してプログラムを設定する方法を示しています。

例 6–1 Curses プログラム

```
1  #include <curses.h>
2  WINDOW  *win1, *win2, *win3;
   main()
   {
3     initscr();
         .
         .
         .
      endwin();
   }
```

例 6–1 の説明:

1. プリプロセッサ・ディレクティブは<curses.h>ヘッダ・ファイルを取り込み，このヘッダ・ファイルが Curses を実装するために使用する構造体と変数を定義します。<curses.h>ヘッダ・ファイルは<stdio.h>ヘッダ・ファイルを取り込みます。したがって，プログラムのソース・コードで<stdio.h>を再び取り込むことにより，この操作を重複して実行する必要はありません。 Curses 関数やマクロを使用するには，<curses.h>を取り込む必要があります。
2. 例の中で， WINDOW は<curses.h>に定義されている構造体です。各ユーザ指定ウィンドウはこの方法で宣言しなければなりません。例 6–1 では，win1，win2，win3の 3 つのウィンドウが定義されています。
3. initscr関数はウィンドウの編集セッションを開始し，endwin関数はウィンドウ編集セッションを終了します。initscr関数は端末画面をクリアし (OpenVMS Curses の場合のみで， BSD ベースの Curses では画面をクリアしません)，ウィンドウstdscrおよびcurscrの領域を割り当てます。endwin関数はすべてのウィンドウを削除し，端末画面をクリアします。

大部分の Curses ユーザはウィンドウの定義と変更を行う必要があります。例 6–2 は， 1 つのウィンドウの定義とそのウィンドウへの書き込みの方法を示しています。

例 6–2 ウィンドウの操作

```
#include <curses.h>

WINDOW  *win1, *win2, *win3;

main()
{
   initscr();
1  win1 = newwin(24, 80, 0, 0);
2  mvwaddstr(win1, 2, 2, "HELLO");
      .
      .
      .
   endwin();
}
```

例 6–2 の説明:

1 newwin関数は，高さが 24 行，幅が 80 カラム，開始位置が座標 (0,0)，つまり端末画面の左上の角であるウィンドウを定義します。プログラムでは，これらの属性をwin1に代入しています。座標は (行, カラム) または (y,x) として指定します。

2 mvwaddstrマクロは，moveマクロとaddstrマクロを別々に呼び出した場合と同じ処理を実行します。mvwaddstrマクロは，カーソルを指定された座標に移動し，文字列をstdscrに書き込みます。

注意

ほとんどの Curses マクロは，デフォルト設定でstdscrを更新します。他のウィンドウを更新する Curses 関数は，マクロと同じ名前ですが，先頭に接頭語“w”が付いています。たとえば，addstrマクロは，stdscrの現在のカーソルの位置に指定された文字列を追加します。waddstr関数は，指定されたウィンドウの現在のカーソルの位置に指定された文字列を追加します。

ウィンドウを更新する場合は，端末画面の原点ではなく，ウィンドウの原点を基準にしてカーソルの位置を指定します。たとえば，ウィンドウの開始位置が (10,10) で，ウィンドウの開始位置に文字を追加する場合は，(10,10) ではなく，座標 (0,0) を指定します。

例 6–2 の HELLO という文字列は，画面を再表示するまで端末画面に表示されません。画面を再表示するには，wrefresh関数を使用します。例 6–3 は，端末画面にwin1の内容を表示する方法を示しています。

例 6–3 端末画面の再表示

```
#include <curses.h>

WINDOW  *win1, *win2, *win3;

main()
{
   initscr();

   win1 = newwin(22, 60, 0, 0);
   mvwaddstr(win1, 2, 2, "HELLO");
   wrefresh(win1);
      .
      .
      .
   endwin();
}
```

wrefresh関数は，端末画面で指定されたウィンドウの領域だけを更新します。このプログラムを実行すると，プログラムがendwin関数を実行するまで， HELLO という文字列が端末画面に表示されます。wrefresh関数は，端末画面上のウィンドウの中で，別のウィンドウに重なっていない部分だけを再表示します。win1が別のウィンドウと重なっているときに，win1の全部を端末画面に表示するには，touchwin関数を呼び出します。

6.5 定義済み変数と定数

<curses.h>ヘッダ・ファイルには， Curses の実装に役立つ変数と定数が定義されています (表 6–2 を参照)。

表 6–2 Curses の定義済み変数と#define 定数

名前	型	説明
curscr	WINDOW *	現在の画面のウィンドウ
stdscr	WINDOW *	デフォルト・ウィンドウ
LINES	int	端末画面の行数
COLS	int	端末画面のカラム数
ERR	—	異常終了したルーチンのフラグ (0)
OK	—	正常終了したルーチンのフラグ (1)
TRUE	—	ブール値 TRUE フラグ (1)
FALSE	—	ブール値 FALSE フラグ (0)
_BLINK	—	setattrとclrattrのパラメータ
_BOLD	—	setattrとclrattrのパラメータ

(次ページに続く)

表 6–2 (続き) Curses の定義済み変数と#define 定数

名前	型	説明
_REVERSE	—	setattrとclrattrのパラメータ
_UNDERLINE	—	setattrと clrattrのパラメータ

たとえば，定義済みマクロ ERR を使用して， Curses 関数の正常終了または異常終了を判定することができます。例 6–4 はこのような判定を行う方法を示しています。

例 6–4 Curses の定義済み変数

```
#include <curses.h>

WINDOW  *win1, *win2, *win3;

main()
{
   initscr();
   win1 = newwin(10, 10, 1, 5);
      .
      .
      .
   if (mvwin(win1, 1, 10) == ERR)
      addstr("The MVWIN function failed.");
      .
      .
      .
   endwin();
}
```

例 6–4 で，mvwin関数が異常終了すると，プログラムは結果を示す文字列をstdscrに追加します。 Curses のmvwin関数はウィンドウの開始位置を移動します。

6.6 カーソルの移動

UNIX システム環境では， Curses 関数を使用して端末画面上でカーソルを移動できます。他の実装では， Curses がmove関数を使用してカーソルを移動することができ，ユーザがmvcur関数にカーソルの始点と終点を指定することもできます。この関数はカーソルを効率のよい方法で移動します。

HP C for OpenVMS Systems では， 2 つの関数は機能的に同じであり，カーソルを移動する効率も同じです。

例 6–5 は，move関数とmvcur関数の使用方法を示しています。

例 6–5 カーソル移動関数

```
#include <curses.h>

main()
{
   initscr();
      .
      .
      .
1    clear();
2    move(10, 10);
3    move(LINES/2, COLS/2);
4    mvcur(0, COLS-1, LINES-1, 0);
      .
      .
      .
   endwin();
}
```

例 6–5 の説明:

1 clearマクロは，stdscrを消去し，カーソルを座標 (0,0) に移動します。

2 最初のmoveはカーソルを座標 (10,10) に移動します。

3 2 番目のmoveは定義済み変数 LINES と COLS を使用して，画面の中心を計算します (画面上の LINES と COLS の値の半分の値を計算)。

4 mvcur関数は絶対アドレッシングを使用します。この関数はカーソルが現在右上の角にあることを要求することにより，画面の左下の角のアドレスを指定できます。カーソルの現在の位置が不確実な場合はこの方法を使用できますが，moveも同様の動作をします。

6.7 プログラムの例

次のプログラムの例は，多くの Curses マクロと関数の結果を示しています。プログラム・コードの各行の内容の理解に役立つように，各行の右側のコメントに説明が示されています。関数の詳細については，ソース・コード・リストの後の説明を参照してください。

例 6–6 は， 1 つのユーザ定義ウィンドウとstdscrの定義および操作を示しています。

例 6–6 stdscr と，それに重なるウィンドウ

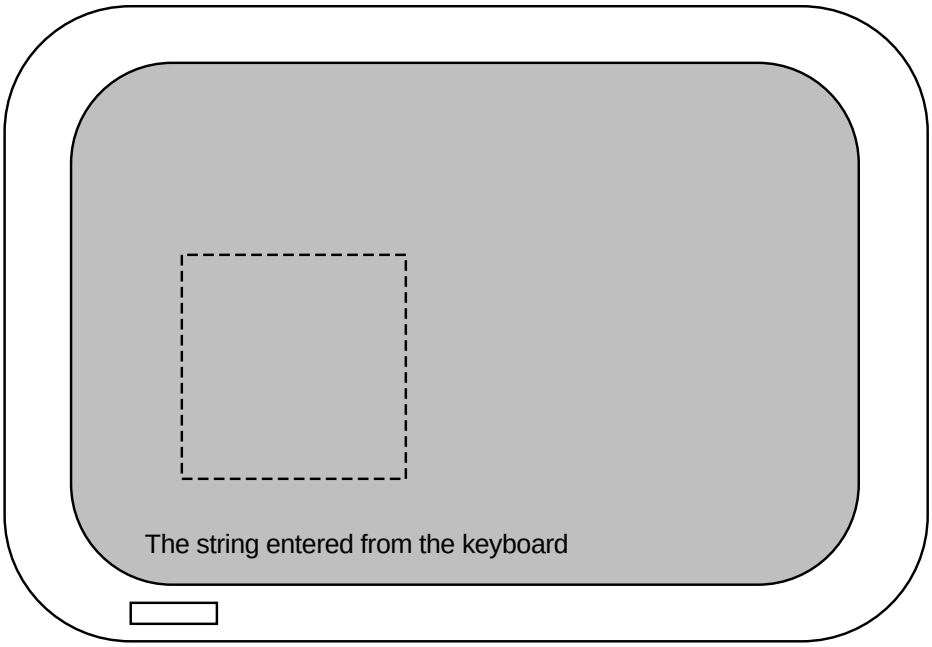
```
/*        CHAP_6_STDSCR_OCCLUDE.C                        */

/* This program defines one window: win1.  win1 is      */
/* located towards the center of the default window     */
/* stdscr.  When writing to an occluding window (win1)  */
/* that is later erased, the writing is erased as well. */

#include <curses.h>     /* Include header file.         */

WINDOW *win1;                   /* Define windows.      */

main()
{
    char str[80];               /* Variable declaration.*/

    initscr();  /* Set up Curses.   */
    noecho();   /* Turn off echo.   */

    /* Create window.             */
    win1 = newwin(10, 20, 10, 10);

    box(stdscr, '|', '-');  /* Draw a box around stdscr. */
    box(win1, '|', '-');    /* Draw a box around win1.   */

    refresh();  /* Display stdscr on screen.  */

    wrefresh(win1);     /* Display win1 on screen.  */

1   getstr(str);      /* Pause. Type a few words!   */

    mvaddstr(22, 1, str);
2   getch();
      /* Add string to win1.        */

    mvwaddstr(win1, 5, 5, "Hello");
    wrefresh(win1);     /* Add win1 to terminal scr.  */

    getch();    /* Pause.  Press Return.      */

    delwin(win1);       /* Delete win1.           */

3   touchwin(stdscr); /* Refresh all of stdscr.     */

    getch();                    /* Pause.  Press Return.      */
    endwin();                   /* Ends session.              */
}
```

例 6–6 の説明:

1 プログラムは入力を待機します。端末画面へのエコー表示は，noechoマクロを使用して禁止されているため，入力した単語はstdscrに表示されません。しかし，マクロは入力された単語をプログラムの他の場所で使用するために，変数strに格納します。

2 getchマクロを実行すると，プログラムは一時停止します。画面の確認が完了したら， Return を押すことにより，プログラムを再開できます。getchマクロは，refreshを呼び出さずに，端末画面でstdscrを再表示します。画面は図 6–4 のようになります。

図 6–4　getch **マクロの例**



ZK−5751−GE

3　touchwin関数は画面を再表示することで，stdscr全体が表示されるようにし，削除した重なり合うウィンドウが画面に表示されないようにします。

7

算術関数

表 7–1 は，HP C Run-Time Library (RTL) の算術関数を示しています。各関数の詳細については，『HP C ランタイム・ライブラリ・リファレンス・マニュアル (下巻)』「リファレンス・セクション」を参照してください。

表 7–1 算術関数

関数	説明
abs	整数の絶対値を返す。
acos	ラジアン単位の引数の逆余弦 (アークコサイン) を[0,π]ラジアンの範囲で返す。
acosd *(Integrity， Alpha)*	ラジアン単位の引数の逆余弦 (アークコサイン) を[0,180]度の範囲で返す。
acosh *(Integrity， Alpha)*	引数の双曲線逆余弦 (アークコサイン) を返す。
asin	ラジアン単位の引数の逆正弦 (アークサイン) を$[-\pi/2, \pi/2]$ラジアンの範囲で返す。
asind *(Integrity， Alpha)*	ラジアン単位の引数の逆正弦 (アークサイン) を$[-90, 90]$度の範囲で返す。
asinh *(Integrity， Alpha)*	引数の双曲線逆正弦 (アークサイン) を返す。
atan	ラジアン単位の引数の逆正接 (アークタンジェント) を$[-\pi/2, \pi/2]$ラジアンの範囲で返す。
atand *(Integrity， Alpha)*	ラジアン単位の引数の逆正接 (アークタンジェント) を$[-90, 90]$度の範囲で返す。
atan2	y/x (2 つのラジアン単位の引数) の逆正接 (アークタンジェント) を$[-\pi, \pi]$ラジアンの範囲で返す。
atand2 *(Integrity， Alpha)*	y/x (2 つのラジアン単位の引数) の逆正接 (アークタンジェント) を$[-180, 180]$度の範囲で返す。
atanh *(Integrity， Alpha)*	ラジアン単位の引数の双曲線逆正接 (アークタンジェント) を返す。
cabs	複素数の絶対値をsqrt ($x^2 + y^2$)として返す。
cbrt *(Integrity， Alpha)*	引数の四捨五入した立方根を返す。
ceil	引数に等しいか，それ以上の最小の整数を返す。
copysign *(Integrity， Alpha)*	最初の引数を 2 番目の引数と同じ符号で返す。
cos	ラジアン単位の引数の余弦 (コサイン) をラジアン単位で返す。
cosd *(Integrity， Alpha)*	ラジアン単位の引数の余弦 (コサイン) を度単位で返す。
cosh	引数の双曲線余弦 (コサイン) を返す。

(次ページに続く)

表 7–1 (続き) 算術関数

関数	説明
cot	ラジアン単位の引数の余接 (コタンジェント) をラジアン単位で返す。
cotd *(Integrity, Alpha)*	ラジアン単位の引数の余接 (コタンジェント) を度単位で返す。
drand48, erand48, jrand48, lrand48, mrand48, nrand48	均一に分布した擬似乱数シーケンスを生成する。 48 ビットの負でない倍精度浮動小数点数値を返す。
erf *(Integrity, Alpha)*	引数のエラー関数を返す。
erfc *(Integrity, Alpha)*	(1.0 − erf(x)) を返す。
exp	e を底とする引数のべき乗を返す。
expm1 *(Integrity, Alpha)*	exp(x) − 1 を返す。
fabs	浮動小数点数値の絶対値を返す。
finite *(Integrity, Alpha)*	引数が有限の数値の場合は 1 を返し，無限の数値の場合は 0 を返す。
floor	引数に等しいか，それより小さい最大の整数を返す。
fmod	最初の引数を 2 番目の引数で除算した余りを浮動小数点数値として計算する。
fp_class *(Integrity, Alpha)*	IEEE 浮動小数点数値のクラスを判断し，<fp_class.h>ヘッダ・ファイルから定数を返す。
isnan *(Integrity, Alpha)*	NaN の判定をする。引数が NaN の場合は 1 を返し，それ以外の場合は 0 を返す。
j0, j1, jn *(Integrity, Alpha)*	第 1 種ベッセル関数を計算する。
frexp	浮動小数点数値の小数部と指数部を計算する。
hypot	2 つの引数の 2 乗の合計の平方根を返す。
initstate	乱数ジェネレータを初期化する。
labs	整数の絶対値を long int として返す。
lcong48	48 ビットの均一に分布した擬似乱数シーケンスを初期化する。
lgamma *(Integrity, Alpha)*	ガンマ関数の対数を計算する。
llabs, qabs *(Integrity, Alpha)*	__int64整数の絶対値を返す。
ldexp	最初の引数に， 2 を底とする 2 番目の引数のべき乗を乗算した値を返す。
ldiv, div	引数を除算して商と余りを返す。
lldiv, qdiv *(Integrity, Alpha)*	引数を除算して商と余りを返す。
log2 *(Integrity, Alpha)*, log, log10	引数の対数を返す。
log1p *(Integrity, Alpha)*	ln(1+x) を正確に計算する。
logb *(Integrity, Alpha)*	基数に依存しない引数の指数を返す。

(次ページに続く)

表 7–1 (続き) 算術関数

関数	説明
nextafter *(Integrity, Alpha)*	yの方向に，マシンで表現可能なxの次の数値を返す。
nint *(Integrity, Alpha)*	引数に最も近い整数値を返す。
modf	最初の引数の正の小数部を返し，2 番目の引数によってアドレスが指定されるオブジェクトに整数部を代入する。
pow	最初の引数を底として，2 番目の引数のべき乗を返す。
rand, srand	0 ～$2^{31} - 1$ の範囲で擬似乱数を返す。
random, srandom	よりランダムなシーケンスで擬似乱数を生成する。
rint *(Integrity, Alpha)*	ユーザが指定した現在の IEEE 丸め方向に従って，引数を整数値に丸める。
scalb *(Integrity, Alpha)*	浮動小数点数値の指数部を返す。
seed48, srand48	48 ビットの乱数ジェネレータを初期化する。
setstate	再起動して，乱数ジェネレータを変更する。
sin	ラジアン単位の引数の正弦 (サイン) をラジアン単位で返す。
sind *(Integrity, Alpha)*	ラジアン単位の引数の正弦 (サイン) を度単位で返す。
sinh	引数の双曲線正弦 (サイン) を返す。
sqrt	引数の平方根を返す。
tan	ラジアン単位の引数の正接 (タンジェント) をラジアン単位で返す。
tand *(Integrity, Alpha)*	ラジアン単位の引数の正接 (タンジェント) を度単位で返す。
tanh	引数の双曲線正接 (タンジェント) を返す。
trunc *(Integrity, Alpha)*	引数を整数値に切り捨てる。
unordered *(Integrity, Alpha)*	引数のいずれか一方または両方が NaN の場合は 1 を返し，それ以外の場合は 0 を返す。
y0, y1, yn *(Integrity, Alpha)*	第 2 種ベッセル関数を計算する。

