Fortran ライブラリ・リファレンス

Sun WorkShop 6
Fortran 77 および Fortran 95

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
U.S.A. 650-960-1300

Part No. 806-4844-01
2000 年 6 月 Revision A

原典： *Fortran Library Reference (Sun WorkShop 6 Fortran 95 - Fortran 77)*
Part No: 806-3590-10
Revision A

製品名の変更について

Sun は新しい開発製品戦略の一環として、Sun の開発ツール群の製品名を Sun WorkShop™ から Forte™ Developer に変更いたしました。製品自体の内容に変更はなく、従来通りの高品質をお届けいたします。

これまでの Sun の主力製品である基本プログラミングツールに、Forte Fusion™ や Forte™ for Java™ といった Forte 開発ツールの得意とする、マルチプラットフォームおよびビジネスアプリケーション実装の機能を盛り込むことで、より広範囲できめ細かな製品ラインが完成されました。

WorkShop 5.0 で使用されていた名称と、Forte Developer 6 で使用される新しい名称の対応については、以下の表をご覧ください。

旧名称	新名称
Sun Visual WorkShop™ C++	Forte™ C++ Enterprise Edition 6
Sun Visual WorkShop™ C++ Personal Edition	Forte™ C++ Personal Edition 6
Sun Performance WorkShop™ Fortran	Forte™ for High Performance Computing 6
Sun Performance WorkShop™ Fortran Personal Edition	Forte™ Fortran Desktop Edition 6
Sun WorkShop Professional™ C	Forte™ C 6
Sun WorkShop™ University Edition	Forte™ Developer University Edition 6

製品名の変更に加えて、次の 2 つの製品について大きな変更があります。

- Forte for High Performance Computing には Sun Performance WorkShop Fortran に含まれていたすべてのツール、および C++ コンパイラが含まれます。したがって、High Performance Computing のユーザーは開発用に 1 つの製品だけを購入すれば済むことになります。
- Forte Fortran Desktop Edition は以前の Sun Performance WorkShop Personal Edition と同じです。ただし、この製品に含まれる Fortran コンパイラでは、自動並列化されたコード、および明示的な指令に基づいた並列コードは生成できません。この機能は Forte for High Performance Computing に含まれる Fortran コンパイラでは使用できます。

Sun の開発製品を引き続きご利用いただきましてありがとうございます。今後もみなさまのご要望にお応えする製品をお届けできるよう努力してまいります。

目次

表目次

はじめに

このマニュアルでは、Sun WorkShop™ Fortran ライブラリルーチンについて説明しています。

このマニュアルは Fortran 言語と Solaris™ オペレーティング環境に関する実用的な知識を持つプログラマを対象にしています。

マルチプラットフォーム対応

この Sun WorkShop リリースは、Solaris 2.6、7、および 8 のオペレーティング環境 (SPARC™ プラットフォームおよび Intel プラットフォーム) をサポートしています。

注 – x86 とは、Pentium、Pentium Pro、Pentium II プロセッサおよび、これらと互換性のある AMD および Cyrix 製のマイクロプロセッサチップを含む、Intel 8086 マイクロプロセッサチップ群を意味しています。このマニュアルでは、これらすべてのプラットフォームアーキテクチャを総称して x86 と呼んでいますが、製品名では Intel プラットフォームと記述しています。

特定のプラットフォームに対して、このリリースの `f77` コンパイラが提供されているかどうかは、Sun WorkShop READMEs ディレクトリにある Fortran 77 README ファイル `fortran_77` を参照してください。

Sun WorkShop 開発ツールへのアクセス方法

Sun WorkShop 製品コンポーネントとマニュアルページは標準ディレクトリ /usr/bin および /usr/share/man にはインストールされません。そのため PATH および MANPATH 環境変数を変更して Sun WorkShop コンパイラとツールにアクセスできるようにする必要があります。

PATH 環境変数を設定する必要があるかどうか判断するには以下を実行します。

1. **次のように入力して、PATH 変数の現在値を表示します。**

```
% echo $PATH
```

2. **出力内容から /opt/SUNWspro/bin を含むパスの文字列を検索します。**

パスがある場合は、PATH 変数は Sun WorkShop 開発ツールにアクセスできるように設定されています。パスがない場合は、この節の指示に従って、PATH 環境変数を設定してください。

MANPATH 環境変数を設定する必要があるかどうか判断するには以下を実行します。

1. **次のように入力して、workshop マニュアルページを表示します。**

```
% man workshop
```

2. **出力された場合、内容を確認します。**

workshop(1) マニュアルページが見つからないか、表示されたマニュアルページがインストールされたソフトウェアの現バージョンのものと異なる場合は、この節の指示に従って MANPATH 環境変数を設定してください。

注 – この節に記載されている情報は Sun WorkShop 6 製品が /opt ディレクトリにインストールされていることを想定しています。 Sun WorkShop ソフトウェアが /opt ディレクトリにインストールされていない場合は、システム管理者に連絡してください。

PATH 変数および MANPATH 変数は、C シェルを使用している場合はホームディレクトリの下の .cshrc ファイルに設定する必要があります。Bourne シェルか Korn シェルを使用している場合は、ホームディレクトリの下の .profile ファイルに設定する必要があります。

- Sun WorkShop コマンドを使用するには、 PATH 変数に以下を追加してください。

  /opt/SUNWspro/bin

- man コマンドで、Sun WorkShop マニュアルページにアクセスするには、MANPATH 変数に以下を追加してください。

  /opt/SUNWspro/man

  PATH 変数についての詳細は、csh(1)、sh(1) および ksh(1) のマニュアルページを参照してください。MANPATH 変数についての詳細は、 man(1) のマニュアルページを参照してください。このリリースにアクセスするために PATH および MANPATH 変数を設定する方法の詳細は、『Sun WorkShop インストールガイド』を参照するか、システム管理者にお問い合わせください。

内容の紹介

このマニュアルは次の章で構成されています。

「Fortran ライブラリルーチン」では、Fortran のライブラリルーチンをアルファベット順に説明します。

書体と記号について

このマニュアルで使用している書体と記号について説明します。

表 P-1　このマニュアルで使用している書体と記号

書体または記号	意味	例
`AaBbCc123`	コマンド名、ファイル名、ディレクトリ名、画面上のコンピュータ出力、コーディング例。	`.login` ファイルを編集します。 `ls -a` を使用してすべてのファイルを表示します。 `machine_name% You have mail.`
`AaBbCc123`	ユーザーが入力する文字を、画面上のコンピュータ出力と区別して表わします。	`machine_name% su` `Password:`
AaBbCc123 または ゴシック	コマンド行の可変部分。実際の名前または実際の値と置き換えてください。	`rm` *filename* と入力します。 `rm` **ファイル名** と入力します。
『 』	参照する書名を示します。	『SPARCstorage Array ユーザーマニュアル』
「 」	参照する章、節、または、強調する語を示します。	第 6 章「データの管理」を参照してください。 この操作ができるのは、「スーパーユーザー」だけです。
\	枠で囲まれたコード例で、テキストがページ行幅を超える場合、バックスラッシュは、継続を示します。	`machinename% grep '^#define \` `  XV_VERSION_STRING'`
➤	階層メニューのサブメニューを選択することを示します。	作成： 「返信」➤「送信者へ」

- FORTRAN 77 の例はタブ書式で、Fortran 95 の例は自由書式で示します。FORTRAN 77 と Fortran 95 に共通の例は、特に指示がない限りタブ書式で示します。

- 大文字は Fortran キーワードと組み込み関数 (例: `PRINT`) を、小文字または大文字と小文字の混合は変数 (例: `TbarX`) を示します。
- Sun Fortran コンパイラを、`f77` または `f95` とコマンド名で略記することがあります。FORTRAN 77 コンパイラと Fortran 95 コンパイラに共通の情報は、`f77/f95` と記述します。
- オンラインマニュアルページへの参照は、トピック名とセクション番号とともに表示されます。たとえば、`GETENV` への参照は、`getenv`(3F) と表示されます。`getenv`(3F) とは、このページにアクセスするためのコマンドが `man -s 3F getenv` であるという意味です。
- FORTRAN 77 規格では、「FORTRAN」とすべて大文字で表記する旧表記規則を使用しています。サンのマニュアルでは FORTRAN と Fortran の両方を使用しています。現在の表記規則では、「Fortran 95」と小文字を使用しています。

シェルプロンプトについて

シェルプロンプトの例を以下に示します。

表 P-2 シェルプロンプト

シェル	プロンプト
UNIX の C シェル	`machine_name%`
UNIX の Bourne シェルと Korn シェル	`machine_name$`
スーパーユーザー (シェルの種類を問わない)	`#`

関連マニュアル

以下の方法で、関連マニュアルにアクセスすることができます。

- インターネットの `docs.sun.com` の Web サイトからアクセスできます。特定の本のタイトルで検索するか、主題、マニュアルコレクションまたは製品別にブラウズすることができます。

  `http://docs.sun.com`

- ローカルシステムまたはローカルネットワークにインストールされた Sun WorkShop 製品からアクセスできます。Sun WorkShop 6 HTML 文書 (マニュアル、オンラインヘルプ、マニュアルページ、各コンポーネントの README ファイル、リリースノート) が、インストールした Sun WorkShop 6 製品から参照可能です。HTML 文書にアクセスするには、次のいずれかを実行します。
  - Sun WorkShop または Sun WorkShop™ TeamWare ウィンドウで、「ヘルプ」➤「オンラインマニュアルについて」を選択します。
  - Netscape™ Communicator 4.0 またはその互換バージョンのブラウザで、以下のファイルを開きます。

    `/opt/SUNWspro/docs/ja/index.html`

  参照できる Sun WorkShop 6 HTML 文書の一覧がブラウザに表示されます。一覧にあるマニュアルを開くには、マニュアルのタイトルをクリックしてください。

表 P-3 は、Sun WorkShop 6 関連マニュアルをマニュアルコレクション別に一覧にしたものです。

表 P-3 マニュアルコレクション別 Sun WorkShop 6 関連マニュアル

マニュアルコレクション	マニュアルタイトル	内容の説明
Forte Developer 6 / Sun WorkShop 6 リリースマニュアル	Sun WorkShop 6 マニュアルの概要	Sun WorkShop 6 で使用可能なマニュアルとそのアクセス方法について説明しています。
	Sun WorkShop の新機能	Sun WorkShop 6 の現在のリリースと以前のリリースでの新機能についての情報を記載しています。
	Sun WorkShop 6 リリースノート	インストールの詳細と Sun WorkShop 6 最終リリースの直前に判明した情報を記載しています。このマニュアルはコンポーネントごとの README ファイルにある情報を補足するものです。

表 P-3　マニュアルコレクション別 Sun WorkShop 6 関連マニュアル

マニュアルコレクション	マニュアルタイトル	内容の説明
Forte Developer 6 / Sun WorkShop 6	プログラムのパフォーマンス解析	新しい標本コレクタと標本アナラザの使い方について説明しています (上級者向けのプロファイリング事例と説明付き)。コマンド行解析ツール `er_print`、ループツール、ループレポートユーティリティおよび UNIX プロファイルツール `prof`、`gprof`、`tcov` についての情報も含んでいます。
	dbx コマンドによるデバッグ	dbx コマンドを使ってプログラムをデバッグする方法について説明しています。参考情報として、同じデバッグ処理を Sun WorkShop デバッグウィンドウを使って実行する方法も記載しています。
	Sun WorkShop の概要	Sun WorkShop 統合プログラミング環境の基本的なプログラム開発機能について説明しています。
Forte C 6 / Sun WorkShop 6 Compilers C	C ユーザーズガイド	C コンパイラオプション、サン固有の機能 (プラグマ、`lint` ツール、並列化、64 ビットオペレーティングシステムへの移行および ANSI/ISO 準拠 C) について説明しています。
Forte C++ 6 / Sun WorkShop 6 Compilers C++	C++ ライブラリ・リファレンス	C++ ライブラリについて説明しています。C++ 標準ライブラリ、Tools.h++ クラスライブラリ、Sun WorkShop Memory Monitor、`Iostream` および複素数の情報も含みます。
	C++ 移行ガイド	コードを本バージョンの Sun WorkShop C++ コンパイラに移行する方法について説明しています。

表 P-3　マニュアルコレクション別 Sun WorkShop 6 関連マニュアル

マニュアルコレクション	マニュアルタイトル	内容の説明
	C++ プログラミングガイド	新しい機能を使ってより効率的なプログラムを記述する方法について説明しています。テンプレート、例外処理、実行時の型識別、キャスト演算、パフォーマンス、およびマルチスレッド対応のプログラムに関する情報も記載されています。
	C++ ユーザーズガイド	コマンド行オプションとコンパイラの使い方についての情報を記載しています。
	Sun WorkShop Memory Monitor ユーザーズガイド	C および C++ のメモリー管理で生じた問題を Sun WorkShop Memory Monitor で解決する方法について説明しています。このマニュアルはインストールした製品 (`/opt/SUNWspro/docs/ja/index.html`) からのみ参照可能で、`docs.sun.com` Web サイトで参照することはできません。
Forte for High Performance Computing 6 / Sun WorkShop 6 Compilers Fortran 77/95	Fortran ライブラリ・リファレンス	Fortran コンパイラによって提供されるライブラリルーチンの詳細について説明しています。
	Fortran プログラミングガイド	入出力、ライブラリ、プログラム分析、デバッグおよびパフォーマンスに関連する内容を記述しています。
	Fortran ユーザーズガイド	コマンド行オプションとコンパイラの使い方についての情報を記載しています。
	FORTRAN 77 言語リファレンス	Fortran 77 言語の包括的な参照情報を記載しています。
	Fortran 95 区間演算プログラミングリファレンス	Fortran 95 コンパイラによってサポートされる組み込み `INTERVAL` データについて説明しています。
Forte TeamWare 6 / Sun WorkShop TeamWare 6	Sun WorkShop TeamWare ユーザーズガイド	Sun WorkShop TeamWare コード管理ツールの使用方法について説明しています。

表 P-3　マニュアルコレクション別 Sun WorkShop 6 関連マニュアル

マニュアルコレクション	マニュアルタイトル	内容の説明
Forte Developer 6/ Sun WorkShop Visual 6	Sun WorkShop Visual ユーザーズガイド	C++ と Java™ の GUI (グラフィカルユーザーインタフェース) を Sun WorkShop Visual を使用して作成する方法について説明しています。このマニュアルには、旧リリース (Sun WorkShop Visual 5.0) から変更のない機能が記載されています。
	Sun WorkShop Visual の新機能	Sun WorkShop Visual 6.0 で追加または変更された機能について説明しています。
Forte / Sun Performance Library 6	Sun Performance Library Reference (英語のみ)	コンピュータによる線形代数および高速フーリエ変換を実行するサブルーチンと関数の最適化ライブラリについて説明しています。
	Sun Performance Library User's Guide (英語のみ)	線形代数で発生した問題の解決に使用されるサブルーチンと関数のコレクションである Sun Performance Library のサン固有の機能の使用方法について説明しています。
数値計算ガイド	数値計算ガイド	浮動小数点演算における数値の精度に関する問題について説明しています。
標準ライブラリ 2	Standard C++ Library Class Reference (英語のみ)	標準 C++ の詳細について説明しています。
	標準 C++ ライブラリ・ユーザーズガイド	標準 C++ ライブラリの使用方法について説明しています。
Tools.h++ 7	Tools.h++ 7.0 ユーザーズガイド	Tools.h++ クラスライブラリの詳細について説明しています。
	Tools.h++ 7.0 クラスライブラリ・リファレンスマニュアル	C++ クラスを使用して、プログラム効率を向上させる方法について説明しています。