7.1 算術関数のバリアント —float, long double *(Integrity, Alpha)*

OpenVMS Alpha システムと OpenVMS Integrity システムでのみ，HP Cに対して追加の算術演算ルーチンがサポートされます。これらのルーチンは<math.h>に定義されており，表 7–1 に示したルーチンのfloatバリアントとlong doubleバリアントです。

floatバリアントはfloat引数を受け付け，floatの値を返します。名前の最後に接尾語としてfが付加されます。次の例を参照してください。

```
float cosf (float x);
float tandf (float x);
```

long doubleバリアントはlong double引数を受け付け，long doubleの値を返します。名前の最後に接尾語としてlが付加されます。次の例を参照してください。

```
long double cosl (long double x);
long double tandl (long double x);
```

算術演算ルーチンのバリアントはすべて，本書の「リファレンス・セクション」に説明されています。

/L_DOUBLE=64 を指定せずにコンパイルしたプログラム (つまり，デフォルトの/L_DOUBLE=128 でコンパイルしたプログラム) の場合，これらのHP C RTL 算術演算ルーチンのlong doubleバリアントは，『HP Portable Mathematics Library (HPML)』マニュアルに説明されている X_FLOAT エントリ・ポイントにマッピングされます。

7.2 エラーの検出

実行時エラーを検出するのに役立つように，<errno.h>ヘッダ・ファイルには，多くの (全部ではない) 算術関数から返される次の 2 つのシンボル値が定義されています。

- EDOM は，引数が不適切であることを示します。引数が関数の領域の範囲内にありません。
- ERANGE は，結果が適切な範囲内にないことを示します。引数が大きすぎるか小さすぎるために，マシンで表現できません。

算術関数を使用する場合，外部変数errnoを調べて，これらの値のいずれか一方または両方が格納されていないかどうかを確認し，エラーが発生している場合は適切な処置を実行します。

次のプログラムの例では，errno変数の値が EDOM でないかどうかを調べます。この値は，sqrt関数に対する入力として負の数値が指定されたことを示します。

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

main()
{
   double input, square_root;

   printf("Enter a number: ");
   scanf("%le", &input);
   errno = 0;
   square_root = sqrt(input);

   if (errno == EDOM)
      perror("Input was negative");
   else
      printf("Square root of %e = %e\n",
              input, square_root);
}
```

errnoでこのシンボル値を調べなかった場合，負の値が入力されると，sqrt関数は 0 を返します。<errno.h>ヘッダ・ファイルの詳細については，第 4 章を参照してください。

7.3 <fp.h>ヘッダ・ファイル

<fp.h>ヘッダ・ファイルは， ANSI X3J11 委員会の Numerical C Extensions Group が定義している一部の機能を実装しています。浮動小数点関数を広範囲にわたって使用するアプリケーションの場合は，このヘッダ・ファイルが役立ちます。

この章に示した一部の倍精度関数は，結果が表現できない範囲である場合，± HUGE_VAL (<math.h>または<fp.h>に定義) という値を返します。これらの関数のfloatバージョンは，同じ条件で HUGE_VALF (<fp.h>にのみ定義) という値を返します。long doubleバージョンは， HUGE_VALL (<fp.h>に定義) という値を返します。

IEEE の無限大および NaN の値を有効にしてコンパイルされたプログラムの場合，HUGE_VAL， HUGE_VALF， HUGE_VALL という値は式であり，コンパイル時定数ではありません。次のような初期化を行うと，コンパイル時エラーが発生します。

```
$ CREATE IEEE_INFINITY.C
#include <fp.h>

double my_huge_val = HUGE_VAL
^Z
$ CC /FLOAT=IEEE/IEEE=DENORM IEEE_INFINITY

double my_huge_val = HUGE_VAL;
.....................^
%CC-E-NEEDCONSTEXPR, In the initializer for my_huge_val, "decc$gt_dbl_infinity"
is not constant, but occurs in a context that requires a constant expression.
at line number 3 in file WORK1$:[RTL]IEEE_INFINITY.C;1
$
```

<math.h>および<fp.h>の両方を使用する場合，<math.h>は関数isnanを定義し，<fp.h>は同じ名前のマクロを定義します。アプリケーションで最初に取り込まれたヘッダ・ファイルがisnanに対する参照を解決します。

7.4 例

例 7–1 は，tan，sin，cos関数がどのように動作するかを示しています。

例 7–1 正接 (タンジェント) 値の計算と検証

(次ページに続く)

例 7–1 (続き) 正接 (タンジェント) 値の計算と検証

```
/*        CHAP_7_MATH_EXAMPLE.C                                  */

/* This example uses two functions --- mytan and main ---       */
/* to calculate the tangent value of a number, and to check     */
/* the calculation using the sin and cos functions.             */

#include <math.h>
#include <stdio.h>

/* This function calculates the tangent using the sin and       */
/* cos functions.                                               */

double mytan(x)
    double x;
{
    double y,
           y1,
           y2;

    y1 = sin(x);
    y2 = cos(x);

    if (y2 == 0)
        y = 0;
    else
        y = y1 / y2;

    return y;
}
main()
{
    double x;

    /*  Print values: compare   */
    for (x = 0.0; x < 1.5; x += 0.1)
        printf("tan of %4.1f = %6.2f\t%6.2f\n", x, mytan(x), tan(x));
}
```

例 7–1 は次の出力を生成します。

```
$ RUN  EXAMPLE
tan of  0.0 =   0.00      0.00
tan of  0.1 =   0.10      0.10
tan of  0.2 =   0.20      0.20
tan of  0.3 =   0.31      0.31
tan of  0.4 =   0.42      0.42
tan of  0.5 =   0.55      0.55
tan of  0.6 =   0.68      0.68
tan of  0.7 =   0.84      0.84
tan of  0.8 =   1.03      1.03
tan of  0.9 =   1.26      1.26
tan of  1.0 =   1.56      1.56
tan of  1.1 =   1.96      1.96
tan of  1.2 =   2.57      2.57
tan of  1.3 =   3.60      3.60
tan of  1.4 =   5.80      5.80
$
```

8
メモリ割り当て関数

表 8–1 は，HP C Run-Time Library (RTL) のすべてのメモリ割り当て関数を示しています。各関数の詳細については，『HP C ランタイム・ライブラリ・リファレンス・マニュアル (下巻)』「リファレンス・セクション」を参照してください。

表 8–1 メモリ割り当て関数

関数	説明
brk, sbrk	プログラムで使用されていない最下位仮想アドレスを判断する。
calloc, malloc	メモリ領域を割り当てる。
cfree, free	calloc，malloc，realloc呼び出しによって割り当てられている領域を解放して，再割り当てできるようにする。
realloc	最初の引数によって示される領域のサイズを，2 番目の引数によって指定されるバイト数に変更する。
strdup	文字列を複製します。

ヒープから追加記憶域を要求するすべてのHP C RTL 関数は，HP C RTL のメモリ割り当て関数malloc，calloc，realloc，free，cfreeを使用して，その記憶域を取得します。これらの関数によって割り当てられるメモリは，クォドワード境界に揃えられます。

ANSI C 標準には，cfreeは含まれていません。この理由から，メモリの割り当てを解除するには，同じ機能を実行するfree関数を使用するようにしてください。

brk関数とsbrk関数では，メモリはアドレス空間の一番上から連続的に割り当てることができるものと仮定されています。しかし，malloc関数と RMS は，これと同じアドレス空間から領域を割り当てる可能性があります。mallocを使用するHP C RTL ルーチンや RMS と組み合わせて，brk関数およびsbrk関数を使用しないでください。

VAX C RTL に関するドキュメントの以前のバージョンには，メモリ割り当てルーチンは OpenVMS RTL の関数 LIB$GET_VM を使用して動的メモリを取得し，LIB$FREE_VM を使用して動的メモリを返すと示されていました。しかし，現在はこのようになっていません。これらのルーチンとHP C RTL メモリ割り当てルーチンの間の相互関係の問題はなくなりました (ただし，LIB$SHOW_VM を使用して，HP C RTL のmallocおよびfreeの使用状況を追跡することはできなくなりました)。

HP C RTL のメモリ割り当て関数calloc，malloc，realloc，freeはそれぞれ，LIB$ルーチン LIB$VM_CALLOC， LIB$VM_MALLOC， LIB$VM_REALLOC，LIB$VM_FREE をベースにしています。

ルーチンVAXC$CALLOC_OPT，VAXC$CFREE_OPT，VAXC$FREE_OPT，VAXC$MALLOC_OPT，VAXC$REALLOC_OPTは破棄されましたので，今後の開発では使用できません。しかし，下位互換性を維持するために，標準的な C メモリ割り当てルーチンと同等として，これらのルーチンのバージョンも提供されています。

8.1 プログラムの例

例 8–1 はmalloc，calloc，free関数の使用方法を示しています。

例 8–1 構造体に対するメモリの割り当てと割り当ての解除

```
/*        CHAP_8_MEM_MANAGEMENT.C                                  */

/* This example takes lines of input from the terminal until    */
/* it encounters a Ctrl/Z, places the strings into an           */
/* allocated buffer, copies the strings to memory allocated for */
/* structures, prints the lines back to the screen, and then    */
/* deallocates all the memory used for the structures.          */

#include  <stdlib.h>
#include  <stdio.h>
#define   MAX_LINE_LENGTH  80

struct line_rec {              /*  Declare the structure.   */
    struct line_rec *next;     /*  Pointer to next line.    */
    char *data;                /*  A line from terminal.    */
};

int main(void)
{
    char *buffer;

    /* Define pointers to        */
    /*  structure (input lines). */

    struct line_rec *first_line = NULL,
            *next_line,
            *last_line = NULL;

    /* Buffer points to memory.  */
    buffer = malloc(MAX_LINE_LENGTH);

    if (buffer == NULL) {      /* If error ...  */
        perror("malloc");
        exit(EXIT_FAILURE);
    }
```

(次ページに続く)

例 8–1 (続き) 構造体に対するメモリの割り当てと割り当ての解除

```
    while (gets(buffer) != NULL) {      /* While not Ctrl/Z ...  */
        /* Allocate for input line.  */
        next_line = calloc(1, sizeof (struct line_rec));

        if (next_line == NULL) {
            perror("calloc");
            exit(EXIT_FAILURE);
        }

        /* Put line in data area.    */
        next_line->data = buffer;

        if (last_line == NULL)  /* Reset pointers.   */
            first_line = next_line;
        else
            last_line->next = next_line;

        last_line = next_line;
        /* Allocate space for the     */
        /*  next input line.          */
        buffer = malloc(MAX_LINE_LENGTH);

        if (buffer == NULL) {
            perror("malloc");
            exit(EXIT_FAILURE);
        }
    }
    free(buffer);       /* Last buffer always unused. */
    next_line = first_line;     /* Pointer to beginning.  */

    while (next_line != NULL) {
        puts(next_line->data);  /* Write line to screen.  */
        free(next_line->data);  /* Deallocate a line.     */
        last_line = next_line;
        next_line = next_line->next;
        free(last_line);
    }

    exit(EXIT_SUCCESS);
}
```

次の例は，例 8–1 の入力と出力を示しています。

```
$ RUN  EXAMPLE
line one
line two
Ctrl/Z
EXIT
line one
line two
$
```

9
システム関数

オペレーティング・システムの開発には，C プログラミング言語が適しています。たとえば，UNIX オペレーティング・システムの大部分は C で書かれています。システム・プログラムを作成する場合，プログラムが動作する環境を検索または変更しなければならないことがあります。この章では，このような作業やその他のシステム・タスクを実行するためのHP C Run-Time Library (RTL) 関数について説明します。

表 9–1 は，HP C RTL で提供されるすべてのシステム関数を示しています。各関数の詳細については，『HP C ランタイム・ライブラリ・リファレンス・マニュアル (下巻)』「リファレンス・セクション」を参照してください。

表 9–1 システム関数

関数	説明
システム関数 — 検索およびソート・ユーティリティ	
bsearch	ソートされたオブジェクトの配列でバイナリ検索を実行して，指定されたオブジェクトを検索する。
qsort	クイック・ソート・アルゴリズムを実装することにより，オブジェクトの配列をソートする。
システム関数 — プロセス情報の検索	
ctermid	SYS$COMMAND の等価文字列を与える文字列を返す。これは制御端末の名前である。
cuserid	現在のプロセスを開始したユーザの名前を格納した文字列を指すポインタを返す。
getcwd	現在のワーキング・ディレクリのファイル指定を指すポインタを返す。
getegid, geteuid, getgid, getuid	OpenVMS の用語で，ユーザ識別コード (UIC) からグループ番号とメンバ番号を返す。
getenv	現在のプロセスの環境配列を検索し，指定された環境に関連付けられている値を返す。
getlogin	現在のセッションに関連付けられているユーザのログイン名を取得する。
getpid	現在のプロセスのプロセス ID を返す。

(次ページに続く)

表 9–1 (続き) システム関数

関数	説明
システム関数 — プロセス情報の検索	
getppid	呼び出しプロセスの親プロセス ID を返す。
getpwnam	ユーザ・データベース内のユーザ名情報にアクセスする。
getpwuid	ユーザ・データベース内のユーザ ID 情報にアクセスする。
システム関数 — プロセス情報の変更	
chdir	デフォルト・ディレクトリを変更する。
chmod	ファイルのファイル保護を変更する。
chown	ファイルのオーナのユーザ識別コード (UIC) を変更する。
mkdir	ディレクトリを作成する。
nice	引数に指定された値だけ，プロセスの基本優先順位に対してプロセス優先順位を上げる，または下げる。
putenv	環境変数を設定する。
setenv	現在の環境リストに環境変数名を挿入する，またはリセットする。
setgid, setuid	プログラムの移植性を確保するために実装されており，機能はない。
sleep, usleep	少なくとも引数に指定された秒数だけ，現在のプロセスの実行を停止する。
umask	新しいファイルが作成されるときに使用されるファイル保護マスクを作成する。古いマスクの値を返す。
システム関数 — 日付/時刻情報の取得と変換	
asctime	年月日時分秒形式の時刻を 26 文字の文字列に変換する。
clock	プログラムの実行開始以降に使用された CPU 時間をマイクロ秒単位で判断する。
clock_getres	指定されたクロックの精度を取得する。
clock_gettime	指定されたクロックの現在の時刻 (秒およびナノ秒) を返す。
clock_settime	指定されたクロックを設定する。
ctime	秒単位の時刻を asctime 関数で生成される形式の ASCII 文字列に変換する。
decc$fix_time	OpenVMS のバイナリ・システム時刻を UNIX のバイナリ時刻に変換する。
difftime	引数によって指定される 2 つの時刻の差を秒単位で計算する。
ftime	1970 年 1 月 1 日 00:00:00 からの経過時間を timeb 構造体に返す。
getclock	システム単位で設定されているクロックの現在の値を取得する。
getdate	書式設定された文字列を時刻/日付構造体に変換する。
getitimer	間隔タイマの値を返す。
gettimeofday	日付と時刻を取得する。
gmtime	時間単位を年月日時分秒形式の UTC 時刻に変換する。

(次ページに続く)

表 9–1 (続き) システム関数

関数	説明
システム関数 — 日付/時刻情報の取得と変換	
localtime	時刻 (1970 年 1 月 1 日 00:00:00 からの経過時間を秒数で表現した時刻) を時，分，秒などに変換する。
mktime	ローカル時刻構造体を Epoch (1970 年 1 月 1 日 00:00:00) からの経過時間に変換する。
nanosleep	高精度の sleep (リアルタイム)。指定された時間，プロセスの実行を一時停止する。
setitimer	間隔タイマの値を設定する。
strftime, wcsftime	指定された書式文字列による制御のもとで，配列に文字を格納する。
strptime	文字列を日付と時刻の値に変換する。
time	1970 年 1 月 1 日 00:00:00 からの経過時間を秒単位で返す。
times	現在のプロセスと終了した子プロセスの累積時間を返す。
tzset	タイム・ゾーン変換を設定する，またはタイム・ゾーン変換にアクセスする。
ualarm	間隔タイマの時間切れを設定または変更する。
wcsftime	tm構造体に格納されている日付と時刻の情報を使用して，ワイド文字の出力文字列を作成する。
システム関数 — その他	
VAXC$CRTL_INIT	実行時環境を初期化し，終了および条件ハンドラを設定する。この結果，HP C RTL 関数を他の言語から呼び出すことができるようになる。

例 9–1 は，cuserid関数の使用方法を示しています。

例 9–1 ユーザ名へのアクセス

```
/*        CHAP_9_GET_USER.C                                    */
/* Using cuserid, this program returns the user name.         */
#include <stdio.h>
main()
{
    static char string[L_cuserid];
    cuserid(string);
    printf("Initiating user: %s\n", string);
}
```

TOLLIVER という名前のユーザがプログラムを実行すると，stdoutに次の情報が表示されます。

```
$ RUN  EXAMPLE1
Initiating user: TOLLIVER
```

例 9–2 は，getenv関数の使用方法を示しています。

例 9–2 端末情報へのアクセス

```
/*        CHAP_9_GETTERM.C                              */

/* Using getenv, this program returns the terminal.     */

#include <stdio.h>
#include <stdlib.h>

main()
{
    printf("Terminal type: %s\n", getenv("TERM"));
}
```

132 カラム・モードの VT100 端末で例 9–2 を実行すると，次の情報が表示されます。

```
$ RUN  EXAMPLE3
Terminal type: vt100-132
```

例 9–3 は，getenvを使用してユーザのデフォルト・ログイン・ディレクトリを調べる方法と，chdirを使用してそのディレクトリに変更する方法を示しています。

例 9–3 デフォルト・ディレクトリの操作

```
/*       CHAP_9_CHANGE_DIR.C                                */

/* This program performs the equivalent of the DCL command      */
/* SET DEFAULT SYS$LOGIN. However, once the program exits, the  */
/* directory is reset to the directory from which the program   */
/* was run.                                                     */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

main()
{
    char *dir;
    int i;

    dir = getenv("HOME");
    if ((i = chdir(dir)) != 0) {
        perror("Cannot set directory");
        exit(0);
    }

    printf("Current directory: %s\n", dir);
}
```

例 9–3 を実行すると，次の情報が表示されます。

```
$ RUN EXAMPLE4
Current directory: dba0:[tolliver]
$
```

例 9–4 は，time，localtime，strftime関数を使用して端末に正しい日付と時刻をプリントする方法を示しています。

例 9–4 日付と時刻のプリント

```
/*        CHAP_9_DATE_TIME.C                                    */

/* The time function returns the time in seconds; the localtime */
/* function converts the time to hours, minutes, and so on;     */
/* the strftime function uses these values to obtain a string   */
/* in the desired format.                                       */

#include <time.h>
#include <stdio.h>

#define MAX_STRING 80

main()
{
    struct tm *time_structure;
    time_t time_val;
    char output_str[MAX_STRING];

    time(&time_val);
    time_structure = localtime(&time_val);

    /* Print the date  */

    strftime(output_str, MAX_STRING,
             "Today is %A, %B %d, %Y", time_structure);

    printf("%s\n", output_str);

    /* Print the time using a 12-hour clock format. */

    strftime(output_str, MAX_STRING,
             "The time is %I:%M %p", time_structure);

    printf("%s\n", output_str);
}
```

例 9–4 を実行すると，次の情報が表示されます。

```
$ RUN  EXAMPLE5
Today is Thursday, May 20, 1993
The time is 10:18 AM
$
```

10

国際化ソフトウェアの開発

この章では，国際化ソフトウェアの典型的な機能と，国際化ソフトウェアを設計および実装するためにHP C Run-Time Library (RTL) で提供される機能について説明します。

この章で説明する関数の詳細については，『HP C ランタイム・ライブラリ・リファレンス・マニュアル (下巻)』「リファレンス・セクション」を参照してください。

10.1 国際化のサポート

HP C RTL では，アプリケーション開発者が国際化ソフトウェアを作成するための機能が追加されました。HP C RTL は，**ロケール**・ファイルから言語とカルチャーに関する情報を読み込んで取得します。

10.1.1 インストール

これらのHP C RTL 機能を使用する場合は，システムにこれらのファイルを提供するために別のキットをインストールする必要があります。『OpenVMS Upgrade and Installation Guide』の付録「Installing OpenVMS Internationalization data kit」を参照してください。

OpenVMS Alpha システムでは，セーブ・セットは Layered Product CD で提供されます。名前は， VMSI18N0nn または ALPVMSI18N0n_07nn です。

このセーブ・セットをインストールするには，このセーブ・セット名をキットの名前として使用して，標準的な OpenVMS のインストール手順に従います。複数のロケール・カテゴリをインストールのために選択できます。次のプロンプトに応答することにより，必要な数だけロケールを選択できます。

```
* Do you want European and US support? [YES]?
* Do you want Chinese GB18030 support (locale and Unicode converters) [YES]?
* Do you want Chinese support? [YES]?
* Do you want Japanese support? [YES]?
* Do you want Korean support? [YES]?
* Do you want Thai support? [YES]?
* Do you want the Unicode converters? [YES]?
```

このキットには Installation Verification Procedure も含まれています。キットが正しくインストールされたかどうかを検証するには，このプロシージャを実行することをお勧めします。

10.1.2 Unicode のサポート

OpenVMS Version 7.2 で，HP Cランタイム・ライブラリに Universal Unicode ロケールが追加されました。このロケールは VMSI18N0nn キットではなく， OpenVMS システムに付属しています。 Unicode ロケールの名前は次のとおりです。

```
UTF8-20
```

VMSI18N0nn キットに付属しているロケールと同様に， Unicode ロケールは SYS$I18N_LOCALE 論理名によって参照される標準的なディレクトリにあります。

UTF8-20 Unicode は， Unicode 標準 V2.0 をベースにしています。 Unicode ロケールでは，ワイド文字エンコーディングとして UCS-4 を使用し，マルチバイト文字エンコーディングとして UTF-8 を使用します。

HP C RTL には， Unicode と他のサポートされる文字セットの間の変換を実行するコンバータも含まれています。コンバータの拡張セットには， UCS-2， UCS-4， UTF-8 の Unicode エンコーディングのためのコンバータが含まれています。 Unicode コンバータは ICONV CONVERT ユーティリティで使用することができ，HP Cランタイム・ライブラリのiconv関数ファミリで使用することもできます。

OpenVMS Version 7.2 では，HP Cランタイム・ライブラリで Microsoft Code Page 437 のための Unicode 文字セット・コンバータが追加されました。

10.2 国際化ソフトウェアの機能

国際化ソフトウェアとは，複数の言語とカルチャーをサポートできるソフトウェアです。国際化プログラムは次のことができなければなりません。

- ユーザの言語でメッセージを表示できなければなりません。画面表示，エラー・メッセージ，プロンプトなどの表示が必要です。
- 次のようなカルチャー固有の情報を取り扱うことができなければなりません。
  - 日付と時刻の書式

    日付と時刻の表現方法は国によって異なります。たとえば，米国では月を最初に指定し，英国では日を最初に指定します。したがって， 12/5/1993 という日付は，米国では 1993 年 12 月 5 日として解釈され，英国では 1993 年 5 月 12 日として解釈されます。
  - 数値の書式

小数点と 3 桁ごとの区切り文字を表す文字は国ごとに異なります。たとえば，英国では小数点を表すためにピリオド (.) を使用し， 3 桁ごとの区切り文字としてコンマを使用します。しかし，ドイツではコンマを小数点として使用し，ピリオドを 3 桁ごとの区切り文字として使用します。したがって，英国の 2,345.67 という数値はドイツの 2.345,67 という数値と同じです。

- 通貨の書式

  通貨は国ごとに異なるシンボルで表され，さまざまな区切り文字を使用して書式設定することができます。

- 異なる文字セット (ASCII だけでない) を取り扱うことができなければなりません。
- シングル・バイト文字とマルチバイト文字の混在を取り扱うことができなければなりません。
- 文字列のマルチパス比較機能を備えていなければなりません。

  strcmpなどの文字列比較関数は，文字列内の文字のコードポイント値を比較することにより，文字列を比較します。しかし，一部の言語では文字列を正しくソートするために，より複雑な比較が必要になることがあります。

上記の要件を満たすには，アプリケーションで言語，各国の習慣，使用される文字セットに関する何らかの仮定を行うべきではありません。このような各国対応データはすべて，プログラムとは別に定義し，実行時に割り当てるようにしなければなりません。

この章では，HP Cを使用して国際化ソフトウェアを作成する方法について説明します。

10.3 HP C を使用した国際化ソフトウェアの開発

HP C環境では，国際化ソフトウェアを開発するために次の機能が提供されます。

- 各国対応データとプログラムを分離する方法

  各国対応データは，**ロケール**というデータベースに格納されます。このデータベースには，プログラムで必要とされるすべての言語情報とカルチャー情報が格納されます。ロケールの構造体の詳細については，第 10.4 節を参照してください。

  プログラムではsetlocale関数を呼び出すことにより，使用するロケールを指定します。詳細については，第 10.5 節を参照してください。

- メッセージ・テキストとプログラム・ソースを分離する方法

  メッセージ・テキストとプログラム・ソースを分離するために，アプリケーションのすべてのメッセージが格納された**メッセージ・カタログ**を使用します。メッセージ・カタログは実行時にアプリケーションにリンクされます。つまり，メッセージを複数の言語に翻訳した後，必要な言語バージョンを実行時に選択することができます。第 10.6 節を参照してください。

- 各国対応データに反応するHP C RTL 関数

  HP C RTL には次の機能を実行する関数が含まれています。

  - 異なるコードセットの間の変換。第 10.7 節を参照。
  - カルチャー固有の情報の取り扱い。第 10.8 節を参照。
  - 文字列のマルチパス照合。第 10.10 節を参照。

- HP C RTL では特殊なワイド文字データ型が定義されているため，シングル・バイト文字とマルチバイト文字が混在するコードセットを簡単に取り扱うことができます。このワイド文字データ型をサポートするために，関数も定義されています。第 10.9 節を参照してください。

10.4 ロケール

ロケールは異なるカテゴリで構成されており，各カテゴリは国際化環境の 1 つの要素を定義します。表 10–1 はロケールのカテゴリと，各カテゴリの説明を示しています。

表 10–1 ロケール・カテゴリ

カテゴリ	説明
LC_COLLATE	照合順序に関する情報が格納されている。
LC_CTYPE	文字の分類に関する情報が格納されている。
LC_MESSAGES	yes/no プロンプトに対する応答として期待される応答を定義する。
LC_MONETARY	通貨の書式情報が格納されている。
LC_NUMERIC	数値の書式情報が格納されている。
LC_TIME	時刻と日付の情報が格納されている。

提供されるロケールは，SYS$I18N_LOCALE論理名によって定義されるディレクトリにあります。ロケールのファイル命名規則は次のとおりです。

```
language_country_codeset.locale
```

ただし，

- languageは言語のニーモニックです。たとえば， EN は English ロケールを示します。
- countryは国のニーモニックです。たとえば， GB は British ロケールを示します。
- codesetはロケールの ISO 標準コードセットの名前です。たとえば， ISO8859-1 は Western European 言語の ISO 8859 コードセットです。サポートされるコードセットの詳細については，第 10.7 節を参照してください。

10.5 setlocale 関数による国際化環境の設定

アプリケーションでは，実行時にsetlocaleを呼び出すことにより国際化環境を設定します。国際化環境は次のいずれかの方法で設定します。

- 1つのロケールによって環境を定義する方法。この場合，各ロケール・カテゴリは同じロケールによって定義されます。
- カテゴリを個別に定義する方法。この場合，実行する操作に応じて異なるロケールを使用する複合環境を定義できます。たとえば，英語のユーザがスペイン語のファイルをアプリケーションで処理する場合，LC_COLLATEカテゴリはスペイン語ロケールによって定義することができますが，他のカテゴリは英語ロケールによって定義されます。この場合，各カテゴリに対して1回ずつsetlocaleを呼び出します。

setlocale関数の構文は次のとおりです。

```
char *setlocale(int category, const char *locale)
```

ただし，

- categoryはカテゴリの名前であるか，またはLC_ALL です。 LC_ALL を指定すると，すべてのカテゴリが同じロケールによって定義されます。複合環境を設定するには，カテゴリ名を指定します。
- localeは次のいずれかです。
  - 使用するロケールの名前

    ユーザがロケールを指定するように設定する場合は，アプリケーションでユーザに対してロケール名の入力を要求し，その名前をsetlocale関数への引数として渡します。ロケール名の形式は次のとおりです。

    ```
    language_country.codeset[@modifier]
    ```

    たとえば，setlocale(LC_COLLATE, "en_US.ISO8859-1")は，LC_COLLATEカテゴリに対してロケールen_US.ISO8859-1を選択します。
  - ""

    このように指定すると，関数は論理名を使用して，指定されたカテゴリに対するロケールを判断します。詳細については，論理名の使用によるロケールの指定を参照してください。

アプリケーションでsetlocale関数を呼び出さなかった場合は，デフォルト・ロケールはCロケールになります。この場合，このようなアプリケーションは現在のロケールで情報を使用する関数を呼び出すことができます。

論理名の使用によるロケールの指定

setlocale関数を呼び出すときに，locale引数として ""を指定した場合，関数はいくつか論理名を調べて，指定されたカテゴリのロケール名を判断します。

国際化環境を定義するためにユーザが設定できる論理名は数多くあります。

- カテゴリに対応する論理名

  たとえば， LC_NUMERIC 論理名は，ユーザの環境の内部でLC_NUMERICカテゴリに関連付けられたロケールを定義します。

- LC_ALL
- LANG

  LANG 論理名はユーザの言語を定義します。

ユーザが定義する論理名の他に，システム・スタートアップ時に設定される多くのシステム単位の論理名があり，これらの論理名はシステムのすべてのユーザのデフォルトの国際化環境を定義します。

- SYS$category

  ただし，categoryはカテゴリの名前です。これはそのカテゴリのシステム・デフォルトを指定します。

- SYS$LC_ALL
- SYS$LANG

setlocale関数は，最初にユーザ定義論理名を確認し，そのような論理名が定義されていない場合は，システム論理名を確認します。

10.6 メッセージ・カタログの使用

国際化ソフトウェアの重要な要件は，ユーザの独自の言語でユーザと通信できなければならないということです。メッセージング・システムを導入することにより，プログラム・メッセージをプログラム・ソースと別に作成し，実行時にプログラムにリンクすることができます。

メッセージはメッセージ・テキスト・ソース・ファイルに定義され， GENCAT コマンドを使用してメッセージ・カタログにコンパイルされます。メッセージ・カタログは，HP C RTL で提供される関数を使用してプログラムからアクセスできます。

カタログに格納されているメッセージにアクセスするために提供される関数は次のとおりです。

- catopen関数は，指定されたカタログをオープンして使用できるようにします。
- catgets関数は，プログラムがカタログから特定のメッセージを読み込むことができるようにします。

- catclose関数は，指定されたカタログをクローズします。オープンされているメッセージ・カタログは，exit関数によってもクローズされます。

メッセージ・カタログの作成の詳細については， OpenVMS システムのドキュメントの GENCAT コマンドの説明を参照してください。

10.7 異なる文字セットの取り扱い

HP C RTL では， ASCII でエンコーディングされた Portable Character Set を含む数多くの状態独立コードセットおよびコードセット・エンコード方式がサポートされます。状態依存コードセットはサポートされません。サポートされるコードセットは次のとおりです。

- ISO8859-n

  ただし，n = 1， 2， 5， 7， 8， 9 のいずれかです。これは北米，ヨーロッパ (西と東)，イスラエル，トルコのコードセットをカバーします。
- eucJP， SJIS， DECKANJI， SDECKANJI: 日本で使用されるコードセット
- eucTW， DECHANYU， BIG5， DECHANZI: 中国 (PRC)，香港，台湾で使用される中国語コードセット
- DECKOREAN: 韓国で使用されるコードセット

10.7.1 charmap ファイル

コードセット内の文字は charmap ファイルに定義されています。弊社が提供する charmap ファイルは， SYS$I18N_LOCALE 論理名によって定義されるディレクトリにあります。 charmap ファイルのファイル・タイプは.CMAP です。

10.7.2 コンバータ関数

HP C RTL では異なる文字セットをサポートするだけでなく，文字をあるコードセットから別のコードセットに変換するための次のコンバータ関数も提供しています。

- iconv_open — 変換のタイプを指定します。iconv関数で必要とされる変換記述子を割り当てます。
- iconv — ファイル内の文字を別のコードセットの対応する文字に変換します。変換された文字は別のファイルに格納されます。
- iconv_close — 変換記述子と，記述子に割り当てられているリソースの割り当てを解除します。

10.7.3 コードセット・コンバータ・ファイルの使用

コードセット・コンバータのファイル命名規則は次のとおりです。