表 P-4 は、docs.sun.com の Web サイトからアクセスできる Solaris 関連マニュアルの一覧です。

表 P-4 Solaris 関連マニュアル

マニュアルコレクション	マニュアルタイトル	内容の説明
Solaris ソフトウェア開発	リンカーとライブラリ	Solaris リンクエディタと実行時リンカーの操作およびそれらが操作するオブジェクトについて説明しています。
	プログラミングユーティリティ	Solaris オペレーティング環境で使用可能な特殊組み込みプログラミングツールに関する開発者向けの情報を記載しています。

Fortran ライブラリルーチン

この章では、Fortran のライブラリルーチンをアルファベット順に説明します。VMS の組み込み関数については、『FORTRAN 77 言語リファレンス』を参照してください。本章で説明するルーチンにはすべて、対応するマニュアルページがマニュアルライブラリのセクション 3F に用意されています。たとえば、`man -s 3F access` を実行すると、`access` というライブラリルーチンに関するマニュアルページの内容が表示されます。

Fortran や C から呼び出しが可能な上記以外の数学ルーチンについては、『数値計算ガイド』も参照してください。呼び出し可能な数学ルーチンには、`libm` や `libsunmath` の標準数学ライブラリルーチン (`Intro`(3M)参照)、これらのライブラリの最適化バージョン、SPARC ベクタ数学ライブラリ `libmvec` などがあります。

データ型について

特に指示がない限り、本章に記載する関数ルーチンは、組み込みルーチンではありません。

したがって、関数から返されるデータ型が、関数名だけを指定した場合に仮定されるデータ型と食い違う可能性がある場合は、ユーザーが明示的にデータ型を宣言する必要があります。たとえば、`getpid`() で `INTEGER*4` を戻す場合は、`INTEGER*4 getpid` と宣言しないと、結果の正しい処理が保証されません (データ型を明示的に指定しないと、関数名が `g` で開始するため、`REAL` (実数) 型の結果が仮定される)。なお、こういったルーチンについては、その機能要約で明示的な型宣言文が覚え書きの目的で記載されています。

引数および戻り値のデータ指定は、IMPLICIT 文および -r8、-i2、-dbl、-xtypemap といった各コンパイラオプションによっても変更されることに注意してください。これらのライブラリルーチンを呼び出す際に、期待するデータ型と実際のデータ型が一致していないと、プログラムは予期しない動きをします。 コンパイラオプション -r8、-dbl を指定すると、INTEGER 関数のデータ型は INTEGER*8 に、REAL 関数は REAL*8 に、DOUBLE 関数は DOUBLE*16 にそれぞれ変更されます。こういった問題を回避するには、ライブラリ呼び出しで指定する関数名と変数について、期待するそれらのサイズを明示的に指定する必要があります。次の例を参考にしてください。

```
integer*4 seed, getuid
real*4 ran
...
seed = 70198
val = getuid() + ran(seed)
...
```

上記の例のようにサイズを明示的に指定しておくと、コンパイラオプションとして -r8 と -dbl を指定しても、ライブラリ呼び出しの際にデータ型の変更が行われません。明示的な指定が行われない場合は、これらのコンパイラオプションによって、予期しない結果を招く可能性があります。これらのオプションの詳細については、『Fortran ユーザーズガイド』および f77(1) と f95(1) マニュアルページを参照してください。

旧オプションの -i2、-r8、-dbl の代わりに、より柔軟な -xtypemap オプションを使用してください。

Fortran コンパイラの広域プログラムチェックオプション -Xlist を使用すると、ライブラリコール全体のデータ型のミスマッチに関連した多数の問題を把握できます。f77 および f95 コンパイラによる広域プログラムチェックについては、『Fortran ユーザーズガイド』、『Fortran プログラミングガイド』、およびマニュアルページの f77(1) と f95(1) で説明しています。

64 ビット環境

プログラムを 64 ビットのオペレーティング環境で動作するようにコンパイルすると (つまり、-xarch=v9 または v9a を使ってコンパイルし、64 ビット Solaris 7 オペレーティング環境を実行する SPARC プラットフォーム上で実行可能プログラムを実行すること)、特定の関数の戻り値が変更されます。この特定の関数は、通常、malloc(3F) (74 ページ参照) などの標準システムレベルのルーチンとのインタフェースとなり、その環境に応じて 32 ビット値または 64 ビット値をとったり、戻したりできます。32 ビットと 64 ビット環境間でコードに互換性を持たせるために、これらのルーチンの 64 ビットバージョンは、必ず 64 ビット値をとるまたは戻す (あるいはこの両方を行う) ように規定されています。次の表に、64 ビット環境で使用するために提供されたライブラリルーチンを表示します。

表 1　64 ビット環境向けライブラリルーチン

ライブラリルーチン		
malloc64	メモリーを割り当て、ポインタを戻す	74 ページ
fseeko64	大規模ファイルの再位置付け	34 ページ
ftello64	大規模ファイルの位置付け	34 ページ
stat64, fstat64, lstat64	ファイルの状態を決定する	95 ページ
time64, ctime64, gmtine64, ltime64	システム時間を取得し、文字に変換するか月、日などに分解する	97 ページ
qsort64	配列の要素をソートする	81 ページ

Fortran 数学関数

次の関数とサブルーチンは、Fortran 数学ライブラリの一部です。これらの関数とサブルーチンは、f77 や f95 でコンパイルしたすべてのプログラムで使用することができます。ルーチンには、その引数と同じデータ型 (単精度、倍精度、または 4 倍精度) を戻す組み込み関数と、引数として特定のデータ型をとり、それと同じデータ型を戻す非組み込み関数があります。この非組み込み関数は、これを参照するルーチン内で宣言する必要があります。

こうしたルーチンの大半は、C 言語ライブラリのルーチンに対する Fortran のインタフェースである「ラッパー」であり、したがって、標準の Fortran ではありません。この中には、IEEE 推奨のサポート関数や特殊な乱数発生関数があります。これらのライブラリの詳細については、『数値計算ガイド』やマニュアルページ libm_single(3F)、libm_double(3F)、libm_quadruple(3F) を参照してください。

数学関数

以下に、数学関数をリストします。これらは、型宣言文内に入れる必要はありません。以下の関数は、引数として単精度、倍精度、または 4 倍精度データをとり、同じものを戻します。

sqrt(x)	asin(x)	cosd(x)
log(x)	acos(x)	asind(x)
log10(x)	atan(x)	acosd(x)
exp(x)	atan2(x,y)	atand(x)
x**y	sinh(x)	atan2d(x,y)
sin(x)	cosh(x)	aint(x)
cos(x)	tanh(x)	anint(x)
tan(x)	sind(x)	nint(x)

関数 sind(x)、cosd(x)、asind(x)、acosd(x)、atand(x)、atan2d(x,y) は、Fortran 標準の一部ではありません。

単精度関数

これらの副プログラムは、単精度の数学関数およびサブルーチンです。

通常、以下の数学単精度関数にアクセスする関数は、Fortran 規格の総称組み込み関数とは対応していません。データ型は通常の型決定規則によって決定されます。

デフォルトの型決定を保持している限り、REAL 文でこれらの関数の型を明示的に指定する必要はありません。(“ r ” で始まる変数は REAL 型、“ i ” で始まる変数は INTEGER 型になります)

これらのルーチンの詳細については、C 数学ライブラリのマニュアルページ (3M) を参照してください。たとえば、r_acos(x) の場合は、マニュアルページの acos(3M) を参照します。

表 2 数学単精度関数

r_acos(x)	REAL	関数	逆余弦
r_acosd(x)	REAL	関数	--
r_acosh(x)	REAL	関数	逆双曲余弦
r_acosp(x)	REAL	関数	--
r_acospi(x)	REAL	関数	--
r_atan(x)	REAL	関数	逆正接
r_atand(x)	REAL	関数	--
r_atanh(x)	REAL	関数	逆双曲正接
r_atanp(x)	REAL	関数	--
r_atanpi(x)	REAL	関数	--
r_asin(x)	REAL	関数	逆正弦
r_asind(x)	REAL	関数	--
r_asinh(x)	REAL	関数	逆双曲正弦
r_asinp(x)	REAL	関数	--
r_asinpi(x)	REAL	関数	--
r_atan2(y, x)	REAL	関数	逆正接
r_atan2d(y, x)	REAL	関数	--
r_atan2pi(y, x)	REAL	関数	--
r_cbrt(x)	REAL	関数	立方根
r_ceil(x)	REAL	関数	小数点以下切り上げ
r_copysign(x, y)	REAL	関数	--
r_cos(x)	REAL	関数	余弦
r_cosd(x)	REAL	関数	--
r_cosh(x)	REAL	関数	双曲余弦
r_cosp(x)	REAL	関数	--
r_cospi(x)	REAL	関数	--

表 2 数学単精度関数 (続き)

r_erf(x)	REAL	関数	誤差関数
r_erfc(x)	REAL	関数	--
r_expm1(x)	REAL	関数	(e**x)-1
r_floor(x)	REAL	関数	小数点以下切り捨て
r_hypot(x, y)	REAL	関数	斜辺
r_infinity( )	REAL	関数	--
r_j0(x)	REAL	関数	ベッセル関数
r_j1(x)	REAL	関数	--
r_jn(x)	REAL	関数	--
ir_finite(x)	INTEGER	関数	
ir_fp_class(x)	INTEGER	関数	
ir_ilogb(x)	INTEGER	関数	
ir_irint(x)	INTEGER	関数	
ir_isinf(x)	INTEGER	関数	
ir_isnan(x)	INTEGER	関数	
ir_isnormal(x)	INTEGER	関数	
ir_issubnormal(x)	INTEGER	関数	
ir_iszero(x)	INTEGER	関数	
ir_signbit(x)	INTEGER	関数	
r_addran()	REAL	関数	乱数発生関数
r_addrans(x, p, l, u)	n/a	サブルーチン	--
r_lcran()	REAL	関数	--
r_lcrans(x, p, l, u)	n/a	サブルーチン	--
r_shufrans(x, p, l, u)	n/a	サブルーチン	--
r_lgamma(x)	REAL	関数	ガンマ関数の対数
r_logb(x)	REAL	関数	--
r_log1p(x)	REAL	関数	--
r_log2(x)	REAL	関数	--
r_max_normal()	REAL	関数	
r_max_subnormal()	REAL	関数	
r_min_normal()	REAL	関数	
r_min_subnormal()	REAL	関数	
r_nextafter(x, y)	REAL	関数	
r_quiet_nan(n)	REAL	関数	
r_remainder(x, y)	REAL	関数	
r_rint(x)	REAL	関数	
r_scalb(x, y)	REAL	関数	
r_scalbn(x, n)	REAL	関数	
r_signaling_nan(n)	REAL	関数	
r_significand(x)	REAL	関数	

表 2 数学単精度関数 (続き)

r_sin(x)	REAL	関数	正弦
r_sind(x)	REAL	関数	--
r_sinh(x)	REAL	関数	双曲正弦
r_sinp(x)	REAL	関数	--
r_sinpi(x)	REAL	関数	--
r_sincos(x, s, c)	n/a	サブルーチン	正弦と余弦
r_sincosd(x, s, c)	n/a	サブルーチン	--
r_sincosp(x, s, c)	n/a	サブルーチン	--
r_sincospi(x, s, c)	n/a	サブルーチン	--
r_tan(x)	REAL	関数	正接
r_tand(x)	REAL	関数	--
r_tanh(x)	REAL	関数	双曲正接
r_tanp(x)	REAL	関数	--
r_tanpi(x)	REAL	関数	--
r_y0(x)	REAL	関数	ベッセル関数
r_y1(x)	REAL	関数	--
r_yn(n, x)	REAL	関数	--

- 変数 c、l、p、s、u、x、y は REAL 型です。
- IMPLICIT 文が有効で、"r" で始まる名前を別のデータ型に対して指定する場合、これらの関数を REAL として明示的に指定します。
- sind(x) や asind(x) などでは、ラジアンではなく度が使用されます。

参照: intro(3M)、『数値計算ガイド』

libm_double：倍精度関数

次の副プログラムは、倍精度の数学関数およびサブルーチンです。

通常、これらの関数は Fortran 規格の総称的な組み込み関数とは対応していません。データ型は、通常のデータ型決定規則によって決定されます。

これらの DOUBLE PRECISION 関数は DOUBLE PRECISION 文に指定する必要があります。

詳細については、C ライブラリのマニュアルページを参照してください。
d_acos(x) のマニュアルページは acos(3M) です。

表 3 数学倍精度関数

d_acos(x)	DOUBLE PRECISION	関数	逆余弦
d_acosd(x)	DOUBLE PRECISION	関数	--
d_acosh(x)	DOUBLE PRECISION	関数	逆双曲余弦
d_acosp(x)	DOUBLE PRECISION	関数	--
d_acospi(x)	DOUBLE PRECISION	関数	--
d_atan(x)	DOUBLE PRECISION	関数	逆正接
d_atand(x)	DOUBLE PRECISION	関数	--
d_atanh(x)	DOUBLE PRECISION	関数	逆双曲正接
d_atanp(x)	DOUBLE PRECISION	関数	--
d_atanpi(x)	DOUBLE PRECISION	関数	--
d_asin(x)	DOUBLE PRECISION	関数	逆正弦
d_asind(x)	DOUBLE PRECISION	関数	--
d_asinh(x)	DOUBLE PRECISION	関数	逆双曲正弦
d_asinp(x)	DOUBLE PRECISION	関数	--
d_asinpi(x)	DOUBLE PRECISION	関数	--
d_atan2(y, x)	DOUBLE PRECISION	関数	逆正接
d_atan2d(y, x)	DOUBLE PRECISION	関数	--
d_atan2pi(y, x)	DOUBLE PRECISION	関数	--
d_cbrt(x)	DOUBLE PRECISION	関数	立方根
d_ceil(x)	DOUBLE PRECISION	関数	小数点以下切り上げ
d_copysign(x, x)	DOUBLE PRECISION	関数	--
d_cos(x)	DOUBLE PRECISION	関数	余弦
d_cosd(x)	DOUBLE PRECISION	関数	--
d_cosh(x)	DOUBLE PRECISION	関数	双曲余弦
d_cosp(x)	DOUBLE PRECISION	関数	--
d_cospi(x)	DOUBLE PRECISION	関数	--
d_erf(x)	DOUBLE PRECISION	関数	誤差関数
d_erfc(x)	DOUBLE PRECISION	関数	--
d_expm1(x)	DOUBLE PRECISION	関数	(e**x)-1
d_floor(x)	DOUBLE PRECISION	関数	小数点以下切り捨て
d_hypot(x, y)	DOUBLE PRECISION	関数	斜辺
d_infinity( )	DOUBLE PRECISION	関数	--
d_j0(x)	DOUBLE PRECISION	関数	ベッセル関数
d_j1(x)	DOUBLE PRECISION	関数	--
d_jn(x)	DOUBLE PRECISION	関数	--

表 3 数学倍精度関数 (続き)

id_finite(x)	INTEGER	関数	
id_fp_class(x)	INTEGER	関数	
id_ilogb(x)	INTEGER	関数	
id_irint(x)	INTEGER	関数	
id_isinf(x)	INTEGER	関数	
id_isnan(x)	INTEGER	関数	
id_isnormal(x)	INTEGER	関数	
id_issubnormal(x)	INTEGER	関数	
id_iszero(x)	INTEGER	関数	
id_signbit(x)	INTEGER	関数	
d_addran()	DOUBLE PRECISION	関数	乱数
d_addrans(x, p, l, u)	n/a	サブルーチン	--
d_lcran()	DOUBLE PRECISION	関数	--
d_lcrans(x, p, l, u)	n/a	サブルーチン	--
d_shufrans(x, p, l, u)	n/a	サブルーチン	--
d_lgamma(x)	DOUBLE PRECISION	関数	ガンマ関数の対数
d_logb(x)	DOUBLE PRECISION	関数	--
d_log1p(x)	DOUBLE PRECISION	関数	--
d_log2(x)	DOUBLE PRECISION	関数	--
d_max_normal()	DOUBLE PRECISION	関数	
d_max_subnormal()	DOUBLE PRECISION	関数	
d_min_normal()	DOUBLE PRECISION	関数	
d_min_subnormal()	DOUBLE PRECISION	関数	
d_nextafter(x, y)	DOUBLE PRECISION	関数	
d_quiet_nan(n)	DOUBLE PRECISION	関数	
d_remainder(x, y)	DOUBLE PRECISION	関数	
d_rint(x)	DOUBLE PRECISION	関数	
d_scalb(x, y)	DOUBLE PRECISION	関数	
d_scalbn(x, n)	DOUBLE PRECISION	関数	
d_signaling_nan(n)	DOUBLE PRECISION	関数	
d_significand(x)	DOUBLE PRECISION	関数	
d_sin(x)	DOUBLE PRECISION	関数	正弦
d_sind(x)	DOUBLE PRECISION	関数	--
d_sinh(x)	DOUBLE PRECISION	関数	双曲正弦
d_sinp(x)	DOUBLE PRECISION	関数	--
d_sinpi(x)	DOUBLE PRECISION	関数	--
d_sincos(x, s, c)	n/a	サブルーチン	正弦と余弦
d_sincosd(x, s, c)	n/a	サブルーチン	--
d_sincosp(x, s, c)	n/a	サブルーチン	--
d_sincospi(x, s, c)	n/a	サブルーチン	--

表 3　数学倍精度関数 (続き)

d_tan(x)	DOUBLE PRECISION	関数	正接
d_tand(x)	DOUBLE PRECISION	関数	--
d_tanh(x)	DOUBLE PRECISION	関数	双曲正接
d_tanp(x)	DOUBLE PRECISION	関数	--
d_tanpi(x)	DOUBLE PRECISION	関数	--
d_y0(x)	DOUBLE PRECISION	関数	ベッセル関数
d_y1(x)	DOUBLE PRECISION	関数	--
d_yn(n,x)	DOUBLE PRECISION	関数	--

- 変数 c、l、p、s、u、x、y は DOUBLE PRECISION 型です。
- DOUBLE PRECISION 文に、または適当な IMPLICIT 文でこれらの関数の型を明示的に指定します。
- sind(x) や asind(x) などでは、ラジアンではなく度が使用されます。

参照 : intro(3M)、『数値計算ガイド』

4 倍精度関数

これらの副プログラムは、4 倍精度 (REAL*16) の数学関数およびサブルーチンです (*SPARC* のみ)。

通常、これらの関数は Fortran 規格の総称組み込み関数とは対応していません。データ型は通常の型決定規則によって決定されます。

4 倍精度関数は REAL*16 文に指定しなくてはなりません。

表 4　数学倍精度関数

q_copysign(x, y)	REAL*16	関数
q_fabs(x)	REAL*16	関数
q_fmod(x)	REAL*16	関数
q_infinity( )	REAL*16	関数
iq_finite(x)	INTEGER	関数
iq_fp_class(x)	INTEGER	関数
iq_ilogb(x)	INTEGER	関数
iq_isinf(x)	INTEGER	関数
iq_isnan(x)	INTEGER	関数
iq_isnormal(x)	INTEGER	関数
iq_issubnormal(x)	INTEGER	関数
iq_iszero(x)	INTEGER	関数
iq_signbit(x)	INTEGER	関数

表 4 数学倍精度関数 (続き)

q_max_normal()	REAL*16	関数
q_max_subnormal()	REAL*16	関数
q_min_normal()	REAL*16	関数
q_min_subnormal()	REAL*16	関数
q_nextafter(x, y)	REAL*16	関数
q_quiet_nan(n)	REAL*16	関数
q_remainder(x, y)	REAL*16	関数
q_scalbn(x, n)	REAL*16	関数
q_signaling_nan(n)	REAL*16	関数

- 変数 c、l、p、s、u、x、y は 4 倍精度です。
- REAL*16 文または適当な IMPLICIT 文でこれらの関数の型を明示的に指定します。
- sind(x) や asind(x) などでは、ラジアンではなく度が使用されます。

その他の 4 倍精度 libm 関数を使用する必要がある場合、その呼び出しの前に $PRAGMA C <関数名> を使用してください。詳細については、『Fortran プログラミングガイド』の 第 11 章「C と Fortran のインタフェース」を参照してください。

abort：終了とコアファイルへの書き込み

サブルーチンは、次のように呼び出します。

```
call abort
```

abort は、入出力バッファをフラッシュ (バッファ内のデータを実際にファイルに書き込むこと) し、現在のディレクトリにコアファイルのメモリーダンプを作成して、処理を異常終了させます。コアダンプを制限、または行わないようにする方法については、limit(1) を参照してください。

access： ファイルのアクセス権または有無の検査

関数は、次のように呼び出します。

INTEGER*4 access status = access (*name, mode*)			
name	CHARACTER	入力	ファイル名
mode	CHARACTER	入力	アクセス権
戻り値	INTEGER*4	出力	*status*=0: 正常 *status*>0: エラーコード

access は、*name* で指定したファイルに *mode* で指定したアクセス権でアクセスできるかどうかを決定します。*mode* で指定したアクセスが正常終了した場合は、ゼロが返されます。

エラーコードを解釈する場合は、gerror(3F) も参照してください。

mode には、r、w、x を単独で指定することも、任意の順序で 2 つ以上組み合わせて指定することも、あるいは空白を指定することもできます。 r、w、x の意味はそれぞれ以下のとおりです。

'r'	読み取りアクセス権をテストする
'w'	書き込みアクセス権をテストする
'x'	実行アクセス権をテストする
'空白'	ファイルの有無をテストする

例 1：読み取りおよび書き込みに関するアクセス権のテスト

```
    INTEGER*4  access, status
    status = access ( 'taccess.data', 'rw' )
    if ( status .eq. 0 ) write(*,*) "ok"
    if ( status .ne. 0 ) write(*,*) '読み/書き不可', status
```

例 2：ファイルの有無のテスト

```
    INTEGER*4  access, status
    status = access ( 'taccess.data', ' ' )! 空白モード
    if ( status .eq. 0 ) write(*,*) "ファイル存在"
    if ( status .ne. 0 ) write(*,*) 'ファイルはない', status
```

alarm：指定時間後のサブルーチンの呼び出し

関数は、次のように呼び出します。

INTEGER*4 alarm n = alarm (*time, sbrtn*)			
time	INTEGER*4	入力	待ち時間の秒数 (0 の場合は呼び出さない)
sbrtn	ルーチン名	入力	実行する副プログラムは EXTERNAL 文で宣言しなければならない
戻り値	INTEGER*4	出力	前回呼び出した alarm の残り時間

alarm の使用例: 9 秒待機してから sbrtn を呼び出します。

```
      integer*4 alarm, time / 1 /
      common / alarmcom / i
      external sbrtn
      i = 9
      write(*,*) i
      nseconds =  alarm ( time, sbrtn )
      do n = 1,100000      ! alarm が sbrtn をアクティブにするまで待機
      r = n      ! (時間に余裕のある計算)
      x=sqrt(r)
      end do
      write(*,*) i
      end
      subroutine sbrtn
      common / alarmcom / i
      i = 3   ! このルーチンでは I/O を行わないでください
      return
      end
```

参照 : alarm(3C)、sleep(3F)、signal(3F) 以下の制限事項に注意してください。

- サブルーチンは自分自身の名前を alarm に渡すことはできません。
- alarm ルーチンは、入出力に干渉する可能性のあるシグナルを発生させます。呼び出されたサブルーチン (*sbrtn*) では、いっさい入出力を実行してはなりません。
- Fortran の並列プログラムまたはマルチスレッドプログラムから alarm() を呼び出すと、予期しない結果を招くことがあります。

bit: ビット関数

定義は以下のとおりです。

and(*word1, word2*)	引数のビット単位の論理積を計算する
or(*word1, word2*)	引数のビット単位の論理和を計算する
xor(*word1, word2*)	引数のビット単位の排他的論理和を計算する
not(*word*)	引数のビット単位の補数を戻す
lshift(*word, nbits*)	循環桁上げなしで左へ論理シフトする
rshift(*word, nbits*)	符号拡張を行い右へ算術シフトする
call bis(*bitnum, word*)	*word* の第 *bitnum* ビットを 1 に設定する
call bic(*bitnum, word*)	*word* の第 *bitnum* ビットを 0 にクリアーする
bit(*bitnum, word*)	*word* の第 *bitnum* ビットを検査し、ビットが 1 であれば .true. を返しビットが 0 であれば .false. を戻す
call setbit(*bitnum, word, state*)	*state* がゼロ以外であれば *word* の第 *bitnum* ビットを 1 に設定し、*state* がゼロであれば 0 にクリアーする

MIL-STD-1753 の代替外部バージョンは以下のとおりです。

iand(*m, n*)	引数のビット単位の論理積を計算する
ior(*m, n*)	引数のビット単位の論理和を計算する
ieor(*m, n*)	引数のビット単位の排他的論理和を計算する
ishft(*m, k*)	循環桁上げなしで論理シフトする (*k*>0 のときは左、*k*<0 のときは右へ)
ishftc(*m, k, ic*)	循環シフト: *m* の、右から *ic* ビットを左へ *k* ビット循環シフトする
ibits(*m, i, len*)	ビットの切り出し: *i* ビット目から始まる *len* ビット分を *m* から切り出す

ibset(*m, i*)	ビットをセットする: ビット *i* が 1 であれば戻り値は *m* と同じです
ibclr(*m, i*)	ビットをクリアーする: ビット *i* が 0 であれば戻り値は *m* と同じです
btest(*m, i*)	ビットのテスト: *m* の *i* 番目のビットをテストする。ビットが 1 のときは .true. を返し、ビットが 0 のときは .false. を戻す

75 ページの「mvbits : ビットフィールドの移動」と『FORTRAN 77 言語リファレンス』 の第 6 章「組み込み関数」も参照してください。

and、or、xor、not、rshift、lshift の使用法

組み込み関数の場合は、次のように使います。

```
x = and( word1, word2 )
x = or( word1, word2 )
x = xor( word1, word2 )
x = not( word )
x = rshift( word, nbits )
x = lshift( word, nbits )
```

word、*word1*、*word2*、および *nbits* は、整数型の入力引数です。これらは組み込み関数で、コンパイラによりインライン展開されます。戻されるデータの型は、第 1 引数のデータ型です。

nbits の値が正当かどうかの検査は行われません。

例 1： and、or、xor、not

```
demo% cat tandornot.f
    print 1, and(7,4), or(7,4), xor(7,4), not(4)
 1  format(4x 'and(7,4)', 5x 'or(7,4)', 4x 'xor(7,4)',
&     6x 'not(4)'/4o12.11)
    end
demo% f77 -silent tandornot.f
demo% a.out
    and(7,4)    or(7,4)    xor(7,4)       not(4)
 00000000004 00000000007 00000000003 37777777773
demo%
```

例 2： lshift、rshift

```
    integer*4 lshift, rshift
    print 1, lshift(7,1), rshift(4,1)
 1  format(1x 'lshift(7,1)', 1x 'rshift(4,1)'/2o12.11)
    end
demo% f77 -silent tlrshift.f
demo% a.out
 lshift(7,1) rshift(4,1)
 00000000016 00000000002
demo%
```

bic、bis、bit、setbit の使用法

```
call bic( bitnum, word )

call bis( bitnum, word )

call setbit( bitnum, word, state )

LOGICAL bit
x = bit( bitnum, word )
```

bitnum、*state*、および *word* は、INTEGER*4 型の入力引数です。bit() 関数では、論理値が返されます。

ビットは、ビット 0 が最下位ビット、ビット 31 が最上位ビットになるように番号が付けられます。

bic、bis および setbit は外部サブルーチン、bit は外部関数です。

例 3 : bic、bis、setbit、bit

```
    integer*4 bitnum/2/, state/0/, word/7/
    logical bit
    print 1, word
 1  format(13x 'ワード', o12.11)
    call bic( bitnum, word )
    print 2, word
 2  format('bic(2,word)の後', o12.11)
    call bis( bitnum, word )
    print 3, word
 3  format('bis(2,word)の後', o12.11)
    call setbit( bitnum, word, state )
    print 4, word
 4  format('setbit(2,word,0)の後', o12.11)
    print 5, bit(bitnum, word)
 5  format('bit(2,word)', L )
    end
<出力>
          ワード 00000000007
bic(2,word)の後 00000000003
bis(2,word)の後 00000000007
setbit(2,word,0)の後 00000000003
bit(2,word) F
```

chdir：デフォルトディレクトリの変更

関数は、次のように呼び出します。

INTEGER*4 chdir *n* = chdir(*dirname*)			
dirname	CHARACTER	入力	ディレクトリ名
戻り値	INTEGER*4	出力	*n*=0: 正常、 *n*>0: エラーコード

例 : chdir — 現在の作業ディレクトリを MyDir に変更します。

```
    INTEGER*4  chdir, n
    n =  chdir ( 'MyDir' )
    if ( n .ne. 0 ) stop 'chdir: エラー'
    end
```

参照 : chdir(2)、cd(1)、gerror(3F) (エラーコードの解釈)

パス名は、 <sys/param.h> で定義されている MAXPATHLEN より長くすることはできません。相対パス名でも、絶対パス名でもかまいません。

この関数を使用すると装置による照会が失敗する場合があります。

いくつかの Fortran のファイル操作は、ファイルを名前で再オープンします。入出力動作中に chdir を使用すると、実行時システムが相対パス名で作成されたファイル (ファイル名を指定せずに open 文で作成されたファイルを含む) を見失ってしまうことがあります。

chmod：ファイルのモードの変更

関数は、次のように呼び出します。

INTEGER*4 chmod *n* = chmod(*name*, *mode*)			
name	CHARACTER	入力	パス名
mode	CHARACTER	入力	chmod(1) に認識されるモード (o-w、444 など)
戻り値	INTEGER*4	出力	*n*= 0: 正常、 *n*> 0: システムエラー番号

例 : chmod — 書き込み権を MyFile に追加します。

```
character*18 name, mode
INTEGER*4 chmod, n
name = 'MyFile'
mode = '+w'
n =  chmod( name, mode )
if ( n .ne. 0 ) stop 'chmod: エラー'
end
```

参照：chmod(1)、gerror(3F) (エラーコードの解釈)

パス名は、<sys/param.h> で定義されている MAXPATHLEN より長くすることはできません。

date：文字列として現在のデータを取得

注 – このルーチンは年を示す場合に 2 桁の値しか返さないので、「2000 年には無効」 になります。このルーチンの出力を使用して日付間の差を計算するプログラムは、1999 年12 月 31 日以降は正しく機能しなくなります。この date( ) ルーチンを使用しているプログラムは、ルーチンの初期呼び出し時に実行時警告メッセージを表示してユーザーに警告します。このルーチンの代わりに呼びだすことのできるルーチンとして、data_and_time() を参照して下さい。

サブルーチンは、次のように呼び出します。

call date(*c*)			
c	CHARACTER*9	出力	変数、配列、配列要素、あるいは部分列

戻される文字列 c の形式は、*dd-mmm-yy* です。ここで、*dd* は 2 桁の数値で表した日、*mmm* は 3 文字に省略した英語の月名、*yy* は 2 桁の数値で表した年 (2000年には対応していません) です。

例： date

```
demo% cat dat1.f
* dat1.f — 日付けを文字列として取得
      character c*9
      call date ( c )
      write(*,"(' 本日の日付けは、 ', A9 )" ) c
      end
demo% f77 -silent dat1.f
 "dat.f", line 2 : 警告 : サブルーチン "date" は 2000 年以降は安全ではありません。
        代わりに "date_and_time" を使用してください。
demo% a.out
 西暦 2000 年以降は、サブルーチン idate の 2 桁の年を使用して時間差を計算するのは安全ではありません。
 日付けは、23-Sep-96
demo%
```