```
fromcode_tocode.iconv
```

ただし，fromcodeはソース・コードセットの名前であり，tocodeは文字の変換先のコードセットの名前です。

コードセット・コンバータをシステムに追加するには， SYS$I18N_ICONV という論理名によって示されるディレクトリにコンバータ・ファイルをインストールします。

コードセット・コンバータ・ファイルは，テーブル・ベースの変換ファイルとして実装することができ， OpenVMS 共用可能イメージとして作成されたアルゴリズム・ベースのコンバータ・ファイルとして実装することもできます。

テーブル・ベースの変換ファイルの作成

テーブル・ベースのコードセット・コンバータ・ファイルを作成するには，次の手順を実行します。

1. ソース・コードセットの文字とターゲット・コードセットの文字の対応関係を記述したテキスト・ファイルを作成します。このファイルの形式については，OpenVMS New Features Manualの DCL コマンド ICONV COMPILE を参照してください。このコマンドはこのようなファイルを処理し，コードセット・コンバータ・テーブル・ファイルを作成します。
2. ステップ 1 で作成したファイルを論理名 SYS$I18N_ICONV によって示されるディレクトリにコピーします。この操作を実行するための特権が必要です。

アルゴリズム・ベースの変換ファイルの作成

共用可能イメージとして実装されたアルゴリズム・ベースのコードセット・コンバータ・ファイルを作成するには，次の操作を実行します。

1. コードセット・コンバータを実装する C ソース・ファイルを作成します。 API はパブリック・ヘッダ・ファイル<iconv.h>に次のように指定されています。
   - ユニバーサル・エントリ・ポイント_u_iconv_openがHP C RTL ルーチンiconv_openによって呼び出され，変換が初期化されます。
   - _u_iconv_openは，__iconv_extern_obj_t構造体を指すポインタであるiconv_openを返します。
   - この構造体の内部で，コンバータは独自の変換エントリ・ポイントと変換クローズ・ルーチンをエクスポートします。変換エントリ・ポイントはHP C RTL ルーチンiconvによって呼び出され，変換クローズ・ルーチンはiconv_closeによって呼び出されます。
   - ライブラリとコンバータの間に矛盾がないかどうかテストするために，iconv_openによってメジャー識別子フィールドとマイナー識別子フィールドが要求されます。コンバータは通常，<iconv.h>ヘッダ・ファイルに定義されている定数__ICONV_MAJOR と__ICONV_MINOR を割り当てます。

- フィールドtcs_mb_cur_maxは，バッファの使用を最適化するためにDCLコマンドICONV CONVERTによってのみ使用されます。このフィールドは，シフト・シーケンスも含めて(そのようなシーケンスがある場合)，ターゲット・コードセット内の1文字を構成する最大バイト数を反映します。

2. コードセット・コンバータを構成するモジュールをOpenVMSの共用可能イメージとしてコンパイルおよびリンクします。その場合，ファイル名が上記の規則に従っているかどうか確認されます。
3. 上記のステップで作成されたファイルをSYS$I18N_ICONVという論理名によって示されるディレクトリにコピーします。この操作を実行するための特権が必要です。

追加注意事項

デフォルト設定では， SYS$I18N_ICONVは検索リストであり，リストSYS$SYSROOT:[SYS$I18N.ICONV.USER]内の最初のディレクトリは，iconvコードセット・コンバータのサイト固有のリポジトリとして使用されます。

インストールされているコードセットとロケールの数はシステムごとに異なります。システムにイントスールされているコードセット，コンバータ，ロケールについては， SYS$I18Nディレクトリ・ツリーを確認してください。

10.8 カルチャー固有の情報の取り扱い

各ロケールには，次のカルチャー情報が格納されています。

- 日付および時刻情報

  LC_TIMEカテゴリは，日付と時刻，曜日，月の名前の表記方法を定義します。

- 数値情報

  LC_NUMERICカテゴリは，通貨以外の数値の書式を定義します。

- 通貨情報

  LC_MONETARYカテゴリは，通貨の書式を設定するために使用される通貨記号と規則を定義します。

- yesとnoに対応する応答

  LC_MESSAGESカテゴリは， yes/noを要求する質問に対する応答で期待される文字列を定義します。

このカルチャー情報は，nl_langinfo関数とlocaleconv関数を使用して部分的に取り出すことができます。第10.8.1項を参照してください。

10.8.1 ロケールからのカルチャー情報の抽出

nl_langinfo関数は，プログラムの現在のロケールから取得した情報項目を含む文字列を指すポインタを返します。ロケールから抽出できる情報は次のとおりです。

- 日付と時刻の書式
- ローカル言語での曜日の名前，月の名前
- 小数点を表す文字
- 通貨以外の数値で 3 桁ごとの区切り文字として使用される文字
- 通貨記号
- ロケールのコードセットの名前
- yes/no 質問の応答に対して定義されている文字列

localeconv関数は， LC_NUMERIC カテゴリと LC_MONETARY カテゴリから数値の書式データと通貨の書式データを格納した構造体を指すポインタを返します。

10.8.2 日付と時刻の書式関数

日付および時刻情報を使用する関数は次のとおりです。

- strftime — 構造体に格納されている日付と時刻の値を取得し，書式を設定して出力文字列を作成します。出力文字列の書式は書式設定文字列によって制御されます。
- strptime — 文字列 (char型) を日付と時刻の値に変換します。文字列の解釈方法は書式設定文字列によって定義されます。
- wcsftime — strftimeと同じ処理を実行しますが，ワイド文字の文字列を作成する点が異なります。

10.8.3 通貨書式設定関数

strfmon関数はロケール内の通貨情報を使用して，数値を文字列に変換します。文字列の書式は書式設定文字列によって制御されます。

10.8.4 数値の書式設定

LC_NUMERIC 内の情報はさまざまな関数で使用されます。たとえば，strtod，wcstod，およびプリント関数やスキャン関数は， LC_NUMERIC カテゴリから小数点文字を判断します。

10.9 ワイド文字を取り扱うための関数

文字はコードセットに応じて，シングル・バイトまたはマルチバイトの値で表現できます。シングル・バイト文字とマルチバイト文字をどちらも同じ方法で簡単に取り扱うことができるように，HP C RTL ではワイド文字データ型wchar_tを定義しています。このデータ型は，1 バイト，2 バイト，3 バイト，4 バイトのいずれかの値で表現される文字を格納できます。

ワイド文字をサポートするために提供される関数は次のとおりです。

- 文字分類関数。第 10.9.1 項を参照。
- 大文字/小文字変換関数。第 10.9.2 項を参照。
- 入出力関数。第 10.9.3 項を参照。
- マルチバイト文字からワイド文字への変換関数。第 10.9.4 項を参照。
- ワイド文字からマルチバイト文字への変換関数。第 10.9.4 項を参照。
- ワイド文字列操作関数。第 10.9.5 項を参照。
- ワイド文字の文字列照合および比較関数。第 10.10 節を参照。

10.9.1 文字分類関数

ロケール内の LC_CTYPE カテゴリは，ロケールのコードセットに含まれる文字を異なるタイプ (英字，数字，小文字，大文字など) に分類します。2 組の関数があり，1 組はワイド文字用，もう 1 組はシングル・バイト文字用で，文字が特定のタイプであるかどうかを判定します。is*関数はシングル・バイト文字を判定し，isw*関数はワイド文字を判定します。

たとえば，iswalnum関数は，ワイド文字が英字として分類されるのか，数字として分類されるのを判定します。文字がこれらのいずれかのタイプである場合は，0 以外の値を返します。分類関数の詳細については，第 3 章と「リファレンス・セクション」を参照してください。

10.9.2 大文字/小文字変換関数

LC_CTYPE カテゴリは，ロケールの 2 組の文字の間のマッピングを定義します。最も一般的な文字マッピングは大文字と小文字の間のマッピングです。しかし，ロケールでは大文字と小文字のマッピング以外のマッピングもサポートできます。

ロケールの LC_CTYPE カテゴリの情報に従って，ある文字を別の文字に変換するために 2 つの関数が用意されています。

- wctrans — 文字の間の指定されたマッピング (ロケールにあらかじめ定義されているマッピング) を検索します。

- towctrans — wctrans関数に指定されたマッピングに従って，ある文字を別の文字に変換します。

大文字と小文字の間のマッピングのために 2 つの関数が用意されています。

- towlower — 大文字のワイド文字を対応する小文字に変換します。
- towupper — 小文字のワイド文字を対応する大文字に変換します。

これらの関数の詳細については，「リファレンス・セクション」を参照してください。

10.9.3 ワイド文字の入出力のための関数

複数の入出力関数によって，ワイド文字およびワイド文字の文字列が管理されます。

読み込み関数
ワイド文字とワイド文字の文字列を読み込むための関数は，fgetwc，fgetws，getwc，getwcharです。

ワイド文字を入力ストリームに戻すためのungetwc関数も用意されています。

書き込み関数
ワイド文字およびワイド文字の文字列を書き込むための関数は，fputwc，fputws，putwc，putwcharです。

スキャン関数
スキャン関数はすべて，現在のロケールの LC_NUMERIC カテゴリに定義されているカルチャー固有の小数点文字を検索します。

%lc，%C，%ls，%S 変換指定子を使用すると，スキャン関数fwscanf，wscanf，swscanf，fscanf，scanf，sscanfはワイド文字を読み込むことができます。

プリント関数
すべてのプリント関数は，現在のロケールの LC_NUMERIC カテゴリに定義されているデータに従って，数値の書式を設定できます。

プリント関数で使用される%lc，%C，%ls，%S 変換指定子は，ワイド文字をマルチバイト文字に変換し，変換結果の文字をプリントします。

入出力関数の詳細については，第 2 章を参照してください。

10.9.4 マルチバイト文字とワイド文字の変換のための関数

ワイド文字は通常，シングル・バイト文字やマルチバイト文字をアプリケーションで管理するために内部的に使用されます。しかし，テキスト・ファイルは一般にマルチバイト文字形式で格納されます。これらのファイルを処理するには，マルチバイト文

字をワイド文字形式に変換する必要があります。この変換は次の関数を使用して行うことができます。

- mbtowc，mbrtowc，btowc — 1 文字のマルチバイト文字をワイド文字に変換します。
- mbsrtowcs，mbstowcs — マルチバイト文字の文字列をワイド文字の文字列に変換します。

同様に，次の関数はワイド文字を対応するマルチバイト文字に変換します。

- wcrtomb，wctomb，wctob — 1 文字のワイド文字をマルチバイト文字に変換します。
- wcsrtombs，wcstombs — ワイド文字の文字列をマルチバイト文字の文字列に変換します。

これらの変換関数に関連して，mblen関数とmbrlen関数はマルチバイト文字のサイズを判断するために使用されます。

複数のワイド文字関数は，「mbstate_tを指すポインタ」型の引数を受け付けます。ただし，mbstate_tは状態依存コードセットの変換状態を保持するために使用される opaque データ型です (FILEやfpos_tなど)。

10.9.5 ワイド文字の文字列および配列を操作するための関数

HP C RTL には，ワイド文字の文字列を操作する関数 (wcs*関数とwmem*関数) があります。たとえば，wcscat関数は，strcat関数がchar型の文字列に対して動作する方法と同じ方法で，ワイド文字の文字列を別の文字列の末尾に追加します。

文字列操作関数の詳細については，第 3 章を参照してください。

10.10 照合関数

国際化環境では，文字列比較関数はマルチパス照合ができなければなりません。照合する場合は次の要件を満たす必要があります。

- アクセント付き文字の並べ替えができること。
- 文字シーケンスを 1 文字として照合することができること。たとえば，スペイン語の ch は c と d の間の照合順序として判定されなければなりません。
- 1 文字を 2 文字シーケンスとして照合することができること。
- 一部の文字を無視することができること。

照合順序情報はロケールの LC_COLLATE カテゴリに格納されています。HP C RTL には，この照合順序情報を使用して 2 つの文字列を比較するために，strcoll関数とwcscoll関数が用意されています。

strcollまたはwcscollによるマルチパス照合は，strcmp関数やwcscmp関数を使用するより速度が遅くなる可能性があります。プログラムでstrcollまたはwcscollを使用して多くの文字列の比較を行う必要がある場合は，strxfrmまたはwcsxfrm関数を使用して文字列をいったん変換した後，strcmp関数やwcscmp関数を使用すると，処理速度を向上できます。

照合順序という用語は，文字の相対的な順序のことを示します。照合順序はロケール固有であり，一部の文字は無視されることがあります。たとえば，米語の辞書では単語の間のハイフンは無視され，take-outという用語はtakeoffとtakeoverの間にリストされます。

一方，**比較**という用語は，文字が同じであるのか，異なるのかを調べることを意味します。たとえば，takeoutとtake-outは照合順序が同じであっても，異なる単語です。

アプリケーションで単語のリストをソートした後，バイナリ検索を実行して単語を迅速に検索できるようにする場合について考えてみましょう。strcmp関数を使用すると，take-in，take-out，take-upはテーブルの 1 つの部分にまとめられます。strcollを使用し，ハイフンを無視するロケールを使用すると，take-outはtakeoffおよびtakeoverと同じグループにまとめられますが，takeoutと重複すると解釈されます。バイナリ検索でtakeoutがtake-outの複製として検索されるのを回避するには，アプリケーションでバイナリ・ツリーを作成するためにstrcollではなく，strcmpを使用する必要があります。

11

日付/時刻関数

この章では，HP C for OpenVMS Systems で提供される日付/時刻関数について説明します。各関数の詳細については，『HP C ランタイム・ライブラリ・リファレンス・マニュアル (下巻)』「リファレンス・セクション」を参照してください。

表 11–1 日付/時刻関数

関数	説明
asctime	localtimeから返された年月日時分秒形式の時刻を 26 文字の文字列に変換する。
ctime	1970 年1 月 1 日 00:00:00からの秒数で表した経過時間を，asctime関数で生成される形式の ASCII 文字列に変換する。
ftime	1970 年1 月 1 日 00:00:00からの経過時間を引数によって示される構造体に返す。
getclock	システム単位のクロックの現在の値を取得する。
gettimeofday	日付と時刻を取得する。
gmtime	時間単位を GMT (グリニッジ標準時) に変換する。
localtime	時間 (1970 年1 月 1 日 00:00:00からの秒数で表した経過時間) を時，分，秒などに変換する。
mktime	ローカル時刻構造体を暦の時刻値に変換する。
time	1970 年1 月 1 日 00:00:00から経過した時間を秒数で返す。
tzset	タイム・ゾーン変換を設定し，アクセスする。

また，fstatおよびstatから返される時刻関連情報では，第 11.1 節で説明している新しい日付/時刻モデルが使用されます。

11.1 日付/時刻のサポート・モデル

OpenVMS Version 7.0 以降，HP C RTL では日付/時刻サポート・モデルがローカル時刻ベースのモデルから UTC (協定世界時) ベースのモデルに変更されました。この結果，以前は実装することができなかったANSI C/POSIX機能をHP C RTL で実装することができるようになりました。 UTC 時刻ベースのモデルを導入したことで，HP C RTL は Tru64 UNIX の時刻関数の動作とも互換性を保持することができるようになりました。

デフォルト設定では，新たにコンパイルされるプログラムは UTC ベースの日付/時刻ルーチンへのエントリ・ポイントを生成します。

V7.0 より前のバージョンの OpenVMS システムとの互換性を維持するために，以前のバージョンでコンパイルされていて， OpenVMS Version 7.0 システムで再リンクされるプログラムは，ローカル時刻ベースの日付/時刻のサポートを保持します。再リンクだけでは UTC のサポートにアクセスできません。

_DECC_V4_SOURCE 機能テスト・マクロと_VMS_V6_SOURCE 機能テスト・マクロを定義してプログラムをコンパイルした場合も，ローカル時刻ベースのエントリ・ポイントが有効になります。つまり，新しい OpenVMS Version 7.0 の日付/時刻関数は有効になりません。

UTC ベースのエントリ・ポイントとローカル時刻ベースのエントリ・ポイントの両方を備えている関数は次のとおりです。