47 ページの「idate : 現在の日付を戻す」と、22 ページの「date_and_time : 日付と時刻の取得」も参照してください。

date_and_time : 日付と時刻の取得

これは、Fortran 95 組み込みルーチンの FORTRAN 77 バージョンで、2000 年以降も有効です。

DATE_AND_TIME サブルーチンはリアルタイムクロックと日付のデータを返します。現地時間の他に、現地時間と世界標準時 (UTC: Universal Coordinated Time)(グリニッジ平均時 (GMT: Greenwich Mean Time) とも呼ぶ) の時差も返します。

date_and_time( ) サブルーチンは、次のように呼び出します。

call date_and_time(*date, time, zone, values*)			
date	CHARACTER*8	出力	日付。書式は CCYYMMDD。CCYY は 4 桁の年、MM は 2 桁の月、DD は 2 桁の日。例 : 19980709
time	CHARACTER*10	出力	現在の時刻。書式は hhmmss.sss。hh は時、mm は分、ss.sss は秒とミリ秒
zone	CHARACTER*5	出力	UTC を使用した場合の時差。時分で示す。書式は hhmm
values	INTEGER*4 VALUES(8)	出力	以下で説明する 8 要素の整数配列

INTEGER*4 values に返される 8 つの値は次のとおりです。

VALUES(1)	4 桁の整数の年。たとえば、1998。
VALUES(2)	1〜12 の整数の月。
VALUES(3)	1〜31 の整数の日。
VALUES(4)	UTC を使用した場合の時差 (分)。
VALUES(5)	1〜23 の整数の時。
VALUES(6)	1〜59 の整数の分。
VALUES(7)	0〜60 の整数の秒。

VALUES(8)　　0〜999 の範囲のミリ秒。

date_and_time の使用例 :

```
demo% cat dtm.f
       integer date_time(8)
       character*10 b(3)
       call date_and_time(b(1), b(2), b(3), date_time)
       print *,'date_timearray 値'
       print *,'年=',date_time(1)
       print *,'month_of_year=',date_time(2)
       print *,'day_of_month=',date_time(3)
       print *,'時差(分)=',date_time(4)
       print *,'時=',date_time(5)
       print *,'分=',date_time(6)
       print *,'秒=',date_time(7)
       print *,'ミリ秒=',date_time(8)
       print *,'DATE=',b(1)
       print *,'TIME=',b(2)
       print *,'ZONE=',b(3)
       end
```

2000 年 2 月 16 日にカリフォルニアで実行した場合の出力は次のとおりです。

```
 date_time 配列値 :
年 = 2000
month_of_year= 2
day_of_month= 16
時差（分） = -420
時 = 11
分 = 49
秒 = 29
ミリ秒 = 236
DATE=20000216
TIME=114929.236
ZONE=-0700
```

dtime、etime：経過実行時間

これらの 2 つの関数は、経過実行時間 (あるいはエラー指示子として -1.0) を返します。返される時間は秒単位です。

Fortran 77 が使用する dtime と etime のバージョンは、実行時のシステムの高分解能クロックによって生成された時刻を返します。実際の分解能はシステムのプラットフォームによって違います。現在のプラットフォームのクロックの分解能は、1 ナノ秒から 1 マイクロ秒の範囲内です。

Fortran 95 が使用する `dtime` と `etime` のバージョンは、デフォルトではシステムの低分解能クロックを使用します。分解能は 100 分の 1 秒です。ただし、プログラムが Sun OS™ オペレーティングシステムのユーティリティ `ptime`(1)、(`/usr/proc/bin/ptime`) の下で実行された場合は、高分解能クロックが使用されます。

dtime： 前回の dtime 呼び出しからの経過時間

`dtime` の場合、経過時間は次のとおりです。

- 最初の呼び出し：実行開始からの経過時間
- 2 回目以降の呼び出し：前回の `dtime` の呼び出しからの経過時間
- シングルプロセッサ：CPU の使用時間
- マルチプロセッサ：すべての CPU 使用合計時間 (あまり便利ではないので、`etime` を使用してください)

注 – 並列化ループ内から `dtime` を呼び出すと、決定性のない結果になります。経過時間カウンタが、ループに関与しているすべてのスレッドに対してグローバルであるためです。

関数は、次のように呼び出します。

e = dtime(*tarray*)			
tarray	REAL(2)	出力	*e*= -1.0: エラー: *tarray* 値は未定義 *e*≠ -1.0: *tarray*(1) にユーザー時間 (エラーがない場合) *tarray*(2) にシステム時間 (エラーがない場合
戻り値	REAL	出力	*e*= -1.0: エラー *e*≠ -1.0: *tarray*(1) と *tarray*(2) の合計時間

例 : dtime()、シングルプロセッサ

```
    real e, dtime, t(2)
    print *, '経過:', e, ', ユーザー:', t(1), ', システム:', t(2)
    do i = 1, 10000
    k=k+1
    end do
    e = dtime( t )
    print *, '経過:', e, ', ユーザー:', t(1), ', システム:', t(2)
    end
demo% f77 -silent tdtime.f
demo% a.out
経過:  0., ユーザー:  0., システム:  0.
経過:   0.180000, ユーザー:    6.00000E-02, システム:   0.120000
demo%
```

etime： 実行開始からの経過時間

etime の場合、経過時間は次のとおりです。

- シングルプロセッサ：呼び出したプロセスの CPU 時間
- マルチプロセッサ：プログラムを処理している間の実時間

Fortran は、シングルプロセッサとマルチプロセッサを以下のように区別します。

libF77_mt とリンクした並列 Fortran プログラムでは、環境変数が PARALLEL の場合、次のようになります。

- 未定義の場合、現在の実行はシングルプロセッサです。

- 定義済みで範囲 1,2,3,... の場合、現在の実行はマルチプロセッサです。
- 定義済みで 1,2,3,... 以外の値の場合、結果は不定です。

関数は、次のように呼び出します。

e = etime(*tarray*)			
tarray	REAL(2)	出力	*e*= -1.0: エラー: *tarray* の値は未定義 *e*≠ -1.0: シングルプロセッサ:*tarray*(1) にユーザー時間 *tarray*(2) にシステム時間 マルチプロセッサ:*tarray*(1) に実時間 *tarray*(2) に 0.0
戻り値	REAL	出力	*e*= -1.0: エラー *e*≠ -1.0: *tarray*(1) と *tarray*(2) の合計時間

etime の初期呼び出しで返される結果は不正確です。初期呼び出しでは、単にシステムクロックを稼働させるだけなので、etime の初期呼び出しで返された値は使用しないでください。

例 : etime() — シングルプロセッサ

```
    real e, etime, t(2)
    e = etime(t)          !  Startup etime - do not use result
    do i = 1, 10000
    k=k+1
    end do
    e = etime( t )
    print *, 'elapsed:', e, ', user:', t(1), ', sys:', t(2)
    end
demo% f77 -silent tetime.f
demo% a.out
elapsed:   0.190000, user:    6.00000E-02, sys:   0.130000
demo%
```

times(2)、f77(1) のマニュアルページ、および『Fortran プログラミングガイド』も参照してください。

exit： プロセスの終了および状態の設定

サブルーチンは、次のように呼び出します。

call exit(*status*)		
status	INTEGER*4	入力

例： exit()

```
    ...
    if(dx .lt. 0.) call exit( 0 )
    ...
    end
```

exit はフラッシュしてからプロセスのすべてのファイルを閉じ、その親プロセスが wait を実行している場合は親プロセスに通知します。

親プロセスは *status* の下位 8 ビットを使用できます。この 8 ビットは左に 8 ビットシフトされ、他のビットはすべてゼロになります (したがって *status* は 256 〜 65280 の範囲になります)。この呼び出しは復帰しません。

C の関数である exit は、最終的なシステム終了動作が実行される前に整理の処理を行うことがあります。

引数なしで exit を呼び出すとコンパイル時警告メッセージが出され、自動的に引数にゼロが与えられます。
参照 : exit(2)、fork(2)、fork(3f)、wait(2)、wait(3f)

fdate： ASCII 文字列で日付および時刻を戻す

サブルーチンまたは関数は、次のように呼び出します。

call fdate(*string*)		
string	CHARACTER*24	出力

または以下のとおりです。

CHARACTER fdate*24 *string* = fdate()			fdate を関数として使用する場合、それを呼び出すルーチンは fdate の型と長さを定義する必要がある
戻り値	CHARACTER*24	出力	

例 1 : サブルーチンとしての使用

```
    character*24 string
    call fdate( string )
    write(*,*) string
    end
```

上記の例の出力は次のようになります。

```
Wed Aug 3 15:30:23 1994
```

例 2 : 関数としての使用。出力は上記の例と同じ。

```
    character*24 fdate
    write(*,*)  fdate()
    end
```

参照 : ctime(3)、time(3F)、idate(3F)

flush：論理装置への出力のフラッシュ

関数は、次のように呼び出します。

INTEGER*4 flush *n* = flush(*lunit*)			
lunit	INTEGER*4	Input	論理装置
Return value	INTEGER*4	Output	n = 0 no error n > 0 error number

flush 関数は、論理装置 lunit に対するバッファの内容を結合されているファイルにフラッシュします。このサブルーチンが最も役に立つのは、論理装置 0 と 6 がどちらもコンソールに結合されていて、それらの装置に対してこのサブルーチンを使用する場合です。

関数はエラーが発生すると、正のエラー番号を返し、エラーが発生しないとゼロを返します。

参照 : fclose(3S)

fork：現プロセスのコピーの生成

関数は、次のように呼び出します。

INTEGER*4 fork *n* = fork()			
戻り値	INTEGER*4	出力	*n*>0: *n*= コピーのプロセス識別子 *n*<0, *n*= (システムエラーコード)

fork 関数はそれを呼び出したプロセスのコピーを生成します。元のプロセスとコピーとの違いは、元のプロセス (親プロセスと呼ばれる) に返される値がコピーのプロセス識別子であるということだけです。コピーは一般に子プロセスと呼ばれます。子プロセスに返される値はゼロです。

書き込み用に開いているすべての論理装置は、fork が実行される前にフラッシュされます。これは入出力バッファの内容が外部ファイルに重複して書き込まれるのを防ぎます。

例 : fork()

```
    INTEGER*4 fork, pid
    pid = fork()
    if(pid.lt.0) stop 'フォーク失敗'
    if(pid.gt.0) then
        print *, '親プロセス'
    else
        print *, '子プロセス'
    endif
```

fork ルーチンと対をなす exec ルーチンは提供されていません。これは論理装置を開いたままで exec ルーチンに渡せる良い方法がないためです。ただし、system(3F) を使用すれば fork/exec の通常の機能を実行することができます。

参照 : fork(2)、wait(3F)、kill(3F)、system(3F)、perror(3F)

free : Malloc により割り当てられた記憶領域の割り当て解除

サブルーチンは、次のように呼び出します。

call free (*ptr*)		
ptr	pointer	入力

free は malloc により先に割り当てられた記憶領域の割り当てを解除します。記憶領域はメモリーマネージャに返されますが、ユーザーのプログラムの中でこれを使用することはできなくなります。

例： free()

```
real x
pointer ( ptr, x )
ptr = malloc ( 10000 )
call free ( ptr )
end
```

詳細については、74 ページの「malloc、malloc 64: 記憶領域の割り当てとそのアドレスの獲得」を参照してください。

fseek、ftell： ファイルのポインタの位置付けと再位置付け

fseek および ftell は、ファイルの再位置付けを可能にするルーチンです。ftell は、ファイルの現在位置をファイルの先頭からのオフセットを示すバイト数で返します。プログラムの後方で、この値を使用して fseek を呼ぶことにより、ファイルの読み込み位置を元に戻すことができます。

fseek：論理装置上のファイルのポインタの再位置付け

関数は、次のように呼び出します。

<table>
<tr><td colspan="4">INTEGER*4 fseek
n = fseek(lunit, offset, from)</td></tr>
<tr><td>lunit</td><td>INTEGER*4</td><td>入力</td><td>開いている論理装置</td></tr>
<tr><td rowspan="2">offset</td><td rowspan="2">INTEGER*4
または
INTEGER*8</td><td>入力</td><td>from で指定された位置からのオフセットを示すバイト数</td></tr>
<tr><td colspan="2">-xarch=v9 を使って、Solaris 7 などの 64 ビット環境用にコンパイルする場合は、INTEGER*8 オフセット値が必要。定数を入力する場合は、それを 64 ビット定数にする必要がある。たとえば、100_8</td></tr>
</table>

INTEGER*4 fseek *n* = fseek(*lunit*, *offset*, *from*)			
from	INTEGER*4	入力	0= ファイルの先頭 1= 現在の位置 2= ファイルの終了
戻り値	INTEGER*4	出力	*n*=0: 正常。 *n*>0: システムエラーコード

注 – 順編成ファイルでは、fseek 64 に続く呼び出しの後の出力操作 (WRITE など) は、fseek の位置に続くすべてのデータレコードの削除、新しいデータレコード (とファイルの終わりのマーク) での書き換えの原因となります。正しい位置へのレコードの書き換えは、直接アクセスファイルでのみ実行可能です。

例 : fseek() — MyFile のポインタを先頭から 2 バイトの位置に再位置付けします。

```
INTEGER*4 fseek, lunit/1/, offset/2/, from/0/, n
open( UNIT=lunit, FILE='MyFile' )
n = fseek( lunit, offset, from )
if ( n .gt. 0 ) stop 'fseek エラー'
end
```

例 : 上記の例を 64 ビット環境で、-xarch=v9 を使ってコンパイルすると次のようになります。

```
INTEGER*4 fseek, lunit/1/, from/0/, n
INTEGER*8 offset/2/
open( UNIT=lunit, FILE='MyFile' )
n = fseek( lunit, offset, from )
if ( n .gt. 0 ) stop 'fseek error'
end
```

ftell：ファイルの現在位置を戻す

関数は、次のように呼び出します。

INTEGER*4 ftell *n* = ftell(*lunit*)			
lunit	INTEGER*4	入力	開いている論理装置
戻り値	INTEGER*4 または INTEGER*8	出力	*n*>=0: *n*= ファイルの先頭からのオフセットを示すバイト数 *n*<0: *n*= システムエラーコード
	-xarch=v9 を使って、Solaris 7 などの 64 ビット環境用にコンパイルすると、INTEGER*8 オフセット値が戻る。ftell とこの戻り値を受け取る変数は、INTEGER*8 と宣言する必要がある。		

例 : ftell()

```
INTEGER*4 ftell, lunit/1/, n
open( UNIT=lunit, FILE='MyFile' )
...
n = ftell( lunit )
if ( n .lt. 0 ) stop 'ftell エラー'
...
```

例 : 上記の例を 64 ビット環境で、-xarch=v9 を使ってコンパイルすると次のようになります。

```
INTEGER*4 lunit/1/
INTEGER*8 ftell, n
open( UNIT=lunit, FILE='MyFile' )
...
n = ftell( lunit )
if ( n .lt. 0 ) stop 'ftell error'
...
```

参照 : fseek(3S)、perror(3F)、fseeko64(3f)、ftello64(3f)

fseeko64、ftello64: 大規模ファイルのポインタの位置付けと再位置付け

fseeko64 と ftello64 は、それぞれ fseek と ftell の 「大規模ファイル」 バージョンです。fseeko64 と ftello64 は、Solaris 2.6 と 7 で INTEGER*8 ファイル位置のオフセットを入出力します。(「大規模ファイル」 とは 2G バイトを超えるファイルのことで、バイト位置は 64 ビットの整数で示します。) これらのバージョンを使用して、大規模ファイルのポインタの位置付けや再位置付けを行います。

fseeko64: 論理装置上のファイルのポインタの再位置付け

関数は、次のように呼び出します。

INTEGER fseeko64 *n* = fseeko64(*lunit, offset64, from*)			
lunit	INTEGER*4	入力	開いている論理装置
offset64	INTEGER*8	入力	*from* で指定された位置からの 64 ビット オフセットを示すバイト数
from	INTEGER*4	入力	0=ファイルの先頭 1=現在の位置 2=ファイルの終了
戻り値	INTEGER*4	出力	*n*=0: 正常。*n*>0: システムエラーコード