```
ctime           mktime
fstat           stat
ftime           strftime
gmtime          time
localtime       wcsftime
```

注意

UTC ベースの日付/時刻モデルが導入された結果，UTC をサポートする時刻関連関数は，ローカル時刻ベースの日付/時刻モデルの場合のようにメモリ内で単純な演算を行うのではなく，タイム・ゾーン・ファイルを読み込んで解釈する必要があるため，性能がある程度低下します。

この性能低下を緩和するために，OpenVMS Version 7.1 以降のバージョンでは，タイム・ゾーン・ファイルを格納したプロセス単位のキャッシュを保持することができるようになりました。キャッシュのサイズ (つまり，メモリ内のファイルの数) は，DECC$TZ_CACHE_SIZE 論理名の値によって決定されます。デフォルト値は 2 です。

タイム・ゾーン・ファイルはかなり小さいため (各ファイルは約 3 ブロック)，アプリケーションで使用するタイム・ゾーンの最大数になるように DECC$TZ_CACHE_SIZE を定義することを検討してください。たとえば，デフォルトのキャッシュ・サイズは，実行時にタイム・ゾーンを切り換えないアプリケーションに適しており，TZ環境変数が標準タイム・ゾーンと夏時刻タイム・ゾーンの両方で定義されているシステムで動作するアプリケーションに適しています。

11.2 日付/時刻関数の概要

UTC ベースのモデルでは，時刻は Epoch 以降の秒数として表現されます。 Epoch は UTC の 1970 年1 月 1 日 0 時，0 分，0 秒という時刻として定義されます。 Epoch 以降の秒数は，指定された時刻から Epoch までの秒数として解釈される値です。

time関数とftime関数は， Epoch からの秒数として時刻を返します。

ctime，gmtime，localtime関数は， Epoch からの秒数として時刻を表現する時刻値を引数として受け付けます。

mktime関数は，ローカル時刻として表現されている年月日時分秒形式の時刻を，Epoch からの秒数を表す時刻値に変換します。

stat関数とfstat関数からstat構造体に返される値st_ctime，st_atime，st_mtimeもUTC として表されます。

OpenVMS Version 7.0 で新たに導入された時刻サポートには，関数tzset，gettimeofday，getclockと，外部変数tzname，timezone，daylightが含まれています。

UTC ベースの時刻モデルを使用すると，HP C RTL で次のことが可能になります。

- ANSI Cのgmtime関数を実装できます。この関数は GMT 時刻として構造体を返します。
- tm構造体の ANSI tm_isdstフィールドを指定することができます。このフィールドは，夏時刻が有効かどうかを指定します。
- 時刻関連の POSIX および X/Open 拡張機能を提供できます (たとえば，tzset関数 (任意のタイム・ゾーンから時刻情報を取得する関数) や外部変数tzname，timezone，daylightなど)。
- 過去のローカル時刻を正確に計算することができます。この機能は，localtimeなどの時刻関数で必要になります。
- 機能テスト・マクロ (第 1.4 節を参照) を使用することにより，localtimeとgmtimeは， 2 つの追加フィールド，tm_zone (タイム・ゾーン名の省略形) とtm_gmtoff (秒数で表した UTC からのオフセット) を，これらの関数が返すtm構造体に返すことができます。

11.3 HP C RTL の日付/時刻の演算 —UTC 時刻とローカル時刻

UTC (協定世界時) は，時刻を表すための国際標準です。 UTC 標準時刻では， 0 時はグリニッジ標準時の午前 0 時に相当します。 UTC はローカル時刻と異なり，常に増大するという利点があります。ローカル時刻は夏時刻の設定に応じて進んだり，遅れたりする可能性があります。

また， UTC には次の 2 つの追加コンポーネントがあります。

- 誤差 (省略可能)。
- 時差係数。各ローカル・タイム・ゾーンを UTC に関連付けます。

  時差係数はローカル時刻を求めるために UTC に適用されます (ローカル時刻は，グリニッジ標準時の西側では最大 –12 時間，東側では +13 時間ずれる可能性があります)。

OpenVMS Version 7.0 以降のバージョンでHP C RTL 時刻サポートが正しく動作するには，次のことに注意する必要があります。

- 正しい OpenVMS TDF を使用するように， OpenVMS システムを正しく構成する必要があります。正しく設定されているかどうかは， SYS$TIMEZONE_DIFFERENTIAL 論理名の値を調べることにより確認できます。この論理名には，ローカル時刻を求めるために UTC に加算される時差が格納されています。
- OpenVMS のインストールで，デフォルトのローカル・タイム・ゾーンになる場所を記述するローカル・タイム・ゾーンを正しく設定する必要があります。一般に，これはシステムが動作しているローカル・タイム・ゾーンです。

詳細については，OpenVMS System Manager's Manual: Essentialsで，異なるタイム・ゾーンに対して補正するためのシステムの設定に関するセクションを参照してください。

HP C RTL では，次に示すように， UTC からローカル時刻を計算するために，ローカル・タイム・ゾーン変換規則を使用します。

1. HP C RTL は UTC 表現で時刻を内部的に計算します。
2. HP C RTL はタイム・ゾーン変換規則を使用して，ローカル時刻を求めるために UTC に適用される時差係数を計算します。タイム・ゾーン変換規則の詳細については，本書の「リファレンス・セクション」のtzset関数を参照してください。

デフォルト設定では， UTC からローカル時刻を計算するために使用されるタイム・ゾーン変換規則は， SYS$LOCALTIME および SYS$POSIXRULES システム論理名によって定義されるタイム・ゾーン・ファイルに指定されています。これらの論理名は OpenVMS のインストール時に，システムの時刻をローカル時刻に最適に近似するためのタイム・ゾーン・ファイルを示すように設定されます。

- SYS$LOCALTIME は，HP C RTL がローカル時刻を計算するために使用するデフォルトの変換規則を格納したタイム・ゾーン・ファイルを定義します。
- SYS$POSIXRULES は，夏時刻への変更および標準時刻への変更の時期が指定されていない POSIX 形式のタイム・ゾーンに適用されるデフォルトの規則を指定するタイム・ゾーン・ファイルを定義します。

SYS$POSIXRULES は SYS$LOCALTIME と同じでも構いません。詳細については，tzset関数を参照してください。

11.4 タイム・ゾーン変換規則ファイル

SYS$LOCALTIME 論理名と SYS$POSIXRULES 論理名によって示されるタイム・ゾーン・ファイルは， OpenVMS Version 7.0 以降のバージョンのシステムにインストールされるパブリック・ドメインのタイム・ゾーン・サポート・パッケージの一部です。

このサポート・パッケージには，全世界のタイム・ゾーンのローカル時刻を UTC から計算するためのタイム・ゾーン変換規則を記述した一連のソース・ファイルが含まれています。 OpenVMS Version 7.0 以降のバージョンのシステムでは， ZIC というタイム・ゾーン・コンパイラが提供されます。 ZIC コンパイラは，タイム・ゾーン・ソース・ファイルをバイナリ・ファイルにコンパイルします。HP C RTL はこのバイナリ・ファイルを読み込んで，タイム・ゾーン変換指定を取得します。これらのソース・ファイルの形式の詳細については， ZIC に関する OpenVMS システム・ドキュメンテーションを参照してください。

タイム・ゾーン・ファイルは次の編成になっています。

- ルート・タイム・ゾーン・ディレクトリは SYS$COMMON:[SYS$TIMEZONE.SYSTEM]です。 SYS$TZDIR というシステム論理名がインストール時にこの領域を示すように設定されます。
- タイム・ゾーン・ソース・ファイルは SYS$COMMON:[SYS$TIMEZONE.SYSTEM.SOURCES]にあります。
- バイナリ・タイム・ゾーン・ファイルは，ルート・ディレクトリとして SYS$COMMON:[SYS$TIMEZONE.SYSTEM]を使用します。一部のバイナリ・ファイルはこのディレクトリにありますが，他のファイルはそのサブディレクトリにあります。
- サブディレクトリにあるバイナリ・ファイルは，より大きな地理的領域の特定のタイム・ゾーンを表すタイム・ゾーン・ファイルです。たとえば，SYS$COMMON:[SYS$TIMEZONE.SYSTEM]には，米国用のサブディレクトリとカナダ用のサブディレクトリがあります。これらの各地域には複数のタイム・ゾーンがあるからです。米国内の各タイム・ゾーンは， United States サブディレクトリのタイム・ゾーン・ファイルによって表されます。カナダの各タイム・ゾーンは， Canada サブディレクトリ内のタイム・ゾーン・ファイルによって表されます。

複数のタイム・ゾーン・ファイルの名前は，それらのファイルが表す地域名の省略形から作成されています。表 11–2 はこれらの省略形を示しています。

表 11–2 タイム・ゾーン・ファイルのファイル名の省略形

タイム・ゾーンの省略形	説明
CET	中央ヨーロッパの時刻
EET	東ヨーロッパの時刻
Factory	タイム・ゾーンがないことを指定する
GB-Eire	英国/アイルランド

(次ページに続く)

表 11–2 (続き) タイム・ゾーン・ファイルのファイル名の省略形

タイム・ゾーンの省略形	説明
GMT	グリニッジ標準時
NZ	ニュージーランド
NZ-CHAT	ニュージーランド，チャタム諸島
MET	中央ヨーロッパの時刻
PRC	中華人民共和国
ROC	中華民国
ROK	大韓民国
SystemV	System V オペレーティング・システム固有
UCT	協定世界時
US	米国
UTC	協定世界時
Universal	協定世界時
W-SU	中央ヨーロッパの時刻
WET	西ヨーロッパの時刻

独自のタイム・ゾーン規則を定義し，実装するための機能が提供されています。詳細については， ZIC コンパイラに関する OpenVMS システムのドキュメンテーションと，『HP C ランタイム・ライブラリ・リファレンス・マニュアル (下巻)』「リファレンス・セクション」の`tzset`の説明を参照してください。

また， SYS$LOCALTIME および SYS$POSIXRULES システム論理名は，ユーザ指定タイム・ゾーンに再定義することができます。

11.5 日付/時刻の例

次の例と説明は，HP C RTL の時刻関数を使用して現在の時刻をプリントする方法を示しています。

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

main ()
{
  time_t t;

  t = time((time_t)0);
  printf ("The current time is: %s\n",asctime (localtime (&t)));
}
```

この例の説明:

1. `time`関数を呼び出して， Epoch からの秒数として現在の時刻 (UTC) を取得します。

2. この値をlocaltime関数に渡します。その関数は，tzsetによって指定された時刻変換情報を使用して， UTC からローカル時刻を計算するためにどのタイム・ゾーン変換規則を使用するのかを判断します。デフォルト設定では，これらの規則は SYS$LOCALTIME によって定義されるファイルに指定されています。

   a. 米国東部地区のユーザがローカル時刻を求めたいとすると，インストール時に SYS$LOCALTIME を SYS$COMMON:[SYS$ZONEINFO.US]EASTERN に設定されています。これは，米国東部のタイム・ゾーンの変換規則が格納されているタイム・ゾーン・ファイルです。

   b. ローカル時刻で夏時刻 (DST) が採用されている場合は，SYS$COMMON:[SYS$ZONEINFO.US]EASTERN は，−4 時間の時差係数を UTC に適用してローカル規則を求めなければならないことを示します。

      ローカル時刻が東部標準時 (EST) の場合は，SYS$COMMON:[SYS$ZONEINFO.US]EASTERN は，−5 時間の時差係数を UTC に適用してローカル時間を求めなければならないことを示します。

   c. HP C RTL は−4 (DST) または−5 (EST) を UTC に適用し，localtimeはtm構造体としてローカル時刻を返します。

3. このtm構造体をasctime関数に渡して，判読可能な形式でローカル時刻をプリントします。

12

シンボリック・リンクと POSIX パス名のサポート

V8.3 以降の OpenVMS では， Open Group 準拠のシンボリック・リンクと，POSIX 準拠のパス名がサポートされています。このサポートは， OpenVMS へ UNIX/LINUX のアプリケーションを移植する場合や， UNIX スタイルの開発環境を使用してアプリケーションを開発する場合に，その移植に伴っていた複雑さや開発コストの負担を軽減するために拡張されたものです。

OpenVMS では，シンボリック・リンクと POSIX パス名の処理をサポートするために，以下の機能を提供しています。

- C ランタイム・ライブラリ (C RTL) に， Open Group 準拠のシンボリック・リンク関数symlink，readlink，unlink，realpath，lchown，およびlstatが追加されています (第 12.3.3 項を参照)。
- creat，open，delete，およびremoveなどの既存の C RTL 関数が， Open Group のシンボリック・リンク仕様に従って動作するようになっています (第 12.3.4 項を参照)。
- RMS が拡張されて， C RTL に上記の関数が実装できるようになりました。SYS$OPEN， SYS$CREATE， SYS$PARSE，および SYS$SEARCH などの RMS ルーチンで，シンボリック・リンクをサポートできるようになっています (第 12.4 節を参照)。
- OpenVMS でファイルを検索したりそのパスをたどったりしているときにシンボリック・リンクが検出されると，その内容が POSIX のパス名として解釈されるようになりました。 OpenVMS の C RTL と RMS のインタフェースで POSIX のパス名が使用できるようになっています (第 12.1.2 項を参照)。
- C RTL に新しい機能論理名 DECC$POSIX_COMPLIANT_PATHNAMES が追加されて，アプリケーションが POSIX 準拠モードで動作することを指示できるようになっています。ただし， POSIX 準拠モードで参照できるのは，最新バージョンのファイルだけです。同じファイルの複数のバージョンにアクセスすることはできません (第 12.3.1 項を参照)。
- シンボリック・リンクの作成には， DCL コマンド CREATE/SYMLINK を使用します (第 12.2.1 項を参照)。
- システムの POSIX ルートを作成するには， DCL コマンド SET ROOT を使用します (第 12.5 節を参照)。
- マウント・ポイントを設定するために， 2 つの GNV ユーティリティmntとumntが用意されています (第 12.7 節を参照)。

- DCL コマンドおよびユーティリティが変更されて，シンボリック・リンクの操作や，シンボリック・リンクを検出した場合の処理が適切に行えるようになっています (第 12.9 節を参照)。
- TCP/IP Services for for OpenVMS Network File System (NFS) のクライアントとサーバが拡張されて， ODS5 ボリューム上でシンボリック・リンクをサポートできるようになっています。
- ln (シンボリック・リンクの作成) や ls (シンボリック・リンクの内容の表示) などの関連 GNV ユーティリティがアップデートされて，シンボリック・リンクのアクセスと管理が行えるようになっています (12.2.2 と 12.10 を参照)。

12.1 POSIX パス名とファイル名

POSIX のパス名は， 0 個以上のファイル名が/ (パス区切り文字) で区切られた，空でない文字列から構成されています。使用するパス区切り文字は，パス名の先頭や末尾に置くことも，ファイル名とファイル名の間に置くこともできます。また， 2 個以上の区切り文字を連続して置くこともできます。ただし，どの区切り文字も 1 文字として扱われます。

POSIX のファイル名には，/ (パス区切り文字) と NUL (多くの API で文字列の終了文字として使用される) を除く，すべての ASCII 文字が使用できます (ただし，現在の OpenVMS システムには， POSIX ファイル名に?文字と *文字も含めてはならないという制限があります)。ファイル名は，**パス名の構成要素**または**パス名のコンポーネント**とも呼ばれます。

12.1.1 POSIX パス名の解釈

先頭が/で始まっているパス名は，絶対パス名として扱われ，その処理は， POSIX **ルート**として指定されているデバイスおよびディレクトリから開始されます (ルートの定義方法は，第 12.5 節を参照してください)。

先頭が/で始まっていないパス名は相対パス名として扱われ，その処理は現在の作業ディレクトリから開始されます。

OpenVMS で現在の作業ディレクトリがどのように解釈されるかについては，第 12.6 節を参照してください。

12.1.1.1 POSIX ルート・ディレクトリ

POSIX **ルート**・ディレクトリは，絶対パス名を解決する際に使用されるディレクトリです。

ルート・ディレクトリは，ディレクトリ階層の一番上にあるノードで，自分自身が自分の親ディレクトリにもなっています。つまり，ルート・ディレクトリのパス名は/で，その親ディレクトリのパス名も/です。

OpenVMS における POSIX ルートの定義については，第 12.5 節を参照してください。

12.1.1.2 シンボリック・リンク

シンボリック・リンクは，他のファイルを指す特殊なファイルです。シンボリック・リンクはディレクトリの 1 つのエントリになっていて，サービスがアクセスするときに POSIX パス名として解釈するテキスト文字列をあるファイルに対応付ける機能があります。シンボリック・リンクへアクセスする操作のほとんどは，そのシンボリック・リンク自体に対してではなく，そのテキスト内容が参照している実際のオブジェクトに対して行われます。したがって，実際のオブジェクトが存在していないと，その操作は失敗します。 OpenVMS におけるシンボリック・リンクは，編成が SPECIAL でタイプが SYMBOLIC_LINK のファイルとして実装されています。

シンボリック・リンクでは，絶対パス名だけでなく相対パス名を指定することもできます。たとえば，内容が ".."というパスで始まるシンボリック・リンクは，その親ディレクトリを参照することになります。また，このシンボリック・リンクを別のディレクトリへ移動すれば，その新しいディレクトリの親ディレクトリを参照することになります。

OpenVMS におけるシンボリック・リンクの使用方法については，第 12.2 節を参照してください。

12.1.1.3 マウント・ポイント

マウント・ポイントとは，ファイル・システムが接続されている位置 (実際にはディレクトリ) のことです。 OpenVMS におけるディスク・ボリュームという概念と，UNIX の**ファイル・システム**という用語は，本質的に同じものを表しています。マウント・ポイントを確立すると，マウント・ポイントになる前の元のディレクトリにあった内容へはアクセスできなくなります。

システムをリブートすると，リブートの前に確立されていたマウント・ポイントは無効になります。

OpenVMS でマウント・ポイントを確立する方法については，第 12.7 節を参照してください。

12.1.1.4 予約ファイル名「.」と「..」

次に示すように， POSIX には予約済みのファイル名が 2 つあります。

. (現在のディレクトリ)

.. (親ディレクトリ)

12.1.1.5 文字型特殊ファイル

UNIX システムでは，コンピュータ・システムの特定のハードウェア・デバイスやリソースに**特殊ファイル**を関連づけており，オペレーティング・システムは，デバイス・ファイルとも呼ぶこの文字型特殊ファイルを使用することで，特定の文字型デバイスに対する入出力を行えるようにしています。

特殊ファイルは，一見しただけでは，通常のファイルと同じように見えます。たとえば次のとおりです。

- ディレクトリの下にそのパス名がある。
- 通常ファイルと同じアクセス保護が可能である。
- 通常ファイルの場合とほとんど同じ方法で使用できる。

しかし，この 2 つのファイルには，大きな違いがあります。つまり，通常ファイルはディスクに記録されているデータを論理的にグループ化したものですが，特殊ファイルはデバイス・エンティティに対応したものです。特殊ファイルの例を次に示します。

- 実際のデバイス (ライン・プリンタなど)。
- 論理サブ・デバイス (ディスク・ドライブ内の大規模セクションなど)。
- 擬似デバイス (コンピュータの物理メモリ (/dev/mem)，ターミナル・ファイル (/dev/tty)，ヌル・ファイル (/dev/null) など)。

サポートされている特殊ファイルは，ヌル特殊ファイル (/dev/null) だけです。このファイルは， GNV 構成処理が実行されている間に作成されます。ヌル特殊ファイルに書き込んだデータは，廃棄されます。また，ヌル特殊ファイルに対して読み取りを実行しても，返ってくるデータは常に 0 バイトです。

12.1.2 OpenVMS インタフェースでの POSIX パス名の使用

OpenVMS のほとんどのアプリケーション，ユーティリティ，および API には，POSIX 準拠の仕様でパス名を渡すことができます。

この仕様には DCL の既存の表記法が採用されていて，引用符で囲んだ文字列の中にさらに引用符が含まれている場合は，その含まれている引用符を 2 回繰り返して指定します。

また，パス名に対して，引用符で囲んだ文字列の使用が将来制約とならないようにしておく場合は (たとえば, 他の構文バリエーションとして Windows と互換性のある名前のサポートを考えている場合など)，引用符で囲んだパス名の先頭に先行タグ^UP^を付加して， DCL， RMS，および OpenVMS のユーティリティでそれが POSIX パス名であることが分かるようにしておく必要があります。ただし， C RTL や GNV に対しては，この要件は必要**ありません**。

以上の要件を守ると，パス名/a/b/cは "^UP^/a/b/c"で，また，/a/b"/cは "^UP^/a/b""/c"でそれぞれ指定することになります。

この章の残りの部分では，「POSIX **パス名**」という用語で変更前のパス名 (/a/b/cなど) を示し，「**引用符で囲んだパス名**」という用語で，引用符で囲んだ先行タグ付きのパス名を示します ("^UP^/a/b/c"など)。

12.1.2.1 POSIX のファイル名に関する特別な考慮事項

OpenVMS で作成するファイルに POSIX ファイル名を使用する場合は，その POSIX ファイル名と OpenVMS のファイル名が同じように表示されると理想的です。たとえば，作成するファイルの名前として POSIX ファイル名a.bを指定した場合に，OpenVMS 側でa.b; (;はファイル・バージョン) というファイルが自動的に作成される，という具合にです。

しかし， OpenVMS のファイル名にはその構成要素としてファイル・タイプが必要なので (少なくともピリオドが 1 文字必要)，ピリオドのない POSIX ファイル名を直接マッピングすることはできません (たとえば， POSIX ファイル名aを OpenVMS のファイル名 a;にマッピングするというようなことはできません)。 OpenVMS にとって妥当な解決方法は，ピリオドを付加して (つまりファイル・タイプ・フィールドがヌルであることを示して)， POSIX ファイル名aに対応する OpenVMS のファイル名をa.;にすることです。

ただし，この場合に注意すべきことは， POSIX ファイル名の末尾がピリオドで終わっていても， OpenVMS ではピリオド (つまり，ヌル・タイプ) を付加するということです。これは，a.という POSIX ファイル名と aという POSIX ファイル名が同時に存在している場合に，それらを区別する必要があるからです。

また， OpenVMS のディレクトリには，.DIR;1というタイプとバージョンが割り当てられています。そのため， POSIX ファイル名がaのディレクトリを OpenVMS で作成しようとすると， OpenVMS ファイル名がa.DIR;1というディレクトリが，作成されます。しかし，a.DIRは POSIX ファイル名としても有効なので，もしそのような POSIX ファイル名のファイルが同じディレクトリに存在すれば，そのファイルと，aという POSIX ファイル名のディレクトリを区別できるようにする必要があります。そのため， OpenVMS では，末尾が.DIRで終わる POSIX ファイル名にピリオド (つまり，ヌル・タイプ) を付加するようになっています。その結果， POSIX ファイル名a.DIRは a.DIR.;になります。