注 – 順編成ファイルでは、fseek 64 に続く呼び出しの後の出力操作 (WRITE など) は、fseek の位置に続くすべてのデータレコードの削除、新しいデータレコード (とファイルの終わりのマーク) での書き換えの原因となります。正しい位置へのレコードの書き換えは、直接アクセスファイルでのみ実行可能です。

例 : fseeko64( ) - MyFile のポインタを先頭から 2 バイトの位置に再位置付けします。

```
INTEGER fseeko64, lunit/1/, from/0/, n
INTEGER*8 offset/200/
open( UNIT=lunit, FILE='MyFile' )
n = fseeko64( lunit, offset, from )
if ( n .gt. 0 ) stop fseek エラー'
end
```

ftello64: ファイルの現在位置を戻す

関数は、次のように呼び出します。

INTEGER*8 ftello64 *n* = ftello64(*lunit*)			
lunit	INTEGER*4	入力	開いている論理装置
戻り値	INTEGER*8	出力	*n*≥0: *n*=ファイルの先頭からのオフセットを示すバイト数 *n*<0: *n*=システムエラーコード

例 : ftello64( ) :

```
INTEGER*8 ftello64, lunit/1/, n
open( UNIT=lunit, FILE='MyFile' )
...
n = ftello64( lunit )
if ( n .lt. 0 ) stop 'ftell エラー'
...
```

getarg、iargc： コマンド行の引数の取得

getarg と iargc は、コマンド行プリプロセッサによって展開されたコマンド行引数にアクセスします。

getarg： コマンド行の引数の取得

サブルーチン は、次のように呼び出します。

call getarg(*k*, *arg*)			
k	INTEGER*4	入力	引数の索引 (0 = 最初の引数 = コマンド名)
arg	CHARACTER**n*	出力	*k* 番目の引数
n	INTEGER*4	引数のサイズ	最も長い引数が入るだけの大きさ

iargc： コマンド行の引数の個数の取得

関数は、次のように呼び出します。

m = iargc()			
戻り値	INTEGER*4	出力	コマンド行の引数の個数

例 : iargc と getarg: 引数の個数を調べ、各引数を読み取ります。

```
demo% cat yarg.f
    character argv*10
    INTEGER*4 i, iargc, n
    n = iargc()
    do 1 i = 1, n
      call getarg( i, argv )
 1    write( *, '( i2, 1x, a )' ) i, argv
    end
demo% f77 -silent yarg.f
demo% a.out *.f
1 first.f
2 yarg.f
```

参照 : execve(2)、getenv(3F)

getc、fgetc : 次の文字の取得

getc と fgetc は、入力ストリームから次の文字を読み取ります。同じ論理装置上では、これらのルーチンの呼び出しを通常の Fortran の入出力と混合して使用しないでください。

getc : 標準入力からの次の文字の取得

関数は、次のように呼び出します。

INTEGER*4 getc *status* = getc(*char*)			
char	CHARACTER	出力	次の文字
戻り値	INTEGER*4	出力	*status*=0: 正常 *status*=-1: ファイルの終了 *status*>0: システムエラーコードまたは f77 入出力エラーコード

例 : getc でキーボードから文字を 1 文字ずつ入力します。Control-D (^D) に注意してください。

```
      character char
      INTEGER*4 getc, status
      status = 0
      do while ( status .eq. 0 )
      status = getc( char )
      write(*, '(i3, o4.3)') status, char
      end do
      end
```

上記のソースプログラムを (コンパイル後に) 実行した例を以下に示します。

```
demo% a.out
ab              ← プログラムが入力された文字を読み取る。
^D              ←      Control-D キーで終了された。
0 141           ← プログラムが入力された文字の状態コードと 8 進値を表す。
0 142           ←      141 は 'a' を、142 は 'b' を表す。
0 012           ←      012 はリターンキーを表す。
-1 012          ← 次の読み取りが試行され Control-D を戻された。
demo%
```

どの論理装置に対しても、通常の Fortran の入力と getc() を混在して使用しないでください。

fgetc : 指定した論理装置からの次の文字の取得

関数は、次のように呼び出します。

INTEGER*4 fgets *status* = fgetc(*lunit*, *char*)			
lunit	INTEGER*4	入力	論理装置
char	CHARACTER	出力	次の文字

INTEGER*4 fgets *status* = fgetc(*lunit*, *char*)			
戻り値	INTEGER*4	出力	*status*=-1: ファイルの終了 *status*>0: システムエラーコードまたは f77 入出力エラーコード

例 : fgetc で tfgetc.data から文字を 1 文字ずつ読み取ります。改行 (8 進の 012) に注意してください。

```
    character char
    INTEGER*4 fgetc, status
    open( unit=1, file='tfgetc.data' )
    status = 0
    do while ( status .eq. 0 )
    status = fgetc( 1, char )
    write(*, '(i3, o4.3)') status, char
    end do
    end
```

上記のソースプログラムを (コンパイル後に) 実行した例を以下に示します。

```
demo% cat tfgetc.data
ab
yz
demo% a.out
0 141        ← 'a' が読み取られる。
0 142        ←  'b' が読み取られる。
0 012        ← 改行が読み取られる。
0 171        ← 'y' が読み取られる。
0 172        ← 'z' が読み取られる。
0 012        ← 改行が読み取られる。
-1 012       ← Control-D が読み取られる。
demo%
```

どの論理装置に対しても、通常の Fortran の入力と fgetc() を混在して使用しないでください。

参照 : getc(3S)、intro(2)、perror(3F)

getcwd : 現在のディレクトリパスの取得

関数は、次のように呼び出します。

INTEGER*4 getcwd *status* = getcwd(*dirname*)			
dirname	CHARACTER**n*	出力 現在のディレクトリのパスが返される。	現在のディレクトリのパス名。 *n* は、最も長いパス名が入るのに十分な大きさであることが必要
戻り値	INTEGER*4	出力	*status*=0: 正常 *status*>0: エラーコード

例 : getcwd

```
INTEGER*4 getcwd, status
character*64 dirname
status = getcwd( dirname )
if ( status .ne. 0 ) stop 'getcwd: エラー'
write(*,*) dirname
end
```

参照 : chdir(3F)、perror(3F)、getwd(3)

注 – パス名を <sys/param.h> で定義されている MAXPATHLEN より長くすることはできません。

getenv：環境変数の値の取得

サブルーチンは、次のように呼び出します。

call getenv(*ename, evalue*)			
ename	CHARACTER*n	入力	検索する環境変数の名前
evalue	CHARACTER*n	出力	見つかった環境変数の値。 見つからなかった場合は空

ename と *evalue* には、それぞれの文字列が十分入るだけの大きさが必要です。

getenv サブルーチンは環境リストから *ename=evalue* の形式の文字列を検索し、その文字列があった場合には *evalue* の値を返し、なかった場合には *evalue* に空白を詰めます。

例 : $SHELL の値を印刷するには、getenv() を使用します。

```
    character*18  evalue
    call getenv( 'SHELL', evalue )
    write(*,*) "'", evalue, "'"
    end
```

参照 : execve(2)、environ(5)

getfd：外部装置番号に対するファイル記述子の取得

関数は、次のように呼び出します。

INTEGER*4 getfd *fildes* = getfd(*unitn*)			
unitn	INTEGER*4	入力	外部装置番号

INTEGER*4 getfd *fildes* = getfd(*unitn*)			
戻り値	INTEGER*4 または INTEGER*8	出力	ファイルが結合されている場合はファイル記述子、結合されていない場合は -1。 64 ビット環境用にコンパイルすると、結果として INTEGER*8 が戻る。

例 : getfd()

```
    INTEGER*4 fildes, getfd, unitn/1/
    open( unitn, file='tgetfd.data' )
    fildes = getfd( unitn )
    if ( fildes .eq. -1 ) stop 'getfd: ファイルは結合されていません'
    write(*,*) 'ファイル記述子 = ', fildes
    end
```

参照 : open(2)

getfilep : 外部装置番号に対するファイルポインタの取得

関数

irtn = *c_read*(getfilep(*unitn*), *inbyte*, 1)			
c_read	*C* 関数	入力	この C 関数はユーザーが書く。下記の例を参照
unitn	INTEGER*4	入力	外部装置番号
getfilep	INTEGER*4 または INTEGER*8	戻り値	ファイルが結合されている場合はファイルポインタ、結合されていない場合は -1。 64 ビット環境用にコンパイルすると、INTEGER*8 の値が戻る。

この関数は標準 Fortran の入出力と C の入出力を混在させるために使用します。このような混在は移植不可能であり、今後リリースされるオペレーティングシステムまたは Fortran で使用できる保証はありません。したがって、この関数の使用は勧められませんし、直接のインタフェースは提供されていません。ユーザーは getfilep が戻す値を使用するために 独自の C ルーチンを作成する必要があります。C ルーチンの例を以下に示します。

例 : Fortran は C の関数に渡すのに getfilep を使用します。

```
tgetfilepF.f

      character*1  inbyte
      integer*4    c_read,  getfilep, unitn / 5 /
      external     getfilep
      write(*,'(a,$)') '数字は何 ? '

      irtn = c_read( getfilep( unitn ), inbyte, 1 )

      write(*,9)  inbyte
    9 format('C の読んだ数字は ', a )
      end
```

getfilep を実際に使用する C 関数の例を以下に示します。

```
tgetfilepC.c

      #include <stdio.h>
      int c_read_ ( fd, buf, nbytes, buf_len )
      FILE **fd ;
      char *buf ;
      int *nbytes, buf_len ;
      {
         return fread( buf, 1, *nbytes, *fd ) ;
      }
```

上記のソースプログラムをコンパイル、リンク、実行した例を以下に示します。

```
demo% 11% cc -c tgetfilepC.c
demo% 12% f77 tgetfilepC.o tgetfilepF.f
tgetfileF.f:
MAIN:
demo% 13% a.out
数字は何 ? 3
C の読んだ数字は 3
demo% 14%
```

詳細については、『Fortran プログラミングガイド』の 第 11 章「C と Fortran のインタフェース」を参照してください。
参照 : open(2)

getlog : ユーザーのログイン名の取得

サブルーチンは、次のように呼び出します。

call getlog(*name*)			
name	CHARACTER*n	出力	ユーザーのログイン名。プロセスが端末から切り離されて実行されている場合はすべて空白。*n* は、最も長い名前が入るのに十分な大きさであることが必要

例 : getlog

```
    character*18 name
    call getlog( name )
    write(*,*) "'", name, "'"
    end
```

参照 : getlogin(3)

getpid : プロセス識別子の取得

関数は、次のように呼び出します。

INTEGER*4 getpid *pid* = getpid()			
戻り値	INTEGER*4	出力	現プロセスのプロセス識別子 (ID)

例 : getpid

```
    INTEGER*4 getpid, pid
    pid = getpid()
    write(*,*) 'プロセスID = ', pid
    end
```

参照 : getpid(2)

getuid、getgid : プロセスのユーザー識別子またはグループ識別子の取得

getuid と getgid はそれぞれ、ユーザー識別子またはグループ識別子を読み取ります。

getuid : プロセスのユーザー識別子の取得

関数は、次のように呼び出します。

INTEGER*4 getuid *uid* = getuid()			
戻り値	INTEGER*4	出力	プロセスのユーザー識別子 (ID)

getgid： プロセスのグループ識別子の取得

関数は、次のように呼び出します。

INTEGER*4 getgid gid = getgid()			
戻り値	INTEGER*4	出力	プロセスのグループ識別子 (ID)

例 : getuid() と getpid()

```
    INTEGER*4 getuid, getgid, gid, uid
    uid = getuid()
    gid = getgid()
    write(*,*) uid, gid
    end
```

参照 : getuid(2)

hostnm： 現在のホスト名の獲得

関数は、次のように呼び出します。

INTEGER*4 hostnm *status* = hostnm(*name*)			
name	CHARACTER*n	出力	現在のホストの名前。 *n* は、ホスト名が入るのに十分な大きさであることが必要
戻り値	INTEGER*4	出力	*status*=0: 正常, *status*>0: エラー

例 : hostnm()

```
INTEGER*4 hostnm, status
character*8 name
status = hostnm( name )
write(*,*) 'ホスト名 = "', name, '"'
end
```

参照 : gethostname(2)

idate : 現在の日付を戻す

idate には 2 つのバージョンがあります。

- **標準バージョン** – 現在のシステム日付を整数配列 (日、月、および年) に入れる。
- **VMS バージョン** – 現在のシステム日付を 3 つの整変数 (日、月、および年) に入れる。
  このバージョンは「2000 年には無効」になる。

-lV77 というコンパイラオプションは、VMS ライブラリを要求し、time() と idate() の VMS バージョンにリンクします。このオプションを指定しないと、リンカーは標準バージョンにアクセスします。(ライブラリルーチンの VMS バージョンは、f77 では -lv77 ライブラリオプションを指定すると使えますが、f95 では使用できません。)

標準バージョンでは、現在のシステム日付を 1 つの整数配列に日、月、年の順で入れます。

サブルーチンは、次のように呼び出します。

call idate(*iarray*)			標準バージョン
iarray	INTEGER*4	出力	3 要素数の配列。日、月、年

例 : idate (標準バージョン)

```
demo% cat tidate.f
    INTEGER*4 iarray(3)
    call idate( iarray )
    write(*, "(' 日付は: ',3i5)" )  iarray
    end
demo% f77 -silent tidate.f
demo% a.out
 日付は: 10 8 1998
demo%
```

VMS idate( ) サブルーチンは、次のように呼び出します。

call idate(*m*, *d*, *y*)　　*VMS* バージョン			
m	INTEGER*4	出力	月 (1 - 12)
d	INTEGER*4	出力	日 (1 - 7)
y	INTEGER*4	出力	年 (1 - 99)　2000 年には無効

VMS idate( ) ルーチンを使用すると、リンク時とルーチンを初めて呼び出して実行するときに警告メッセージが出力されます。

注 – idate( ) ルーチンの VMS バージョンは年を示す場合に 2 桁の値しか返さないので、2000 年には無効になります。このルーチンの出力を使用して日付間の差を計算するプログラムは、1999 年 12 月 31 日以降は正しく機能しなくなります。この idate( ) ルーチンを使用しているプログラムは、ルーチンの初期呼び出し時に実行時警告メッセージを表示してユーザーに警告します。このルーチンの代わりに呼び出すことのできるルーチンとして、date_and_time( ) を参照してください。

例 : idate (VMS バージョン)

```
deom% cat titime.f
      INTEGER*4 m, d, y
      call idate ( m, d, y )
      write (*, "(' 日付は: ',3i5)" ) m, d, y
      end
demo% f77 -silent tidateV.f -lV77
"titime.f", line 2: 警告 : サブルーチン "idate" は 2000 年以降は安全ではありません。代わりに "date_and_time" を使用してください。
demo% a.out
西暦 2000 年以降はサブルーチン idate の 2 桁の年を使用して時間差を計算するのは安全ではありません。
日付は: 8 10 94
```

ieee_flags、ieee_handler、sigfpe : IEEE 算術演算

これらの副プログラムは、Fortran プログラムで ANSI/IEEE 規格 754-1985 の算術演算機能を十分に利用するために必要なモードと状態を提供します。これらの副プログラムは関数 ieee_flags(3M)、ieee_handler(3M)、および sigfpe(3) と密接に対応しています。

要約

表 5 IEEE 算術演算サポートルーチン

ieeer = ieee_flags (*action*、*mode*、*in*、*out*)		
ieeer = ieee_handler (*action*、*exception*、*hdl*)		
ieeer = sigfpe (*code*, *hdl*)		
action	CHARACTER	入力
code	sigfpe_code_type	入力
mode	CHARACTER	入力
in	CHARACTER	入力

表 5 IEEE 算術演算サポートルーチン

exception	`CHARACTER`	入力
hdl	`sigfpe_handler_type`	入力
out	`CHARACTER`	出力
戻り値	`INTEGER*4`	出力