上記の規則に従ったファイル名のマッピング例を以下に示します。次に示すのは，OpenVMS で変更されない (バージョンの追加を除く) POSIX ファイル名の例です。

POSIX ファイル名	OpenVMS 名	タイプ	バージョン	DIR での表示
a.b	a	.b	;	a.b;
a.b;	a	.b;	;	a.b;;
a.b;2	a	.b;2	;	a.b;2;

次に示すのは，末尾がピリオド (.) または.DIRで終わっているために， OpenVMS でファイル名を作成するときにピリオドを付加する POSIX ファイル名の例です。

```
POSIX ファイル名   OpenVMS 名   タイプ   バージョン   DIR での表示
---------------   ----------   ------   ----------   ------------
     a               a           .         ;           a.;
     a.              a.          .         ;           a^..;
     a..             a..         .         ;           a^.^..;
     a.b.            a.b.        .         ;           a^.b^..;
     a.DIR           a.DIR       .         ;           a^.DIR.;
```

最後に示すのは， POSIX ディレクトリの例で， OpenVMS ではそのすべての終わりに .DIR;1を付加しています。

```
POSIX ファイル名   OpenVMS 名   タイプ   バージョン   DIR での表示
---------------   ----------   ------   ----------   ------------
     a               a          .DIR       ;1          a.DIR;1
     a.dir           a.dir      .DIR       ;1          a^.dir.DIR;1
     a.              a.         .DIR       ;1          a^..DIR;1
```

POSIX ファイル名に*または?文字が含まれている場合は，マッピングできません。この問題は未解決で，将来の OpenVMS リリースで対処される予定です。

12.1.2.2 OpenVMS のファイル名に関する特別な考慮事項

OpenVMS のファイル名を POSIX ファイル名にマッピングするときは (C RTL の関数readdirの機能)，次の規則が適用されます。

OpenVMS ファイル名の末尾が.DIR;1で終わっていて，そのファイルのタイプがディレクトリである場合は，.DIR;1が削除されます。それ以外のファイルは， OpenVMS のファイル名から末尾のセミコロンとバージョン番号が削除されます。

OpenVMS ファイル名の末尾がピリオドで終わっている場合は，そのピリオドが削除されます。

OpenVMS のファイル名に/または NUL 文字が含まれている場合はマッピングできないので，呼び出し側へは，対応するエントリが返されません。

OpenVMS のディレクトリ..DIR;1と ...DIR;1は，対応する POSIX ファイル名が特別なファイル名 .と..になるので，マッピングできません。

OpenVMS のファイル.は，結果として作成される POSIX ファイル名が NULL 文字列になるので，マッピングできません。

12.2 シンボリック・リンクの使用

シンボリック・リンクの作成には，次の方法があります。

- DCL コマンド CREATE/SYMLINK を使用する (第 12.2.1 項)。

- GNV bash シェル内でln -sユーティリティを使用する (第 12.2.2 項)。
- C RTL のsymlink関数を使用する (第 12.3.3 項)。

12.2.1 DCL によるシンボリック・リンクの作成と使用

DCL コマンドによるシンボリック・リンクの作成方法とアクセス方法の例を，次に示します。

例

1.
```
$ create/symlink="a/b.txt" link_to_b.txt
$
$ dir/date link_to_b.txt

Directory DKB0:[TEST]

link_to_b.txt -> a/b.txt
                 27-MAY-2005 09:45:15.20
```

この例では，シンボリック・リンク・ファイルlink_to_b.txtを作成していて，その内容は，指定したテキスト文字列，つまり，シンボリック・リンクが指すパス名a/b.txtと同じものになっています。a/b.txtファイルは，存在していてもいなくてもかまいません。

2.
```
$ type link_to_b.txt
  %TYPE-W-OPENIN, error opening DKB0:[TEST]LINK_TO_B.TXT; as input
 -RMS-E-FNF, file not found
```

この例では，シンボリック・リンクで参照しているファイルへアクセスするために，そのシンボリック・リンク名をコマンドで指定しています。ここではTYPEコマンドに対してエラー・メッセージが出されて，最初の例で作成したシンボリック・リンクに，対応する参照ファイルa/b.txtが存在していないことが示されています。

3.
```
  $ create [.a]b.txt
     This is a text file.
  Exit
  $
  $ type link_to_b.txt
  This is a text file
  $
```

この例では，存在していなかったファイルb.txtを，リンクを通して作成しています。また，その後，シンボリック・リンクを通してTYPEコマンドを実行していますが，今回は参照するファイルが存在しているので，成功しています。

このファイルは，次のようにシンボリック・リンクを通して作成することもできます。

```
$ create link_to_b.txt
   This is a text file.
   Exit
$ dir [.a]

Directory DKB0:[TEST.A]

b.txt;1

$
$ type link_to_b.txt
   This is a text file.
$
```

12.2.2 GNV POSIX コマンドと DCL コマンドによるシンボリック・リンクの使用

GNV bash シェルでは，lnユーティリティを次のように使用して，シンボリック・リンクを作成することができます。

```
bash$ ln -s filename slinkname
```

slinknameを参照するユーティリティとコマンドのほとんどは，filenameに対して動作するようにリダイレクトされます。しかし，lsなどの一部のコマンドは，それがシンボリック・リンクであることを認識して，そのリンク自体の情報を表示します (その内容など)。

リンクは，rmコマンドで削除できます (ただし，リンクをたどって，それが指しているもの自体を削除することはできません)。

以下の例では， GNV の POSIX コマンドでシンボリック・リンクを管理する方法と， DCL コマンドで管理する方法を比較して示します。

例

1. GNV の場合:

```
bash$ cd /symlink_example
bash$ echo This is a test. > text
bash$ cat text
This is a test.
bash$ ln -s text LINKTOTEXT
bash$
```

   DCL の場合:

```
$ set def DISK$XALR:[PSX$ROOT.symlink_example]
$ create text.
This is a test.
Exit
$ type text.
This is a test.
$ create/symlink="text" LINKTOTEXT
```

   この例では，テキスト・ファイルtextを指す LINKTOTEXT というシンボリック・リンクを作成しています。

2.

GNV の場合:

```
bash$ cat LINKTOTEXT
This is a test.
bash$
```

DCL の場合:

```
$ type LINKTOTEXT.
This is a test.
$
```

この例では，シンボリック・リンクを指定して，そのテキスト・ファイルの内容を表示しています。

3.

GNV の場合:

```
bash$ ls
LINKTOTEXT  text
bash$ ls -l
total 1
lrwxr-x---   1 SYSTEM   1        4 May 12 08:41 LINKTOTEXT -> text
-rw-r-----   1 SYSTEM   1       16 May 12 08:40 text
bash$
```

DCL の場合:

```
$ DIR

Directory DISK$XALR:[PSX$ROOT.symlink_example]

LINKTOTEXT.;1       text.;1

Total of 2 files.

$ DIR/DATE

Directory DISK$XALR:[PSX$ROOT.symlink_example]

LINKTOTEXT.;1 -> /symlink_example/text
                12-MAY-2005 08:46:31.40
text.;1         12-MAY-2005 08:43:47.53

Total of 2 files.
$
```

この例では，シンボリック・リンクの内容を表示しています。ここで注目して欲しいのは，lsまたは DIR の実行でファイル属性の表示スイッチを指定すると，シンボリック・リンクの内容も表示されるということです。

4.

GNV の場合:

```
bash$ rm text
bash$ cat LINKTOTEXT
cat: linktotext: i/o error
bash$ rm LINKTOTEXT
```

DCL の場合:

```
$ DEL text.;1
$ TYPE LINKTOTEXT.
%TYPE-W-OPENIN, error opening
DISK$XALR:[PSX$ROOT.symlink_example]LINKTOTEXT.;1 as input
-RMS-E-ACC, ACP file access failed
-SYSTEM-F-FILNOTACC, file not accessed on channel

$ DEL LINKTOTEXT.;1
$
```

この例では，ファイルを削除しています。

12.3 C RTL でのサポート

この節では， C RTL で行われている POSIX 準拠のパス名とシンボリック・リンクのサポートについて説明します。

12.3.1 DECC$POSIX_COMPLIANT_PATHNAMES 機能論理名

POSIX 準拠のユーザに対して，シンボリック・リンクへ格納するパス名と C RTL 関数へ入力するパス名との間に一貫性を持たせることは，非常に重要です。そのために C RTL では，アプリケーションから C RTL 関数に対して POSIX 準拠のパス名を渡せるようにするための機能論理名， DECC$POSIX_COMPLIANT_PATHNAMES が用意されています。

DECC$POSIX_COMPLIANT_PATHNAMES は特に指定しない限り無効になっていて，通常の C RTL 動作が優先されます。その動作の中には UNIX 仕様のパス名を解釈する方法も含まれていて， POSIX 準拠のパス名処理とは無関係な，異なる規則が使用されます。

DECC$POSIX_COMPLIANT_PATHNAMES を無効にする必要がある場合は，次のように DEFINE を使用して，その値として 0 または "DISABLE"を設定します。

```
$ DEFINE DECC$POSIX_COMPLIANT_PATHNAMES 0
```

DECC$POSIX_COMPLIANT_PATHNAMES を有効にする場合は，その値として，次のいずれかを設定します。

1 POSIX のみ - パス名をすべて， POSIX のスタイルで表す。
2 POSIX 寄り - ";"で終わっているかまたは任意のかっこ "[ ]<>"を含んでいて，しかも SYS$FILESCAN サービスで解析可能なパス名は，そのすべてを OpenVMS のスタイルで表し，それ以外のパス名は， POSIX のスタイルで表す。
3 OpenVMS 寄り - "/"を含むパス名と，パス名 "."および ".."は， POSIX のスタイルで表し，それ以外のパス名は， OpenVMS のスタイルで表す。

4　OpenVMS のみ - パス名をすべて，OpenVMS のスタイルで表す。

注意

モード 1 および 4 は，他の共用ライブラリやユーティリティとの間で相互作用があるので，お勧めしません。

DECC$POSIX_COMPLIANT_PATHNAMES を有効にすると，C RTL では，規則 (つまり DECC$POSIX_COMPLIANT_PATHNAMES に割り当てた値で決まる規則) に従ってパス名を調べ，それが POSIX のスタイルで表されているのか，OpenVMS のスタイルで表されているのかを判断します。

たとえば，パス名のほとんどを POSIX 準拠で表す場合は，次のように設定します。

```
$ DEFINE DECC$POSIX_COMPLIANT_PATHNAMES 2
```

C RTL がパス名を POSIX スタイルで表すようになっている場合，C RTL では，受け取った POSIX パス名を OpenVMS のファイル指定へ変換しないで，引用符で囲んだ形式 (この章の前半で説明) に変換します。その結果，RMS ではそのパス名を，シンボリック・リンクで指定されたパス名と同じように処理することができます。

注意

以前の OpenVMS **バージョンではサポートされていませんでしたが，**OpenVMS Version 8.4 **以降では，**DECC$POSIX_COMPLIANT_PATHNAMES **が定義されている場合も論理名とデバイス名がサポートされます。**

DECC$POSIX_COMPLIANT_PATHNAMES を有効にすると，次の C RTL 機能論理名は無視されます。

DECC$ARGV_PARSE_STYLE
DECC$DISABLE_POSIX_ROOT
DECC$DISABLE_TO_VMS_LOGNAME_TRANSLATION
DECC$EFS_CASE_PRESERVE
DECC$EFS_CASE_SPECIAL
DECC$EFS_CHARSET
DECC$EFS_NO_DOTS_IN_DIRNAME
DECC$ENABLE_TO_VMS_LOGNAME_CACHE
DECC$FILENAME_UNIX_NO_VERSION
DECC$FILENAME_UNIX_ONLY
DECC$FILENAME_UNIX_REPORT
DECC$NO_ROOTED_SEARCH_LISTS
DECC$READDIR_DROPDOTNOTYPE
DECC$READDIR_KEEPDOTDIR
DECC$RENAME_ALLOW_DIR

DECC$RENAME_NO_INHERIT
DECC$UNIX_PATH_BEFORE_LOGNAME

12.3.2 decc$to_vms，decc$from_vms，および decc$translate_vms

POSIX 準拠モードで動作している場合，C RTL 関数decc$to_vmsでは，POSIX パス名を，引用符で囲まれた RMS のパス名へ変換します。逆に，decc$from_vms関数とdecc$translate_vms関数では，引用符で囲まれたパス名から引用符と接頭辞^UP^を削除して POSIX パス名に変更します (たとえば，"^UP^a/b"はa/bになります)。POSIX 準拠モードでなければ，これらの関数は以前と同じように動作します。

12.3.3 シンボリック・リンク関数

表 12–1 に，HP C Run-Time Library (RTL) のシンボリック・リンク関数とその説明を示します。各関数の詳細については，「リファレンス・セクション」を参照してください。

表 12–1 シンボリック・リンク関数

関数	説明
lchown	ファイルのオーナとグループを変更します。ファイルがシンボリック・リンクである場合は，そのシンボリック・リンクのオーナを変更します (chownは，シンボリック・リンクの指しているファイルを変更します)。
lstat	指定したファイルの情報を取得します。ファイルがシンボリック・リンクである場合は，そのリンク自体の情報が返されます (statは，シンボリック・リンクが指しているファイルの情報を返します)。
readlink	シンボリック・リンクの内容を読み取って，ユーザの用意したバッファに置きます。
realpath	POSIX ルートから始まる絶対パス名が返されます。この関数は，POSIX 準拠モード (つまり，DECC$POSIX_COMPLIANT_PATHNAMES が有効な場合) でだけサポートされています。
unlink	システムからシンボリック・リンクを削除します。
symlink	指定した内容を持つシンボリック・リンクを作成します。

12.3.4 既存関数の変更

今までに説明した新しい関数の追加に加えて，次の C RTL 関数が，シンボリック・リンクをサポートするために変更されています。

creat

ファイルの新しいバージョンを作成するときに同じ名前のシンボリック・リンクがすでに存在していると，そのシンボリック・リンクが参照しているファイルを作成するように変更されています。

open

open関数は，file_specパラメータで指定したファイルがシンボリック・リンクであると，そのシンボリック・リンクが指しているファイルをオープンするように変更されています。

delete，remove

deleteとremove関数は，指定したファイルがシンボリック・リンクであると，そのシンボリック・リンクの参照しているファイルではなく，リンク自体を削除するように変更されています。

また，シンボリック・リンクの解決でループが見つかったときは， Open Group の仕様に従ってerrno値 ELOOP が返されます。

12.3.5 POSIX に準拠していない動作

12.3.5.1 同じファイルにバージョンが複数個ある場合の動作

POSIX 準拠のパス名を使用している場合は，対象となるファイルに複数のバージョンが存在していても，アクセスできるのは最新バージョンのファイルだけです。また，ファイルを削除しても，実際に削除されるのは，バージョン番号の最も高いファイルだけです。

12.3.5.2 ファイルの名前があいまいな場合の動作

同じディレクトリにファイル"a"とディレクトリ "a.DIR;1"が存在している場合にPOSIX 準拠モードでアクセスすると，次のように処理されます。

- 入力としてディレクトリの名前しか受け付けない関数 (chdirやopendirなど) からは，エラー (ENOEXIST) が返されます。
- 入力としてディレクトリでないファイルしか受け付けない関数 (openなど) では，ディレクトリでない方のファイルを対象にして処理が行われます。
- 入力としてファイルもディレクトリも受け付ける関数 (statなど) では，ディレクトリでない方のファイルを対象にして処理が行われます。
- readdirからは，ファイルのエントリとディレクトリのエントリが両方とも返されます。

12.4 RMS インタフェース

12.4.1 POSIX パス名と RMS の入出力

ここで使用している「引用符で囲んだパス名」とは，この章の前半で説明したように，先頭にタグ文字列^UP^を付けて，さらにその全体を引用符で囲んだ POSIX パス名のことです。また，そのような POSIX パス名の中に引用符がある場合は，その引用符を 2 個続けることで，引用符で囲まれた文字列全体が DCL の場合と同じように扱われるようにする必要があります。

引用符で囲んだパス名は，プライマリ名，デフォルト名，および関連名で使用することができます。 RMS では，^UP^接頭辞と開始/終了引用符を検出すると，残りの文字を標準の POSIX 名として解釈します (ただし，パス名の中に引用符が 2 つ連続してある場合は，そのペアを 1 つの引用符として扱います)。また (「引用符で囲んだパス名」にワイルドカードが使用されている場合などは), 「引用符で囲んだパス名」を展開して得られたファイル指定も，引用符で囲んだパス名として返します。

返された「引用符で囲んだパス名」の構成要素は， NAM または NAML で次のように参照されます。

- nodeは NULL である。
- devは "^UP^である。
- dirは， dev 文字列と最後のスラッシュ(/)との間にあるすべての文字と最後のスラッシュから構成される。
- nameは， dir の直後から最後のピリオド(.)までの間にあるすべての文字から構成される (最後のピリオドがない場合は，dirの直後から終了二重引用符(")までの間にあるすべての文字から構成される)。
- typeは，nameの直後から終了二重引用符(")までの間にあるすべての文字から構成される。
- versionは二重引用符(")である。

dirフィールド，nameフィールド，およびtypeフィールドは NUL であっても構いません。

たとえば，引用符で囲んだパス名"^UP^/a/b.c"は，次の要素から構成されています。

nodeは NUL。
devは "^UP^。
dirは/a/。
nameは b。
typeは.c。
versionは "。

シンボリック・リンクは，次の RMS ルーチンでサポートされています。

- SYS$OPEN — シンボリック・リンクが指しているファイルを対象にして動作します。
- SYS$CREATE — シンボリック・リンクが指しているファイルを作成します。
- SYS$PARSE — ファイル指定文字列を解析して，さまざまな NAM ブロック・フィールドを埋めます。
- SYS$SEARCH — ターゲット・ファイルの DVI と FID を返します。 DID はゼロです。結果として得られる名前は，ターゲット・ファイルの名前ではなく，シンボリック・リンクの名前です。

NAML$V_OPEN_SPECIAL を設定すると，SYS$OPEN と SYS$SEARCH はシンボリック・リンクをたどらなくなります。

RMS で行う POSIX パス名の処理には，次の制約があります。

- 引用符で囲んだパス名でプライマリ・ファイルを指定した場合，SYS$SEARCH から返されるファイルは 1 つだけです。また，パス名には，ワイルドカードとして定義されている文字は使用できません。
- 引用符で囲んだパス名でプライマリ・ファイルを指定すると，デフォルトの指定と，関連する指定が無視されます。
- OpenVMS の従来のファイル指定方法でプライマリ・ファイルを指定する場合は，デフォルトの指定に，引用符で囲んだパス名を使用できます。ただし，そのデフォルトの指定から使用されるのは，ファイル名とタイプだけです。

12.4.2 アプリケーションから制御可能な RMS のシンボリック・リンク処理

RMS が一般的なファイル操作でシンボリック・リンクに出会ったときの動作は，DCL コマンドやユーティリティなどのアプリケーションで制御できるようにする必要があります。その中でも特に必要なのは，シンボリック・リンクに格納されているパス名を，パス名の処理とディレクトリの検索に含めるかどうかの制御です。また，シンボリック・リンクに出会ったときに，その状況に応じて異なる動作を指定できる機能も必要です。

こうしたアプリケーションに必要な柔軟性を実現するために NAML$L_INPUT_FLAGS フラグが新たに定義されて，次のような制御が行えるようになっています。

- SYS$OPEN のデフォルト動作では，渡されたパス名がシンボリック・リンクであると，そのシンボリック・リンクに格納されているパス名で指定されたファイルがオープンされます。 NAML$V_OPEN_SPECIAL フラグを設定しておくと，そのシンボリック・リンク・ファイル自体がオープンされます。
- NAML$V_OPEN_SPECIAL フラグを設定しておくと， SYS$SEARCH を呼び出して得られる結果がシンボリック・リンクである場合に，そのシンボリック・リンク自体の DVI と FID が返されます。それ以外の場合は，デフォルトの動作が実行されます。つまり，そのシンボリック・リンクに格納されているパス名で指定されたファイルの DVI と FID が返されます。ただし，そのパス名もシンボリック・リンクになっている場合は，シンボリック・リンクでないパス名が見つかるまで処理が繰り返された後， DVI と FID が返されます。
- SYS$SEARCH でワイルドカードによるディレクトリ検索を行う場合は，その過程でシンボリック・リンクを検出しても，それがディレクトリを参照しているかどうかを調べません。

また，ODS-5 ボリュームでは，RMS 操作でシンボリック・リンクに従うかどうかを指定するのに SPECIAL_FILES フラグが使用されます。シンボリック・リンクに従う RMS 操作としては，SYS$OPEN，SYS$CREATE，SYS$SEARCH，およびすべてのディレクトリ・パスの解釈があります。

SPECIAL_FILES フラグの設定方法については，『HP OpenVMS System Services Reference Manual』および『HP OpenVMS DCL ディクショナリ』を参照してください。

12.5 POSIX ルートの定義

RMS では，POSIX 仕様の絶対パス名にある先頭の '/' を，UNIX スタイルのルートとして解釈する必要があります。POSIX の仕様は，この POSIX ルートからすべてが始まるものとして決められています。POSIX ルートは，GNV のインストール・プロシージャや，新しい DCL コマンド SET ROOT で設定できます。

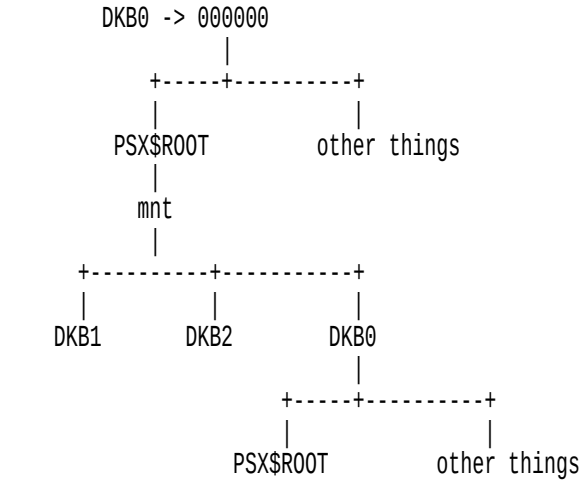
12.5.1 POSIX ルートの推奨位置

システムに設定する POSIX ルートの位置を指定する場合は，GNV のインストール/構成プロシージャを実行しているときに指定するか，または，SET ROOT コマンドを使用して指定する必要があります。特に指定しないで GNV をインストールすると，SYS$SYSDEVICE:[PSX$ROOT]がルートの位置として設定されます。このディレクトリをルートとして受け入れてもかまいませんが，別のディレクトリを指定することもできます。GNV では，ルートを設定した後，オプションで "/mnt" (POSIX ルート・ディレクトリにある "mnt" ディレクトリ) に，そのシステムに接続されている各ディスク・デバイスのマウント・ポイントを作成します。

ルートの位置としてデフォルトの値を受け入れるか，デフォルト以外の位置を指定してそのディレクトリがデバイスの最上位レベルにないと，ルート・ディレクトリ以下のディレクトリ・ツリーにないファイルは，そのファイルのあるディレクトリ・ツリーのルート・ディレクトリを含むデバイスを，マウント・ポイントを介してルートの下にマウントしない限り，POSIX パス名でアクセスできません。(このマウント・ポイントは，GNV が構成中にオプションで作成します)。図 12–1 を参照してください。

上記以外の方法としては，デバイスの最上位レベルにあるディレクトリ，つまり MFD (Master File Directory) をルートにして，そのデバイスに対してはマウント・ポイントを作成しないという方法もあります。この方法をとれば，デバイスにあるファイルはすべて，POSIX パス名でアクセスできます。ただし，デバイス (特にシステム・デバイス) の MFD をルートにすると，ユーザがルート・ディレクトリのファイルを表示したときに ("ls /"など)，MFD にあるファイルがすべて見えてしまいます。

図 12–1 POSIX ルートの位置



つまり，GNV ユーザにとっては，予想もしなかった OpenVMS ファイルが見えてしまうということです。この事情は DCL ユーザにとっても同様で，やはり予想外の UNIX ファイルが見えてしまいます。デバイスの MFD は，大半の一般ユーザからはアクセスできないようにしておく必要があるので，それらが見えてしまうと，予期しない障害につながることがあります。

したがって，ルートとしてデフォルトのルート・ディレクトリを使用するようお勧めします。

マウント・ポイントについての詳細は，第 12.7 節を参照してください。

12.5.2 SET ROOT コマンド

このコマンドのフォーマットは，次のとおりです。

SET ROOT デバイス名:ディレクトリ指定

SET ROOT は， POSIX ルートを定義するためのコマンドです。 POSIX パス名処理モードで動作している RMS と C ランタイム・ライブラリでは，パス名の先頭にある '/' を，ルートを表す文字として扱います。特に指定しなければ，SYS$SYSDEVICE:[PSX$ROOT]がルートになります。

ルートの定義は，リブートすると無効になります。

SET ROOT コマンドでは，次の修飾子を指定できます。

/LOG (**デフォルト**)
/NOLOG
SET ROOT コマンドでルートの設定に成功したことを表示するかどうかを制御します。

12.5.3 SHOW ROOT コマンド

SHOW ROOT は，システムのルートを表示するためのコマンドです。このコマンドは，プロセスのルートが定義されていれば，その現在の値も表示します。

このコマンドのフォーマットは，次のとおりです。

```
SHOW ROOT
```

12.6 現在の作業ディレクトリ

UNIX システムでは，先頭がスラッシュで始っていないパス名の解決に，現在の作業ディレクトリを使用します。

RMS でシンボリック・リンクに出会った場合は，そのシンボリック・リンクのあるディレクトリを現在の作業ディレクトリにして，パス名を解決します。

RMS に直接渡された POSIX パス名を解決する場合は，そのプロセスに現在割り当てられている OpenVMS のデフォルト・ディレクトリが，現在の作業ディレクトリになります。 OpenVMS のデフォルト・ディレクトリが複数個ある場合 (たとえば，ディレクトリの指定に検索リストが含まれている場合) は，検索リストを通して最初に見つかったディレクトリが，現在の作業ディレクトリになります。

12.7 マウント・ポイントの設定

ファイル・システムをマウント・ポイントにマウントしたり，マウント・ポイントからディスマウントしたりするためのユーティリティとして，mntとumntという新しいユーティリティが 2 つ使用できるようになりました。ただし，ファイル・システムをマウントまたはディスマウントできるのは， CMKRNL 特権を持つユーザだけです。これらのユーティリティは， GNV の一部として出荷されています。

これらのユーティリティには，マウント・ポイントの論理名テーブルが専用に用意されていて，管理されています。現在利用可能なマウント・ポイントを表示するように要求すると，これらのユーティリティはこのテーブルを使用してそれらを表示します。

これらのユーティリティのインタフェースは，次のとおりです。

`mnt`

既存のマウント・ポイントをリストにして表示します。

`mnt` file_system mount_point

ファイル・システムfile_systemを，マウント・ポイントmount_pointにマウントします。

umnt mount_point

マウント・ポイントmount_pointにマウントされているファイル・システムをアンマウントします。

umnt file_system

ファイル・システムfile_systemを，マウント・ポイントからアンマウントします。

umnt -A

ファイル・システムをすべて，マウント・ポイントからアンマウントします。

引数には， OpenVMS のファイル指定または POSIX パス名を指定できます。

12.8 NFS

V5.4 とそれより前の TCP/IP Services for OpenVMS でも， UNIX サーバと OpenVMS のセーブセットとの間でシンボリック・リンクのバックアップと復元を正しく行えるようにするために， NFS クライアントでシンボリック・リンクをサポートしていました。しかし，その機能は限られたものでした。そのサポートでは，クライアントがアプリケーション ACE を使用して，ファイルにそれがシンボリック・リンクであることを示すフラグを付けていました。 Version 5.5 になっても， ODS-2 のマウントについては，クライアントでこれと同じ動作が引き継がれています。しかし， ODS-5 のマウントについては，クライアントでアプリケーション ACE を検出したときにシンボリック・リンクが作成されるようになりました。このサポートによって，バージョンの古いクライアントで作成されたセーブセットでも，新しいバージョンで復元できるようになっています。

12.9 DCL

OpenVMS の DCL コマンドとユーティティは，シンボリック・リンクを検出すると，所定の動作を行うようになっています。つまり，シンボリック・リンクは，通常の処理として，そのリンク先までたどるようになっています。たとえば，シンボリック・リンク LINKFILE.TXT が TEXT.TXT を指している場合に "type LINKFILE.TXT"というコマンドを入力すると，そのリンクがたどられて，TEXT.TXT に含まれているテキストが表示されます。

また，シンボリック・リンクは，それ自身が直接指定されていなくても，たどられるようになっています。たとえば， "dir [...]*.TXT"というコマンドを処理しているときに，ディレクトリを参照しているシンボリック・リンクが検出されると，その参照先ディレクトリ (およびそのサブディレクトリ) にある， ".TXT"というタイプのファイルがすべて表示されます (注意: OpenVMS Version 8.3 でもディレクトリのワイルド

カードを処理しますが，そのときにシンボリック・リンクの参照先がディレクトリであるかどうかは調べません)。

しかし，シンボリック・リンクを常にたどらない方がよい場合もあります。また，シンボリック・リンクをたどるかどうかをユーザが指定できる方がよい場合もあります。

以下のコマンドには，シンボリック・リンクを決してたどらないものと，シンボリック・リンクをたどるかどうかを指定できるものがあります。

BACKUP

バックアップ操作でシンボリック・リンクが検出されると，そのシンボリック・リンク自体がコピーされます。この動作は，すべてのバックアップ・タイプ (物理，イメージ，およびファイル) に適用されます。

COPY

/SYMLINK
/NOSYMLINK (デフォルト)

入力ファイルがシンボリック・リンクの場合，そのシンボリック・リンクから参照されているファイルがコピーされます。

/SYMLINK 修飾子を指定すると，入力したシンボリック・リンクがすべてコピーされます。

CREATE

/SYMLINK="text"

シンボリック・リンクが作成されて，そこに，指定されたテキストが引用符なしで含められます。

作成されたシンボリック・リンクがその後でファイル名の処理中に検出されると，そのシンボリック・リンクの内容が読み取られ， POSIX パス名が指定されているものとして処理されます。

バージョンの古いシンボリック・リンクと共存させることはできません。

DELETE

入力ファイルがシンボリック・リンクの場合，そのシンボリック・リンク自体が削除されます。

DIRECTORY

/SYMLINK (デフォルト)
/NOSYMLINK

入力ファイルがシンボリック・リンクの場合，そのシンボリック・リンク自体のファイル属性が表示されます。ファイルの属性を表示するように要求すると，シンボリック・リンクの内容も表示されます。その際，ファイル名と内容の間に矢印が置かれます (たとえば， LINK.TXT -> FILE.TXT)。

ファイル指定としてシンボリック・リンクを入力するとともに /NOSYMLINK 修飾子を指定すると，そのシンボリック・リンクから参照されているファイルのファイル属性が表示されます。ただし，表示される名前は，シンボリック・リンク自体の名前のままです。

DUMP

/SYMLINK
/NOSYMLINK (デフォルト)

入力ファイルがシンボリック・リンクの場合，シンボリック・リンクから参照されているファイルがダンプされます。

/SYMLINK 修飾子を指定すると，入力したシンボリック・リンクがすべてダンプされます。

PURGE

入力ファイルがシンボリック・リンクの場合，そのシンボリック・リンク自体がパージされます。存在できるシンボリック・リンクのバージョンは 1 つだけであるため，このコマンドを実行しても，そのファイルに対する影響はありません。

RENAME

変換元ファイルがシンボリック・リンクの場合，そのシンボリック・リンク自体の名前が変更されます。

変更先ファイルがシンボリック・リンクであれば，この操作は失敗します。

SET FILE

/SYMLINK
/NOSYMLINK (デフォルト)

入力ファイルがシンボリック・リンクの場合，そのシンボリック・リンクから参照されているファイルが設定されます。

/SYMLINK 修飾子を指定すると，そのシンボリック・リンク自体が設定されます。

注意

- /EXCLUDE 修飾子を使用すれば，指定したシンボリック・リンクを対象から除外できます。除外されるシンボリック・リンクは，修飾子のファイル指定に一致したものです。シンボリック・リンクから参照されているファイルのファイル名は，その名前を指定しても除外されません。

  例:

  シンボリック・リンク LINK.TXT が，別のディレクトリにある TEXT.TXT を参照しているとします。

  /EXCLUDE=LINK.TXT と指定すると LINK.TXT が操作から除外されます (そのため，TEXT.TXT も除外されます)。

  /EXCLUDE=TEXT.TXT と指定すると，LINK.TXT も TEXT.TXT も操作から除外されません。

- シンボリック・リンクを指定して F$FILE_ATTRIBUTES を実行すると，そのシンボリック・リンクから参照されているファイルが対象になります。これに対して，新しいレキシカル関数 F$SYMLINK_ATTRIBUTES の場合は，そのシンボリック・リンク自体の情報が返されます。また，新しいレキシカル関数 F$READLINK の場合は，入力ファイルがシンボリック・リンクであるとそのテキスト形式の内容が返され，入力ファイルがシンボリック・リンクでないと NULL が返されます。

- 一部のコンパイラやユーティリティでは，入力ファイル指定，または修飾子に対する引数として，引用符で囲んだパス名を指定できます。このようなコンパイラやユーティリティには，C コンパイラ，C++ コンパイラ，CXXLINK，Linker，および Librarian があります。Linker については，オプション・ファイルにも引用符で囲んだパス名を含めることができます。コンパイラの/INCLUDE_DIRECTORY 修飾子については，引用符で囲んだパス名ではなく，以前と同じ標準の POSIX パス名を指定できます。Java コンパイラについても，以前と同じ標準の POSIX パス名を指定できます。また，SET DEFAULT の入力には，引用符で囲んだパス名を指定できます。

12.10 GNV

GNV は，OpenVMS システムで Open Group のユーティリティを使用できるようにするためのメカニズムです。GNV はオープン・ソース・プロジェクトになっていて，Sourceforge の Web サイトでアップデートがリリースされています。GNV は，OpenVMS Open Source CD から入手することもできます。

GNV は，そのファイル・アクセス機能を C RTL にすべて依存しているので，いっさい変更されていません。その理由は，提供されているユーティリティが，Open Group で定義されているシンボリック・リンクの仕様に従って動作するようにする必要があるからです。ユーザが新しいシンボリック・リンクと POSIX パス名のサポートを利用する場合は，BASH を起動する前に，C RTL の DECC$POSIX_COMPLIANT_PATHNAMES 機能論理名を適切に設定する必要があります。

12.11 制限事項

シンボリック・リンクのサポートには，次の制約があります。

- シンボリック・リンクに対して SET FILE/REMOVE は実行できない。

  シンボリック・リンクを対象にして SET FILE/REMOVE を実行しても，そのシンボリック・リンクは削除されません。また，エラーも表示されません。

- RMS とシンボリック・リンクに関するコーディング上の注意事項

  シンボリック・リンクと POSIX パス名の処理を導入すると，それに伴って，ファイル指定の中のデバイス名が，そのファイルの存在しているデバイスを正しく示さなくなってしまうことがあります。 SYS$PARSE/SYS$SEARCH から返される NAM$T_DVI を調べれば，そのファイルの存在しているデバイスを確認することができます。

A

バージョンへの依存性を示す表

HP Cの各バージョンでは，新しい関数がHP Cランタイム・ライブラリに追加されています。これらの関数は実装され，OpenVMS オペレーティング・システムとともに提供されますが，これらの関数のプロトタイプを格納したヘッダ・ファイルとドキュメントは，HP Cコンパイラの各バージョンに付属しています。