これらの関数を効果的に使用する方法については、『数値計算ガイド』を参照してください。

`sigfpe` を使用する場合、浮動小数点状態レジスタ内の対応するトラップ可能マスクビットをユーザーが設定する必要があります。詳細は『SPARC アーキテクチャマニュアルバージョン 8』(トッパン刊) で説明されています。`libm` 関数の `ieee_handler` を呼び出すと、ユーザーに代わってトラップ可能マスクビットを設定してくれます。

mode と*exception* が受け付ける文字型のキーワードは、*action* の値によって異なります。

表 6 `ieee_flags` (*action, mode, in, out*)

action = `'clearall'`	*mode*、*in*、*out* は未使用。戻り値は `0`		
action = `'clear'` 浮動小数点の *mode* と *in*をクリアーする。 *out* は未使用。戻り値は `0`	*mode* = `'direction'`		
	mode = `'precision'` (*x86* プラットフォームのみ)		

	mode = `'exception'`	*in* = `'inexact'`
		`'division'`
		`'underflow'`
		`'overflow'`
		`'invalid'`
		`'all'`
		`'common'`
		のいずれか	

action = `'set'`
 	
浮動小数点の *mode* と *in* を設定する。
out は未使用。戻り値は `0`
	mode = `'direction'`	*in* = `'nearest'`
		`'tozero'`
		`'positive'`
		`'negative'`
		のいずれか	

	mode = `'precision'`
	(*x86* のみ)
		in = `'extended'`
		`'double'`
		`'single'`
		のいずれか	

	mode = `'exception'`	*in* = `'inexact'`
		`'division'`
		`'underflow'`
		`'overflow'`
		`'invalid'`
		`'all'`
		`'common'`
		のいずれか	

表 6 `ieee_flags` (*action, mode, in, out*)

action = 'get' *mode* の設定値を調査する。 *in*、*out* は、空白にするか、テスト対象の設定値の 1 つを設定する。 *in*、*out* に設定値を設定すると、*mode* の設定値に従った現在の設定値または 'not available' (無効) が戻る。 関数は 0 を戻す。ただし、*mode* = 'exception' の場合は、現在の例外フラグを戻す。	*mode* = 'direction'	*out* = 'nearest' 'tozero' 'positive' 'negative' のいずれか
	mode = 'precision' (*x86* のみ)	*out* = 'extended' 'double' 'single' のいずれか
	mode = 'exception'	*out* = 'inexact' 'division' 'underflow' 'overflow' 'invalid' 'all' 'common' のいずれか

表 7 `ieee_handler` (*action, in, out*)

action = 'clear' *in* に指定したユーザー例外処理をクリアーする。 *out* は未使用。	*in* = 'inexact' 'division' 'underflow' 'overflow' 'invalid' 'all' 'common' のいずれか	
action = 'set' *in* にユーザー例外処理を設定する。 *out* は、ルーチンのアドレス、または f77/f77_floating_point.h に定義されている SIGFPE_DEFAULT、SIGFPE_ABORT または SIGFPE_IGNORE。	*in* = 'inexact' 'division' 'underflow' 'overflow' 'invalid' 'all' 'common' のいずれか	

例 1 : (ハードウェアが方向をもつ丸めモードをサポートしていない場合を除いて) 丸め方向をゼロの方向に設定します。

```
INTEGER*4 ieeer
character*1 mode, out, in
ieeer = ieee_flags( 'set', 'direction', 'tozero', out )
```

例 2 : 丸め方向をクリアーします (デフォルトの方向、つまり四捨五入して丸めます)。

```
character*1 out, in
ieeer = ieee_flags('clear','direction', in, out )
```

例 3 : 設定されている例外発生ビットをすべてクリアーします。

```
character*18 out
ieeer = ieee_flags( 'clear', 'exception', 'all', out )
```

例 4 : 例 3 でオーバーフロー例外が発生すると、次のように検出します。

```
character*18 out
ieeer = ieee_flags( 'get', 'exception', 'overflow', out )
if (out .eq. 'overflow' ) stop 'overflow'
```

上記の例は、`out` を `overflow` にし、`ieeer` を 25 に設定しています。同様にコーディングすれば、`invalid` や `inexact` のような例外を検出できます。

例 5 : handl.f の内容。シグナルハンドラを書き込み、使用しています (f77)。

```
      external hand
      real r / 14.2 /,  s / 0.0 /
      i = ieee_handler( 'set', 'division', hand )
      t = r/s
      end

      INTEGER*4 function hand ( sig, sip, uap )
      INTEGER*4 sig, address
      structure /fault/
      INTEGER*4 address
      end structure
      structure /siginfo/
      INTEGER*4 si_signo
      INTEGER*4 si_code
      INTEGER*4 si_errno
      record /fault/ fault
      end structure
      record /siginfo/ sip
      address = sip.fault.address
      write (*,10) address
   10 format('例外の起きたアドレス (16進) ', z8 )
      end
```

address と function hand の宣言を INTEGER*8 に変更すると、64 ビットの SPARC V9 環境 (-xarch=v9) で例 5 が実行できます。

『数値計算ガイド』を参照してください。
参照 : floatingpoint(3)、signal(3)、sigfpe(3)、f77_floatingpoint(3F)、ieee_flags(3M)、ieee_handler(3M)

f77_floatingpoint.h： Fortran IEEE 定義

ヘッダーファイル f77_floatingpoint.h は、ANSI/IEEE 規格 754-1985 に従って、標準浮動小数点の実装に使用される定数と型を定義します。

このファイルの FORTRAN 77 ソースプログラムへのインクルードは、次のように行います。

```
#include "f77_floatingpoint.h"
```

このインクルードファイルを使用するには、Fortran のコンパイル前に前処理が必要になります。このインクルードファイルを参照するソースファイルは、名前の拡張子が .F、f90 または .F95 の場合に、自動的に前処理が行われます。

Fortran 95 プログラムは、ファイル /floatingpoint.h を代わりにインクルードします。

IEEE 丸めモード

fp_direction_type	IEEE 丸め方向モードの型。列挙の順序はハードウェアにより異なるので注意すること。

SIGFPE 処理

sigfpe_code_type	SIGFPE コードの型
sigfpe_handler_type	ユーザー定義の SIGFPE 例外ハンドラの型。特定の SIGFPE コードを処理するために呼び出される。
SIGFPE_DEFAULT	デフォルトの SIGFPE 例外処理を指示するマクロ。IEEE 例外。デフォルトの結果で実行を継続させ、他の SIGFPE コードに対しては、実行を異常終了させる。
SIGFPE_IGNORE	代替 SIGFPE 例外処理を指示するマクロ。無視して実行を継続させる。
SIGFPE_ABORT	代替 SIGFPE 例外処理を指示するマクロ。コアダンプを取り、実行を異常終了させる。

IEEE 例外処理

N_IEEE_EXCEPTION	IEEE 浮動小数点例外の数
fp_exception_type	N_IEEE_EXCEPTION 個の例外の型。各例外はビット番号を与えられる。
fp_exception_field_type	fp_exception_type により番号が与えられた IEEE 例外に対応する N_IEEE_EXCEPTION 個のビットだけをとることを目的とした型。たとえば fp_inexact は最下位ビットに対応し、fp_invalid は最下位から 5 番目のビットに対応する。操作によっては 2 つ以上の例外を設定できる。

IEEE クラス分類

fp_class_type	IEEE 浮動小数点の値と記号のクラスの並び

『数値計算ガイド』を参照してください。

参照 : ieee_environment(3M)、f77_ieee_environment(3F)

index、rindex、lnblnk : 部分列の索引または長さ

これらの関数は、次のように文字列による探索を行います。

index (*a1*, *a2*)	文字列 *a1* の中で最初に出現する文字列 *a2* の索引
rindex (*a1*, *a2*)	文字列 *a1* の中で最後に出現する文字列 *a2* の索引
lnblnk (*a1*)	文字列 *a1* の中の最後の空白以外の文字の索引

index は以下の形式をとります。

index : 文字列の中で最初に出現する部分文字列

index は、組み込み関数で次のように呼び出します。

n = index(*a1*, *a2*)			
a1	CHARACTER	入力	文字列
a2	CHARACTER	入力	部分列
戻り値	INTEGER	出力	*n*>0: *a1* の中で最初に出現する *a2* の索引 *n*=0: *a1* の中に *a2* が出現しない

INTEGER*8 と宣言されている場合は、64 ビット環境用にコンパイルされ、さらに文字変数 *a1* が非常に大きな文字列であるときに (2 ギガバイトを超えるもの)、index( ) は INTEGER*8 値を戻します。

rindex： 文字列の中で最後に出現する部分文字列

関数は、次のように呼び出します。

INTEGER*4 rindex *n* = rindex(*a1*, *a2*)			
a1	CHARACTER	入力	文字列
a2	CHARACTER	入力	部分列
戻り値	INTEGER*4 または INTEGER*8	出力	*n*>0: *a1* の中で最後に出現する *a2* の索引 *n*=0: *a1* の中に *a2* が出現しない。 64 ビット環境の場合は、INTEGER*8 が戻る。

lnblnk： 文字列の中の最後の空白以外の文字

関数は、次のように呼び出します。

n = lnblnk(*a1*)			
a1	CHARACTER	入力	文字列
戻り値	INTEGER*4 または INTEGER*8	出力	*n*>0: *a1* の中の最後の空白以外の文字の索引 *n*=0: *a1* はすべて空白以外の文字。 64 ビット環境の場合は、INTEGER*8 が戻る。

例 : index()、rindex()、lnblnk()

```
*           123456789012345678901
    character s*24 / 'abcPDQxyz...abcPDQxyz' /
    INTEGER*4 declen, index, first, last, len, lnblnk, rindex
    declen = len( s )
    first = index( s, 'abc' )
    last = rindex( s, 'abc' )
    lastnb = lnblnk( s )
    write(*,*) declen, lastnb
    write(*,*) first, last
    end
demo% f77 -silent tindex.f
demo% a.out
24 21    <- 組み込み関数 len() が宣言された s の長さを返すため、declen は 24
1 13
```

注 – 64 ビット環境で動作するようコンパイルされたプログラムは、非常に大きな文字列を処理するには index、rindex、および lnblnk (および返される変数) INTEGER*8 を宣言しなければなりません。

inmax : 正の整数の最大値の返却

関数は、次のように呼び出します。

m = inmax()			
戻り値	INTEGER*4	出力	正の整数の最大値

例 : inmax

```
      INTEGER*4 inmax, m
      m = inmax()
      write(*,*) m
      end
demo% f77 -silent tinmax.f
demo% a.out
   2147483647
demo%
```

参照 : libm_single(3f)、libm_double(3f)

参照 :『FORTRAN 77 言語リファレンス』に記載された、組み込み関数 ephuge ( )

ioinit : 入出力プロパティの初期化

IOINIT ルーチン (FORTRAN 77 のみ) は、IOINIT 呼び出しの後に開かれているファイルに対して、入出力ファイルのプロパティを確定します。IOINIT の制御する入出力プロパティは、次のとおりです。

- キャリッジ制御 – すべての論理装置でキャリッジ制御を認識する
- 空白 /ゼロ – 入力データフィールド内の空白を空白またはゼロとして扱う
- ファイルのポインタの位置 – ファイルを開く際、ポインタをファイルの先頭またはファイルの終了に位置付ける
- 接頭辞 – 0 ≤ *NN* ≤ 19 で、 <接頭辞> *NN* という名前のファイルを探索し、開く

IOINIT は次のことを行います。

- f77 入出力プロパティファイルを指定する広域のパラメータを初期化します。
- 指定した入出力プロパティファイルで、論理装置 0 から 19 までを開きます (外部に定義したファイルを実行時に論理装置に接続します)。

入出力プロパティファイルの継続

入出力プロパティファイルは接続がある限り適用されます。装置を閉じると、プロパティは適用されなくなります。あらかじめ割り当てられている装置 5 と 6 は例外で、それらの装置にはキャリッジ制御と空白/ゼロが常に適用されます。

内部フラグ

IOINIT は、実行時の入出力システムと通信するために名前付き共通ブロックを使用します。IOINIT は、次の名前付き共通ブロックと同等に内部フラグを格納します。

```
INTEGER*2 IEOF, ICTL, IBZR
COMMON /__IOIFLG/ IEOF, ICTL, IBZR ! ユーザーの名前空間ではない
```

3.0.1 以前のリリースでは、名前付き共通ブロックは IOIFLG と呼ばれていました。ユーザー定義の共通ブロックとの競合を防ぐため、この名前を __IOIFLG と変更しました。

ソースコード

IOINIT の一般的なバージョンではユーザーの要求に応じきれない場合があるため、ソースコードを提供しています。これは Fortran 77 で書かれており、次のように配置されています。

- <*install*>/SUNWspro/<*release*>/src/ioinit.f
- Sun WorkShop ソフトウェアパッケージの標準インストールを行った場合は、<*install*> は通常 /opt になりますが、<*release*> パスはコンパイラのリリースごとに変わります。

ioinit の使用法

ioinit サブルーチンは、次のように呼び出します。

call ioinit (*cctl*、*bzro*、*apnd*、*prefix*、*vrbose*)			
cctl	LOGICAL	入力	真 = すべての書式付き出力に対してキャリッジ制御が認識される (装置 0 は除く)
bzro	LOGICAL	入力	真 = 後続の空白と埋め込み空白をゼロとして扱う
apnd	LOGICAL	入力	真 = ファイルの終わりでファイルを開く (追加)
prefix	CHARACTER* *n*	入力	空白以外の文字 = 装置 *NN* で *prefixNN* という名前のファイルを検索し、開く
vrbose	LOGICAL	入力	真 = ioinit のアクティビティが発生するたびにそれを報告する

参照 : getarg(3F)、getenv(3F)

制限

次のような制限があります。

- *prefix* は 30 文字を超えることはできません。
- 環境名に関連付けられるパス名は 255 文字を超えることはできません。

引数の説明

ioinit の引数は次のとおりです。

cctl (キャリッジ制御)

デフォルトでは、キャリッジ制御はどの論理装置でも認識されません。*cctl* が .true. の場合、キャリッジ制御は、診断用チャネルである装置 0 を除いたすべての論理装置に対する書式付き出力で認識されます。*cctl* が .false. の場合は、デフォルトの設定に戻ります。

bzro (空白)

デフォルトでは、入力データフィールド内の後続の空白と埋め込み空白は無視されます。*bzro* が .true. の場合、これらの空白はゼロとして扱われます。*bzro* を .false. にした場合は、デフォルトの設定に戻ります。

apnd (追加)

デフォルトでは、順次アクセス用に開かれたファイルは先頭に位置付けられます。ファイルの既存データに書き込みを追加する場合には、ファイル終了マークにポインタを位置付けてファイルを開くことが必要になったり、便利であったりします。*apnd* を .true. にした場合、どの論理装置であっても、ファイルを開いた時にポインタはファイルの終わりに位置付けられます。*apnd* を .false. にした場合はデフォルトの設定に戻ります。

prefix (自動ファイル結合)

引数 *prefix* が空白以外の文字列ならば、*prefixNN* という形式の名前がプログラム環境内で検索されます。検索されたそれぞれの名前に関連付けられている値は、論理装置 *NN* を書式付き順次アクセス用に開くために使用されます。

この検索と接続は、0 から 19 間の 0 と 19 を含む *NN* に対してのみ行われます。19 より大きい *NN* に対しては何も行われません。61 ページの「ソースコード」を参照してください。

vrbose (IOINIT の活動状況)

引数 *vrbose* を .true. にした場合、ioinit は自分自身の活動状況について報告します。

例 : プログラム myprogram は次のような呼び出しを行います。

```
call ioinit( .true., .false., .false., 'FORT', .false.)
```