HP Cの新しいバージョンには，古い OpenVMS システムではサポートされていない関数のヘッダ・ファイルとドキュメントが含まれている可能性があります。たとえば，ターゲットのオペレーティング・システム・プラットフォームが OpenVMS Version 7.2 の場合，OpenVMS Version 7.3 で導入されたHP C RTL 関数を使用することはできませんが，これらの関数もドキュメントに記載されています。

この付録では，OpenVMS の最近のバージョンでどのHP C RTL 関数がサポートされているかをまとめた表を示します。これらの表は，ターゲットの OpenVMS プラットフォームで使用できない関数を判断するのに役立ちます。

A.1 OpenVMS VAX，Alpha，および Integrity のすべてのバージョンで使用できる関数

表 A–1 は，OpenVMS VAX，OpenVMS Alpha および OpenVMS Integrity のすべてのバージョンで使用できる関数を示しています。

表 A–1 すべての OpenVMS システムで使用できる関数

abort	abs	access	acos
alarm	asctime	asin	assert
atan2	atan	atexit	atof
atoi	atoll (Alpha)	atol	atoq (Alpha)
box	brk	bsearch	cabs
calloc	ceil	cfree	chdir
chmod	chown	clearerr	clock
close	cosh	cos	creat
ctermid	ctime	cuserid	decc$crtl_init
decc$fix_time	decc$from_vms	decc$match_wild	decc$record_read

(次ページに続く)

表 A–1 (続き) すべての OpenVMS システムで使用できる関数

decc$record_write	decc$set_reentrancy	decc$to_vms	decc$translate_vms
delete	delwin	difftime	div
dup2	dup	ecvt	endwin
execle	execlp	execl	execve
execvp	execv	exit	_exit
exp	fabs	fclose	fcvt
fdopen	feof	ferror	fflush
fgetc	fgetname	fgetpos	fgets
fileno	floor	fmod	fopen
fprintf	fputc	fputs	fread
free	freopen	frexp	fscanf
fseek	fsetpos	fstat	fsync
ftell	ftime	fwait	fwrite
gcvt	getchar	getcwd	getc
getegid	getenv	geteuid	getgid
getname	getpid	getppid	gets
getuid	getw	gmtime	gsignal
hypot	initscr	isalnum	isalpha
isapipe	isascii	isatty	iscntrl
isdigit	isgraph	islower	isprint
ispunct	isspace	isupper	isxdigit
kill	labs	ldexp	ldiv
llabs (Alpha)	lldiv (Alpha)	localeconv	localtime
log10	log	longjmp	longname
lseek	lwait	malloc	mblen
mbstowcs	mbtowc	memchr	memcmp
memcpy	memmove	memset	mkdir
mktemp	mktime	modf	mvwin
mv[w]addstr	newwin	nice	open
overlay	overwrite	pause	perror
pipe	pow	printf	putchar
putc	puts	putw	qabs (Alpha)
qdiv (Alpha)	qsort	raise	rand
read	realloc	remove	rename
rewind	sbrk	scanf	scroll
setbuf	setgid	setjmp	setlocale
setuid	setvbuf	sigblock	signal
sigpause	sigstack (VAX)	sigvec	sinh

(次ページに続く)

表 A–1 (続き) すべての OpenVMS システムで使用できる関数

sin	sleep	sprintf	sqrt
srand	sscanf	ssignal	stat
strcat	strchr	strcmp	strcoll
strcpy	strcspn	strerror	strftime
strlen	strncat	strncmp	strncpy
strpbrk	strrchr	strspn	strstr
strtod	strtok	strtoll (Alpha)	strtol
strtoq (Alpha)	strtoull (Alpha)	strtoul	strtouq (Alpha)
strxfrm	subwin	system	tanh
tan	times	time	tmpfile
tmpnam	toascii	tolower	_tolower
touchwin	toupper	_toupper	ttyname
umask	ungetc	vaxc$calloc_opt	vaxc$cfree_opt
vaxc$crtl_init	vaxc$establish	vaxc$free_opt	vaxc$malloc_opt
vaxc$realloc_opt	va_arg	va_count	va_end
va_start	va_start_1	vfork	vfprintf
vprintf	vsprintf	wait	wcstombs
wctomb	write	[w]addch	[w]addstr
[w]clear	[w]clrattr	[w]clrtobot	[w]clrtoeol
[w]delch	[w]deleteln	[w]erase	[w]getch
[w]getstr	[w]inch	[w]insch	[w]insertln
[w]insstr	[w]move	[w]printw	[w]refresh
[w]scanw	[w]setattr	[w]standend	[w]standout

A.2 OpenVMS Version 6.2 およびそれ以降で使用できる関数

表 A–2 は， OpenVMS VAX および OpenVMS Alpha Version 6.2 およびそれ以降のバージョンで使用できる関数を示しています。

表 A–2 OpenVMS Version 6.2 で追加された関数

catclose	catgets	catopen	fgetwc
fgetws	fputwc	fputws	getopt
getwc	getwchar	iconv	iconv_close
iconv_open	iswalnum	iswalpha	iswcntrl
iswctype	iswdigit	iswgraph	iswlower
iswprint	iswpunct	iswspace	iswupper
iswxdigit	nl_langinfo	putwc	putwchar
strnlen	strptime	towlower	towupper
ungetwc	wcscat	wcschr	wcscmp
wcscoll	wcscpy	wcscspn	wcsftime
wcslen	wcsncat	wcsncmp	wcsncpy
wcspbrk	wcsrchr	wcsspn	wcstol
wcstoul	wcswcs	wcswidth	wcsxfrm
wcstod	wctype	wcwidth	wcstok

A.3 OpenVMS Version 7.0 およびそれ以降で使用できる関数

表 A–3 は，OpenVMS VAX および OpenVMS Alpha Version 7.0 およびそれ以降のバージョンで使用できる関数を示しています。

表 A–3 OpenVMS Version 7.0 で追加された関数

basename	bcmp	bcopy	btowc
bzero	closedir	confstr	dirname
drand48	erand48	ffs	fpathconf
ftruncate	ftw	fwide	fwprintf
fwscanf	getclock	getdtablesize	getitimer
getlogin	getpagesize	getpwnam	getpwuid
gettimeofday	index	initstate	jrand48
lcong48	lrand48	mbrlen	mbrtowc
mbsinit	mbsrtowcs	memccpy	mkstemp
mmap	mprotect	mrand48	msync
munmap	nrand48	opendir	pathconf
pclose	popen	putenv	random
readdir	rewinddir	rindex	rmdir
seed48	seekdir	setenv	setitimer
setstate	sigaction	sigaddset	sigdelset
sigemptyset	sigfillset	sigismember	siglongjmp
sigpending	sigprocmask	sigsetjmp	sigsuspend
srand48	srandom	strcasecmp	strdup
strfmon	strncasecmp	strsep	swab
swprintf	swscanf	sysconf	telldir
tempnam	towctrans	truncate	tzset
ualarm	uname	unlink	unsetenv
usleep	vfwprintf	vswprintf	vwprintf
wait3	wait4	waitpid	wcrtomb
wcsrtombs	wcsstr	wctob	wctrans
wmemchr	wmemcmp	wmemcpy	wmemmove
wmemset	wprintf	wscanf	

A.4 OpenVMS Alpha Version 7.0 およびそれ以降で使用できる関数

表 A–4 は， OpenVMS Alpha Version 7.0 およびそれ以降のバージョンで使用できる関数を示しています。

表 A–4 OpenVMS Alpha Version 7.0 で追加された関数

_basename32	_basename64	_bsearch32	_bsearch64
_calloc32	_calloc64	_catgets32	_catgets64
_ctermid32	_ctermid64	_cuserid32	_cuserid64
_dirname32	_dirname64	_fgetname32	_fgetname64
_fgets32	_fgets64	_fgetws32	_fgetws64
_gcvt32	_gcvt64	_getcwd32	_getcwd64
_getname32	_getname64	_gets32	_gets64
_index32	_index64	_longname32	_longname64
_malloc32	_malloc64	_mbsrtowcs32	_mbsrtowcs64
_memccpy32	_memccpy64	_memchr32	_memchr64
_memcpy32	_memcpy64	_memmove32	_memmove64
_memset32	_memset64	_mktemp32	_mktemp64
_mmap32	_mmap64	_qsort32	_qsort64
_realloc32	_realloc64	_rindex32	_rindex64
_strcat32	_strcat64	_strchr32	_strchr64
_strcpy32	_strcpy64	_strdup32	_strdup64
_strncat32	_strncat64	_strncpy32	_strncpy64
_strpbrk32	_strpbrk64	_strptime32	_strptime64
_strrchr32	_strrchr64	_strsep32	_strsep64
_strstr32	_strstr64	_strtod32	_strtod64
_strtok32	_strtok64	_strtol32	_strtol64
_strtoll32	_strtoll64	_strtoq32	_strtoq64
_strtoul32	_strtoul64	_strtoull32	_strtoull64
_strtouq32	_strtouq64	_tmpnam32	_tmpnam64
_wcscat32	_wcscat64	_wcschr32	_wcschr64
_wcscpy32	_wcscpy64	_wcsncat32	_wcsncat64
_wcsncpy32	_wcsncpy64	_wcspbrk32	_wcspbrk64
_wcsrchr32	_wcsrchr64	_wcsrtombs32	_wcsrtombs64
_wcsstr32	_wcsstr64	_wcstok32	_wcstok64
_wcstol32	_wcstol64	_wcstoul32	_wcstoul64
_wcswcs32	_wcswcs64	_wmemchr32	_wmemchr64
_wmemcpy32	_wmemcpy64	_wmemmove32	_wmemmove64
_wmemset32	_wmemset64		

A.5 OpenVMS Version 7.2 およびそれ以降で使用できる関数

表 A–5 は， OpenVMS VAX および OpenVMS Alpha Version 7.2 およびそれ以降のバージョンで使用できる関数を示しています。

表 A–5 OpenVMS Version 7.2 で追加された関数

asctime_r	dlerror
ctime_r	dlopen
decc$set_child_standard_streams	dlsym
decc$validate_wchar	fcntl
decc$write_eof_to_mbx	gmtime_r
dlclose	localtime_r

A.6 OpenVMS Version 7.3 およびそれ以降で使用できる関数

表 A–6 は， OpenVMS VAX および OpenVMS Alpha Version 7.3 およびそれ以降のバージョンで使用できる関数を示しています。

表 A–6 OpenVMS Version 7.3 で追加された関数

fchown
link
utime
utimes
writev

A.7 OpenVMS Version 7.3-1 およびそれ以降で使用できる関数

表 A–7 は， OpenVMS Alpha Version 7.3-1 およびそれ以降のバージョンで使用できる関数を示しています。

表 A–7 OpenVMS Version 7.3-1 で追加された関数

access	ftello
chmod	ftw
chown	readdir_r
decc$feature_get_index	stat
decc$feature_get_name	vfscanf

(次ページに続く)

表 A–7 (続き) OpenVMS Version 7.3-1 で追加された関数

decc$feature_get_value	vfwscanf
decc$feature_set_value	vscanf
fseeko	vwscanf
fstat	vsscanf
	vswscanf

A.8 OpenVMS Version 7.3-2 およびそれ以降で使用できる関数

表 A–8 は, OpenVMS Alpha Version 7.3-2 およびそれ以降のバージョンで使用できる関数を示しています。

表 A–8 OpenVMS Version 7.3-2 で追加された関数

a64l	clock_getres	clock_gettime	clock_settime
endgrent	getgrent	getgrgid	getgrgid_r
getgrnam	getgrnam_r	getpgid	getpgrp
_getpwnam64	getpwnam_r	_getpwnam_r64	_getpwent64
getpwuid	_getpwuid64	getpwuid_r	_getpwuid_r64
getsid	l64a	nanosleep	poll
pread	pwrite	rand_r	readv
_readv64	seteuid	setgrent	setpgid
setpgrp	setregid	setreuid	setsid
sighold	sigignore	sigrelse	sigtimedwait
sigwait	sigwaitinfo	snprintf	ttyname_r
vsnprintf	__writev64	decc$set_child_ default_dir	

A.9 OpenVMS Version 8.2 およびそれ以降で使用できる関数

表 A–9 は, OpenVMS Alpha および Integrity の V8.2 以降で使用できる関数を示しています。

表 A–9 OpenVMS Version 8.2 で追加された関数

clearerr_unlocked	feof_unlocked
ferror_unlocked	fgetc_unlocked
fputc_unlocked	flockfile

(次ページに続く)

表 A–9 (続き) OpenVMS Version 8.2 で追加された関数

ftrylockfile	funlockfile
getc_unlocked	getchar_unlocked
putc_unlocked	putchar_unlocked
statvfs	fstatvfs
_glob32	_glob64
_globfree32	_globfree64
socketpair	

A.10 OpenVMS Version 8.3 およびそれ以降で使用できる関数

表 A–10 は, OpenVMS Alpha および Integrity の V8.3 以降で使用できる関数を示しています。

表 A–10 OpenVMS Version 8.3 で追加された関数

crypt	readlink
encrypt	realpath
setkey	symlink
lchown	unlink
lstat	fchmod

A.11 OpenVMS Version 8.4 およびそれ以降で使用できる関数

表 A–11 は, OpenVMS Alpha および Integrity の V8.4 以降で使用できる関数を示しています。

表 A–11 OpenVMS Version 8.4 で追加された関数

ftok	sem_init
semctl	sem_open
semget	sem_post
semop	sem_timedwait
sem_close	sem_trywait
sem_destroy	sem_unlink
sem_getvalue	sem_wait

B

非標準ヘッダに複製されているプロトタイプ

さまざまな標準では，各標準関数をどのヘッダ・ファイルで定義しなければならないかを規定しています。これは，本書の「リファレンス・セクション」の各関数プロトタイプに示している，取り込まれるヘッダ・ファイルです。

しかし，標準で定義されている多くの関数はすでに複数のオペレーティング・システムに存在し，異なるヘッダ・ファイルに定義されています。特に，ヘッダ・ファイル<processes.h>，<unixio.h>，<unixlib.h>を使用する OpenVMS システムでは，このことが当てはまります。

したがって，これらの関数の上位互換性を提供するために，プロトタイプは標準で定義されているヘッダ・ファイルだけでなく，必要とされる他のヘッダ・ファイルへも複製されています。

表 B–1 はこれらの関数を示しています。

表 B–1 複製されているプロトタイプ

関数	複製先	標準で規定されているファイル
access	<unixio.h>	<unistd.h>
alarm	<signal.h>	<unistd.h>
bcmp	<string.h>	<strings.h>
bcopy	<string.h>	<strings.h>
bzero	<string.h>	<strings.h>
chdir	<unixio.h>	<unistd.h>
chmod	<unixio.h>	<stat.h>
chown	<unixio.h>	<unistd.h>
close	<unixio.h>	<unistd.h>
creat	<unixio.h>	<fcntl.h>
ctermid	<stdio.h>	<unistd.h>
cuserid	<stdio.h>	<unistd.h>
dirname	<string.h>	<libgen.h>
dup	<unixio.h>	<unistd.h>
dup2	<unixio.h>	<unistd.h>
ecvt	<unixlib.h>	<stdlib.h>

(次ページに続く)

表 B–1 (続き) 複製されているプロトタイプ

関数	複製先	標準で規定されているファイル
execl	<processes.h>	<unistd.h>
execle	<processes.h>	<unistd.h>
execlp	<processes.h>	<unistd.h>
execv	<processes.h>	<unistd.h>
execve	<processes.h>	<unistd.h>
execvp	<processes.h>	<unistd.h>
_exit	<stdlib.h>	<unistd.h>
fcvt	<unixlib.h>	<stdlib.h>
ffs	<string.h>	<strings.h>
fsync	<stdio.h>	<unistd.h>
ftime	<time.h>	<timeb.h>
gcvt	<unixlib.h>	<stdlib.h>
getcwd	<unixlib.h>	<unistd.h>
getegid	<unixlib.h>	<unistd.h>
getenv	<unixlib.h>	<stdlib.h>
geteuid	<unixlib.h>	<unistd.h>
getgid	<unixlib.h>	<unistd.h>
getopt	<stdio.h>	<unistd.h>
getpid	<unixlib.h>	<unistd.h>
getppid	<unixlib.h>	<unistd.h>
getuid	<unixlib.h>	<unistd.h>
index	<string.h>	<strings.h>
isatty	<unixio.h>	<unistd.h>
lseek	<unixio.h>	<unistd.h>
mkdir	<unixlib.h>	<stat.h>
mktemp	<unixio.h>	<stdlib.h>
nice	<stdlib.h>	<unistd.h>
open	<unixio.h>	<fcntl.h>
pause	<signal.h>	<unistd.h>
pipe	<processes.h>	<unistd.h>
read	<unixio.h>	<unistd.h>
rindex	<string.h>	<strings.h>
sbrk	<stdlib.h>	<unistd.h>
setgid	<unixlib.h>	<unistd.h>
setuid	<unixlib.h>	<unistd.h>
sleep	<signal.h>	<unistd.h>
strcasecmp	<string.h>	<strings.h>

(次ページに続く)

表 B–1 (続き) 複製されているプロトタイプ

関数	複製先	標準で規定されているファイル
strncasecmp	<string.h>	<strings.h>
system	<processes.h>	<stdlib.h>
times	<time.h>	<times.h>
umask	<stdlib.h>	<stat.h>
vfork	<processes.h>	<unistd.h>
wait	<processes.h>	<wait.h>
write	<unixio.h>	<unistd.h>

索引

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X

イ

ウ

エ

オ

カ

キ

ク

コ

サ

シ

メ

モ

ユ

ヨ

ラ

リ

ル

レ

ロ

ワ

OpenVMS
HP C **ランタイム・ライブラリ・リファレンス・マニュアル（上巻）**

2011 年 5 月　発行

日本ヒューレット・パッカード株式会社

〒 102-0076 東京都千代田区五番町 7 番地

電話　(03)3512-5700 (大代表)

5991-6625.2