ファイル名の割り当ては、少なくとも次の 2 つの方法で行うことができます。

sh:

```
demo$ FORT01=mydata
demo$ FORT12=myresults
demo$ export FORT01 FORT12
demo$ myprogram
```

csh:

```
demo% setenv FORT01 mydata
demo% setenv FORT12 myresults
demo% myprogram
```

どちらの方法を使用しても、上記の例の `ioinit` 呼び出しからは以下の結果が得られます。

- 開かれた論理装置 1 に対してファイル `mydata` を割り当てます。
- 開かれた論理装置 12 に対してファイル `myresults` を割り当てます。
- 上記 2 つのファイルのポインタを先頭に位置付けます。
- 書式付き出力の 1 桁目は除去され、キャリッジ制御と解釈されます。
- 埋め込み空白と後続の空白は入力時には無視されます。

例 : ioinit() — 並びとコンパイル

```
demo% cat tioinit.f
    character*3  s
    call ioinit( .true., .false., .false., 'FORT', .false.)
    do i = 1, 2
    read( 1, '(a3,i4)')  s, n
    write( 12, 10 ) s, n
    end do
10  format(a3,i4)
    end
demo% cat tioinit.data
abc 123
PDQ 789
demo% f77 -silent tioinit.f
demo%
```

sh または csh を使用して以下のように環境変数を設定してください。

sh: ioinit()

```
demo$ FORT01=tioinit.data
demo$ FORT12=tioinit.au
demo$ export FORT01 FORT12
demo$
```

csh: ioinit()

```
demo% setenv FORT01 tioinit.data
demo% setenv FORT12 tioinit.au
```

テストと実行: ioinit()

```
demo% a.out
demo% cat tioinit.au
abc 123
PDQ 789
```

itime : 現在の時刻

itime は、現在のシステム時刻の時、分、秒を整数配列に入れます。サブルーチンは、次のように呼び出します。

call itime(*iarray*)			
iarray	INTEGER*4	出力	3 要素の配列: *iarray*(1) = 時 *iarray*(2) = 分 *iarray*(3) = 秒

例 : itime

```
demo% cat titime.f
    INTEGER*4 iarray(3)
    call itime( iarray )
    write (*, "(' 時刻は: ',3i5)" )  iarray
    end
demo% f77 -silent titime.f
demo% a.out
 時刻は: 15 42 35
```

参照 : time(3f)、ctime(3F)、fdate(3F)

kill : プロセスへのシグナルの送信

関数は、次のように呼び出します。

status = kill(*pid, signum*)			
pid	INTEGER*4	入力	ユーザーのプロセスのプロセス識別子
signum	INTEGER*4	入力	有効なシグナル番号。 signal(3) を参照

status = kill(*pid*, *signum*)			
戻り値	INTEGER*4	出力	*status*=0: 正常 *status*>0: エラーコード

例 (該当部分のみ): kill() を使用してメッセージを送ります。

```
      INTEGER*4 kill, pid, signum
*     …
      status = kill( pid, signum )
      if ( status .ne. 0 ) stop 'kill: エラー'
      write(*,*) 'シグナル ', signum, 'をプロセス ', pid, ' に送付しました'
      end
```

関数は、*signum* という整数型の番号で現わされるシグナルを *pid* というプロセスに送ります。有効なシグナル番号は、/usr/include/sys/signal.h という C 言語のインクルードファイル中にリストされています。

参照 : kill(2)、signal(3)、signal(3F)、fork(3F)、perror(3F)

link、symlnk : 既存ファイルへのリンクの作成

link は既存ファイルへのリンクを作成します。symlnk は既存ファイルへのシンボリックリンクを作成します。

関数は、次のように呼び出します。

status = link (*name1*, *name2*)			
INTEGER*4 symlnk *status* = symlnk (*name1*, *name2*)			
name1	CHARACTER*n	入力	既存ファイルのパス名
name2	CHARACTER*n	入力	ファイル *name1* にリンクさせるパス名 ファイル *name2* は、既存ファイルであってはならない

status = link (*name1*, *name2*)			
戻り値	INTEGER*4	出力	*status*=0: 正常 *status*>0: システムエラーコード

link： 既存ファイルへのリンクの作成

例 1 : link — ファイル tlink.db.data.1 に対して、data1 という名前のリンクを作成します。

```
demo% cat tlink.f
    character*34 name1/'tlink.db.data.1'/, name2/'data1'/
    integer*4 link, status
    status = link( name1, name2 )
    if ( status .ne. 0 ) stop 'link: エラー'
    end
demo% f77 -silent tlink.f
demo% ls -l data1
data1:ファイルもディレクトリもありません
demo% a.out
demo% ls -l data1
-rw-rw-r-- 2 generic 2 8月 11日 08:50 data1
demo%
```

symlnk： 既存ファイルへのシンボリックリンクの作成

例 2： symlnk — ファイル tlink.db.data.1 に対して、data1 という名前のシンボリックリンクを作成します。

```
demo% cat tsymlnk.f
    character*34 name1/'tlink.db.data.1'/, name2/'data1'/
    INTEGER*4 status, symlnk
    status = symlnk( name1, name2 )
    if ( status .ne. 0 ) stop 'symlnk: エラー'
    end
demo% f77 -silent tsymlnk.f
demo% ls -l data1
data1:ファイルもディレクトリもありません
demo% a.out
demo% ls -l data1
lrwxrwxrwx 1 generic 15 8月 11日 11:09 data1 -> tlink.db.data.1
demo%
```

参照： link(2)、symlink(2)、perror(3F)、unlink(3F)

注 – パス名を <sys/param.h> で定義されている MAXPATHLEN より長くすることはできません。

loc： オブジェクトのアドレスを戻す

この組み込み関数は、次のように呼び出します。

k = loc(*arg*)			
arg	任意の型	入力	任意の変数、配列、または構造体の名前
戻り値	INTEGER*4 または INTEGER*8	出力	*arg* のアドレス

k = loc(*arg*)	
	-xarch=v9 を使って、64 ビット環境で動作するようにコンパイルした場合は、INTEGER*8 ポインタが戻る。以下の注を参照。

例 : loc

```
INTEGER*4 k, loc
real arg / 9.0 /
k = loc( arg )
write(*,*) k
end
```

注 – 64 ビット Solaris 7 で動作するようコンパイルされたプログラムは、loc( ) 関数から出力を返す変数 INTEGER*8 を宣言しなければなりません。

long、short : 整数オブジェクトの変換

long および short は INTEGER*4 と INTEGER*2 間で整数オブジェクトの変換を行います。この変換は、サブプログラム呼び出し一覧では特に有効です。

long : 短整数 (INTEGER*2) から長整数 (INTEGER*4) への変換

関数は、次のように呼び出します。

call 長整数をとるサブルーチン (long(*int2*))		
int2	INTEGER*2	入力
戻り値	INTEGER*4	出力

short： 長整数から短整数への変換

関数は、次のように呼び出します。

INTEGER*2 short call 短整数をとるサブルーチン (short(*int4*))		
int4	INTEGER*4	入力
戻り値	INTEGER*2	出力

例 (該当部分のみ): long() と short()

```
integer*4 int4/8/, long
integer*2 int2/8/, short
call ExpecLong( long(int2) )
call ExpecShort( short(int4) )
…
end
```

ExpecLong はユーザープログラムによって呼び出されるサブルーチンで、長整数 (INTEGER*4) の引数をとります。ExpecShort は短整数 (INTEGER*2) の引数をとります。

long はライブラリルーチンの呼び出しに定数が使用され、-i2 オプションを指定してコードをコンパイルする場合に役立ちます。

short は、長い型のオブジェクトを短い型の整数として渡す必要がある場合に役立ちます。短い型に渡す整数が大きすぎた場合、エラーは発生しませんが、プログラムが予期しない動きをします。

longjmp、isetjmp： isetjmp で設定した位置に戻す

isetjmp は longjmp の位置を設定します。
longjmp は isetjump で設定した位置に戻ります。

isetjmp： longjmp の設定

この組み込み関数は、次のように呼び出します。

ival = isetjmp(*env*)			
env	INTEGER*4	出力	*env* は 12 要素の整数配列
戻り値	INTEGER*4	出力	*ival* = 0、isetjmp が明示的に 呼び出された場合 *ival* ≠ 0、isetjmp が longjmp から呼び出された場合

longjmp： isetjmp で設定した位置に戻す

サブルーチンは、次のように呼び出します。

call longjmp(*env, ival*)			
env	INTEGER*4	入力	*env* は isetjmp で初期化された 12 語の整数配列
ival	INTEGER*4	出力	*ival* = 0、isetjmp が明示的に呼び出された場合 *ival* ≠ 0、isetjmp が longjmp から呼び出された場合

説明

isetjmp と longjmp ルーチンは、プログラムの低レベルルーチンで遭遇するエラーや障害を処置するために使用します。この 2 つは、f77の組み込み関数です。

これらのルーチンは、最後の手段としてのみ使用してください。これらの取り扱いには、十分注意してください。また、移植性はありません。バグやその他の詳細については、
setjmp(3V) のマニュアルページを参照してください。

isetjmp は *env* にスタック環境を保存します。またレジスタ環境も保存します。

longjmp は、最後に isetjmp を呼び出して保存した環境を復元し、あたかも isetjmp の呼び出しが値 *ival* を返したかのように戻り、実行を継続します。

isetjmp から返された整数式 *ival* は、longjmp が呼び出されなければゼロです。longjmp が呼び出されれば、ゼロ以外になります。

例 : isetjmp と longjmp を使用したコード部分

```
INTEGER*4  env(12)
common /jmpblk/ env
j = isetjmp( env )                    ! ←isetjmp
if ( j .eq. 0 ) then
call  sbrtnA
else
call error_processor
end if
end
subroutine sbrtnA
INTEGER*4  env(12)
common /jmpblk/ env
call longjmp( env, ival )             !← longjmp
return
end
```

制限

- longjmp() を呼び出す前に isetjmp を起動しなければなりません。
- isetjmp と longjmp で使用される整数型の配列引数 *env* は、最低 12 個の要素からなる配列でなければなりません。
- isetjmp を呼び出すルーチンから longjmp を呼び出すルーチンへ、共通ブロック経由であるいは引数として *env* 変数を渡さなければなりません。
- longjmp はスタックをクリーンアップしようとします。longjmp は isetjmp よりも低レベルのルーチンから呼び出さなければなりません。
- 手続き名の引数として isetjmp を渡しても作用しません。

参照: setjmp(3V)

malloc、malloc 64: 記憶領域の割り当てとそのアドレスの獲得

malloc() 関数は、次のように呼び出します。

k = malloc(*n*)			
n	INTEGER*4	入力	記憶領域のバイト数
戻り値	INTEGER*4 または INTEGER*8	出力	*k*>0: *k* は割り当てられた記憶領域の開始アドレス *k*=0: エラー
	-xarch=v9 を使って、64 ビット環境用にコンパイルした場合は、INTEGER*8 ポインタ値が戻る。以下の注を参照。		

注 – Solaris 7 などの 64 ビット環境で動作するようにコンパイルされたプログラムでは、malloc() 関数とその出力を受け取る変数を INTEGER*8 と宣言する必要があります。互換性の問題を解決するには、32 ビット環境または 64 ビット環境の両方で実行する必要があるプログラム内で、malloc() の代わりに malloc64() を使用します。

関数 malloc64(3F)は、プログラムを 32 ビット環境と 64 ビット環境間で互換性を持たせるために提供された関数です。

k = malloc64 (*n*)			
n	INTEGER*8	入力	記憶領域のバイト数
戻り値	INTEGER*8	出力	*k*>0: *k* は割り当てられた記憶領域の開始アドレス *k*=0: エラー

これらの関数は、記憶領域を割り当て、その領域の開始アドレスを返します。(64 ビット環境では、この返されたバイトアドレスは、INTEGER*4 の数値範囲外になる可能性があります。受け取り側の変数では INTEGER*8 と宣言し、メモリーアドレス

が切り捨てられないようにする必要があります。) このメモリーの領域は、初期化できません。そのためメモリー領域が、ある値、特にゼロに事前設定されているとは仮定しないでください。

例： malloc を使用したコード部分

```
    parameter (NX=1000)
    pointer ( p1, X )
    real*4 X(NX)
    …
    p1 = malloc( NX*4 )
    if ( p1 .eq. 0 ) stop 'malloc: 割り当て不可'
    do 11 i=1,NX
11    X(i) = 0.
    …
    end
```

上記の例では、p1 で指定される 4,000 バイトのメモリーを獲得し、この領域を 0 に初期化しています。

詳細については、30 ページの「free : Malloc により割り当てられた記憶領域の 割り当て解除」を参照してください。

mvbits：ビットフィールドの移動

サブルーチンは、次のように呼び出されます。

call mvbits(*src, ini1, nbits, des, ini2*)			
src	INTEGER*4	入力	移動元
ini1	INTEGER*4	入力	移動元でのビットの初期位置
nbits	INTEGER*4	入力	移動させるビット数
des	INTEGER*4	出力	移動先
ini2	INTEGER*4	入力	移動先でのビットの初期位置

例： mvbits

```
demo% cat mvb1.f
* mvb1.f － 移動元 src の初期ビット位置 0 から 3 ビットを des のビット 3 へ移動。
*    src    des
* 543210 543210 ← ビット番号
* 000111 000001 ← 移動前の値
* 000111 111001 ← 移動後の値
    INTEGER*4 src, ini1, nbits, des, ini2
    data src, ini1, nbits, des, ini2
&         / 7,    0,     3,    1,    3 /
    call mvbits ( src, ini1, nbits, des, ini2 )
    write (*,"(5o3)") src, ini1, nbits, des, ini2
    end
demo% f77 -silent mvb1.f
demo% a.out
  7   0   3 71   3
demo%
```

以下の点に注意してください。

- 各ビットには、最下位ビットから最上位ビットまで、0 から 31 までの番号が付けられます。
- MVBITS は *des* のビット *ini2* から *ini2*+*nbits*-1 までを変更し、*src* のビットは変更しません。
- 制限事項:
  - *ini1* + *nbits*≧32
  - *ini2* + *nbits*≦32

perror、gerror、ierrno : エラーメッセージの取得

これらのルーチンは、以下の関数を実行します。

perror	Fortran 論理装置 0（stderr) へのメッセージの出力
gerror	システムエラーメッセージ (最後に検出されたシステムエラーの) の読み取り
ierrno	最後に検出されたシステムエラーのエラー番号の読み取り

perror： 論理装置 0 (stderr) へのメッセージ出力

サブルーチンは、次のように呼び出されます。

call perror(*string*)			
string	CHARACTER*n	入力	メッセージ。標準エラーメッセージに先だって出力される。最後に検出されたシステムエラーに対するメッセージ

例 1 :

```
    call perror( "ファイルは書式付き入出力用" )
```

gerror： 最後に検出されたエラーメッセージの取得

サブルーチンまたは関数は、次のように呼び出されます。

call gerror(*string*)			
string	CHARACTER*n	出力	最後に検出されたシステムエラーのメッセージ

例 2 : gerror のサブルーチンとしての使用

```
character string*30
…
call gerror ( string )
write(*,*) string
```

例 3 : gerror の関数としての使用 (この場合、*string* は使用しません)

```
character gerror*30, z*30
…
z = gerror( )
write(*,*) z
```

ierrno : 最後に検出されたエラー番号の取得

関数は、次のように呼び出されます。

n = ierrno()			
戻り値	INTEGER*4	出力	最後に検出されたシステムエラーの番号

この番号はエラーが実際に起こった時にしか更新されません。このようなエラーを発生させるほとんどのルーチンと入出力文は、呼び出しの後でエラーコードを返します。その値はエラー条件を引き起こした原因を示す信頼度の高いデータです。

例 4 : ierrno ()

```
INTEGER*4 ierrno, n
…
n = ierrno()
write(*,*) n
```

参照 : intro(2)、perror(3)

注意:

- perror を呼び出す際の *string* は、127 文字を超えてはいけません。

- gerror により返される文字列の長さは、それを呼び出すプログラムにより決められます。
- f77 と f95 の実行時入出力エラーについては、『Fortran ユーザーズガイド』に記載されています。

putc、fputc：論理装置への 1 文字出力

putc は論理装置 6 に出力します。通常は制御端末への出力になります。

fputc は任意の論理装置に出力します。

これらの関数は、通常の Fortran 入出力をバイパスして、Fortran 論理装置に関連付けられているファイルに 1 文字を出力します。

同じ装置上では、通常の Fortran 出力とこれらの関数の出力を混在させて使用しないでください。

putc： 論理装置 6 への出力

関数は、次のように呼び出します。

INTEGER*4 putc *status* = putc(*char*)			
char	CHARACTER	入力	装置に出力する文字
戻り値	INTEGER*4	出力	*status*=0: 正常 *status*>0: システムエラーコード

例 : putc

```
    character char, s*10 / 'OK by putc' /
    INTEGER*4 putc, status
    do i = 1, 10
    char = s(i:i)
    status = putc( char )
    end do
    status = putc( '\n' )
    end
demo% f77 -silent tputc.f
demo% a.out
OK by putc
demo%
```

fputc : 指定した論理装置への出力

関数は、次のように呼び出します。

INTEGER*4 fputc *status* = fputc(*lunit*, *char*)			
lunit	INTEGER*4	入力	出力先装置
char	CHARACTER	入力	装置に出力する文字
戻り値	INTEGER*4	出力	*status*=0: 正常 *status*>0: システムエラーコード

例 : fputc()

```
    character char, s*11 / 'OK by fputc' /
    INTEGER*4 fputc, status
    open( 1, file='tfputc.data')
    do i = 1, 11
    char = s(i:i)
    status = fputc( 1, char )
    end do
    status = fputc( 1, '\n' )
    end
demo% f77 -silent tfputc.f
demo% a.out
demo% cat tfputc.data
OK by fputc
demo%
```

参照 : putc(3S)、intro(2)、perror(3F)

qsort、qsort 64 : 1 次元配列の要素のソート

サブルーチンは、次のように呼び出します。

call qsort (*array, len, isize, compar*) call qsort 64 (*array, len8, isize8, compar*)			
array	配列	入力	ソートする要素が入っている配列
len	INTEGER*4	入力	配列内の要素の個数
len8	INTEGER*8	入力	配列内の要素の個数
isize	INTEGER*4	入力	要素のサイズ: 4= 整数または実数 8= 倍精度または複素数 16= 倍精度複素数 文字配列の場合は文字オブジェクトの長さ

call qsort (*array, len, isize, compar*) call qsort 64 (*array, len8, isize8, compar*)			
isize8	INTEGER*8	入力	要素のサイズ: 4_8= 整数または実数 8_8= 倍精度または複素数 16_8= 倍精度複素数 文字配列の場合は文字オブジェクトの長さ
compar	関数名	入力	ユーザーが提供する INTEGER*2 型の関数の名前。compar (*arg1, arg2*) と指定して、ソート順を決定する

64 ビット環境においては、2 ギガバイトを超える配列には qsort64 を使用します。この場合、INTEGER*8 データとして、配列の長さは *len8*、要素サイズは *isize8* に必ず指定してください。Fortran 95 型の定数を使用して INTEGER*8 定数を明示的に指定します。

compar の引数である *arg1* と *arg2* は、配列の要素であり、次の結果を返します。

負数	*arg1* は *arg2* の前に置かれると見なされる場合
ゼロ	*arg1* と *arg2* が等しい場合
正数	*arg1* は *arg2* の後に置かれると見なされる場合

次に例を示します。

```
demo% cat tqsort.f
    external compar
    integer*2 compar
    INTEGER*4 array(10)/5,1,9,0,8,7,3,4,6,2/, len/10/, isize/4/
    call qsort( array, len, isize, compar )
    write(*,'(10i3)') array
    end
    integer*2 function compar( a, b )
    INTEGER*4 a, b
    if ( a .lt. b ) compar = -1
    if ( a .eq. b ) compar = 0
    if ( a .gt. b ) compar = 1
    return
    end
demo% f77 -silent tqsort.f
demo% a.out
  0 1 2 3 4 5 6 7 8 9
```

ran： 0 - 1 間の乱数の生成

反復して ran を呼び出すと、均一した分布で一連の乱数を生成します。

r = ran(*i*)			
i	INTEGER*4	入力	変数または配列要素
r	REAL	出力	変数または配列要素

lcrans(3m) を参照してください。

例： ran

```
demo% cat ran1.f
* ran1.f — 乱数を生成する。
    INTEGER*4 i, n
    real r(10)
    i = 760013
    do n = 1, 10
    r(n) = ran ( i )
    end do
    write ( *, "( 5 f11.6 )" ) r
    end
demo% f77 -silent ran1.f
demo% a.out
   0.222058 0.299851 0.390777 0.607055 0.653188
   0.060174 0.149466 0.444353 0.002982 0.976519
demo%
```

以下の点に注意してください。

- 0.0 は範囲に含まれますが、1.0 は含まれません。
- 乗算合同型の一般乱数発生アルゴリズムを使用しています。
- 一般に、i の値は呼び出し元のプログラム実行中で一度だけ設定されます。
- i の初期値は大きい奇数の整数でなければなりません。
- 続けて ran を呼び出すと、次々に別の乱数が得られます。
- プログラムを実行するたびに異なる乱数の列を得るには、実行ごとに引数を異なる初期値に設定しなければなりません。
- ran は引数を下記のアルゴリズムに従って、次の乱数計算用に値を格納するために使用します。

```
    SEED = 6909 * SEED + 1 (MOD 2**32)
```

- SEED は 32 ビット数値を含み、上位 24 ビットは浮動小数点に変換され、その値が返されます。

rand、drand、irand：乱数を戻す

rand は 0.0 から 1.0 の範囲の実数値の乱数を返します。

drand は 0.0 から 1.0 の範囲の倍精度値の乱数を返します。

irand は 0 から 2147483647 の範囲の正の整数値の乱数を返します。

これらの関数は、random(3) を使用して乱数の列を発生させます。これらの 3 つの関数は、同じ 256 バイトの状態配列を共有します。3 つの関数の唯一の利点は、UNIX システムで幅広く利用できることです。乱数を生成するさらに優れた関数としては、lcrans、addrans、および shufrans があります。
random(3) および『数値計算ガイド』も参照して比較してください。

i = irand(*k*)			
r = rand(*k*)			
d = drand(*k*)			
k	INTEGER*4	入力	*k=0*: 次の乱数を乱数列から取り出す *k=1*: 乱数列を再開し、最初の数を戻す *k>0*: 新しい乱列数の種として使用し、最初の数を戻す
rand	REAL*4	出力	
drand	REAL*8	出力	
irand	INTEGER*4	出力	

例 : irand()

```
    integer*4 v(5), iflag/0/
    do i = 1, 5
    v(i) = irand( iflag )
    end do
    write(*,*) v
    end
demo% f77 -silent trand.f
demo% a.out
2078917053 143302914 1027100827 1953210302 755253631
demo%
```

rename : ファイルの名称変更

関数は、次のように呼び出します。

INTEGER*4 rename *status* = rename(*from*, *to*)			
from	CHARACTER**n*	入力	既存ファイルのパス名
to	CHARACTER**n*	入力	ファイルの新しいパス名
戻り値	INTEGER*4	出力	*status*=0: 正常 *status*>0: システムエラーコード

to で指定されたファイルがすでに存在する場合、*from* と *to* はどちらも同じタイプのファイルでなければならず、かつ同じファイルシステムに存在していなければなりません。
to がすでに存在する場合、最初にそのファイルが削除されます。

例 : rename() — ファイル trename.old の名前を trename.new に変更します。

```
demo% cat trename.f
    INTEGER*4 rename, status
    character*18 from/'trename.old'/, to/'trename.new'/
    status = rename( from, to )
    if ( status .ne. 0 ) stop 'rename: エラー'
    end
demo% f77 - silent trename.f
demo% ls trename*
trename.f trename.old
demo% a.out
demo% ls trename*
trename.f trename.new
demo%
```

参照 : rename(2)、perror(3F)

注 – パス名を <sys/param.h> で定義されている MAXPATHLEN より長くすることはできません。

---

secnds : 秒単位のシステム時間 (マイナス引数) を取得

t = secnds(*t0*)			
t0	REAL	入力	定数、変数、あるいは配列要素
戻り値	REAL	出力	午前 0 時からの秒数から *t0* を引いた値

例 : secnds

```
demo% cat sec1.f
    real elapsed, t0, t1, x, y
    t0 = 0.0
    t1 = secnds( t0 )
    y = 0.1
    do i = 1, 1000
    x = asin( y )
    end do
    elapsed = secnds( t1 )
    write ( *, 1 ) elapsed
1   format ( ' 1000 arcsines: ', f12.6, ' sec' )
    end
demo% f77 -silent sec1.f
demo% a.out
 1000 arcsines: 6.699141 sec
demo%
```

以下の点に注意してください。

- secnds からの戻り値は、0.01 秒の桁まで正確です。
- 値は真夜中からのシステム秒数で、次の真夜中を越えても正確です。
- 日の終り近くのわずかな期間だけ精度がいくらか失われることがあります。

sh : sh コマンドの高速実行

関数は、次のように呼び出します。

INTEGER*4 sh *status* = sh(*string*)			
string	CHARACTER**n*	入力	実行するコマンドを含んだ文字列
戻り値	INTEGER*4	出力	実行されたシェルの終了状態。 この値の説明については wait(*2*) を参照

例 : sh()

```
character*18 string / 'ls > MyOwnFile.names' /
INTEGER*4 status, sh
status = sh( string )
if ( status .ne. 0 ) stop 'sh: エラー'
...
end
```

関数 sh は、あたかも文字列がコマンドとしてキー入力されたかのように、sh シェルに *string* を渡します。

現プロセスはコマンドが終了するまで待機します。

fork されたプロセスは、開いているファイルをすべてフラッシュします。

- 出力ファイルの場合は、バッファは実ファイルにフラッシュされます。
- 入力ファイルの場合は、ポインタの位置が予測不能になります。

sh() はマルチスレッド対応ではありません。マルチスレッドプログラム、または並列プログラムからは呼び出さないでください。

参照 : execve(2)、wait(2)、system(3)

注 – *string* の長さは 1,024 文字を超えることができません。

signal： シグナルに対する動作の変更

関数は、次のように呼び出します。

INTEGER*4 signal *n* = signal(*signum*, *proc*, *flag*)			
signum	INTEGER*4	入力	シグナル番号。signal(3) を参照。
proc	ルーチン名	入力	ユーザーが作成したシグナル処理ルーチンの名前 (*EXTERNAL* 文で指定する必要がある)
flag	INTEGER*4	入力	*flag*<0: proc をシグナル処理として使用する *flag*≥0: *proc* を無視する。*flag* を動作の定義として渡す *flag*=0: デフォルト動作を使用する *flag*=1: このシグナルを無視する
戻り値	INTEGER*4 INTEGER*8	出力	*n*=-1: システムエラー *n*>0: 変更前の動作の定義 *n*>1: 呼び出されたルーチンのアドレス *n*<-1: signum が有効なシグナル番号である場合、 *n* は呼び出されたルーチンのアドレス。 *signum* が有効なシグナル番号ではない場合、*n* はエラー番号 64 ビット環境では、signal とその出力を受け取る変数は、INTEGER*8 と宣言する必要がある。

proc が呼び出される場合、シグナル番号が整数の引数として渡されます。

プロセスがシグナルを受信した場合のデフォルトの動作は、通常はプロセスの終了処理と異常終了です。シグナル処理ルーチンは、特殊な処理を行うために特定の例外やインタラプトを捕獲することができます。

この関数の戻り値を後で signal を呼び出す時に使用して、処理の定義を変更前に戻すことができます。

エラーがなくても負数の戻り値が返されることがあるので注意してください。有効なシグナル番号を signal() に渡して、戻り値が -1 未満であった場合は問題ありません。

f77 はプロセスが起動されると一定のシグナルをトラップするための準備をします。デフォルトの f77 の処理に戻すためには、signal を最初に呼び出した時の戻り値を保存しておかなければなりません。

f77_floatingpoint.h では、SIGFPE_DEFAULT、SIGFPE_IGNORE および SIGFPE_ABORTという *proc* 値を定義しています。55 ページを参照してください (f95 では floatingpoint.h を使用)。

64 ビット環境では、signal は、その出力を受け取る変数とともに INTEGER*8 と宣言し、返される可能性のあるアドレスが切り捨てられるのを防ぐ必要があります。

参照 : kill(1)、signal(3)、kill(3F)、『数値計算ガイド』

sleep : 一定時間の実行中断

サブルーチンは、次のように呼び出します。

call sleep(*itime*)			
itime	INTEGER*4	入力	スリープする秒数

システム時間記録が粗いので、実際の時間は *itime* よりも最大で 1 秒短くなることがあります。

例 : sleep()

```
    INTEGER*4 time / 5 /
    write(*,*) '開始'
    call sleep( time )
    write(*,*) '終了'
    end
```

参照 : sleep(3)

stat、lstat、fstat：ファイルの状態の取得

これらの関数が戻す情報は次のとおりです。

- デバイス
- i ノード番号
- 保護
- ハードリンクの数
- ユーザー識別子
- グループ識別子
- デバイスタイプ
- サイズ
- アクセス時刻
- 更新時刻
- 状態変更時刻
- 最適ブロックサイズ
- 割り当てられているブロック

`stat` と `lstat` は、どちらもファイル名を用いて問い合わせをします。`fstat` は論理装置を用いて問い合わせをします。

stat： ファイル名によるファイルの状態の取得

関数は、次のように呼び出します。

`INTEGER*4 stat` *ierr* = `stat`(*name*, *statb*)			
name	`CHARACTER*`*n*	入力	ファイルの名前
statb	`INTEGER*4`	出力	ファイル状態の情報が格納される要素数 13 の配列
戻り値	`INTEGER*4`	出力	*ierr*=0: 正常 *ierr*>0: エラーコード

例 1 : stat()

```
character name*18 /'MyFile'/
INTEGER*4 ierr, stat, lunit/1/, statb(13)
open( unit=lunit, file=name )
ierr = stat ( name, statb )
if ( ierr .ne. 0 ) stop 'stat: エラー'
write(*,*)'所有者の UID = ',statb(5),', ブロック数 = ',statb(13)
end
```

fstat : 論理装置によるファイルの状態の取得

関数は、次のように呼び出します。

INTEGER*4 fstat
ierr = fstat(*lunit*, *statb*)

lunit	INTEGER*4	入力	論理装置番号
statb	INTEGER*4	出力	ファイル状態の情報が格納される要素数 13 の配列
戻り値	INTEGER*4	出力	*ierr*=0: 正常 *ierr*>0: エラーコード

例 2 : fstat()

```
character name*18 /'MyFile'/
INTEGER*4 fstat, lunit/1/, statb(13)
open( unit=lunit, file=name )
ierr = fstat ( lunit, statb )
if ( ierr .ne. 0 ) stop 'fstat: エラー'
write(*,*)'所有者の UID = ',statb(5),', ブロック数 = ',statb(13)
end
```

lstat：ファイル名によるファイルの状態の取得

関数は、次のように呼び出します。

ierr = lstat (*name*, *statb*)			
name	CHARACTER*n	入力	ファイル名
statb	INTEGER*4	出力	ファイル状態の情報が格納される要素数 13 の配列
戻り値	INTEGER*4	出力	*ierr*=0: 正常 *ierr*>0: エラーコード

例 3 : lstat()

```
character name*18 /'MyFile'/
INTEGER*4 lstat, lunit/1/, statb(13)
open( unit=lunit, file=name )
ierr = lstat ( name, statb )
if ( ierr .ne. 0 ) stop 'lstat: エラー'
write(*,*)'所有者の UID = ',statb(5),', ブロック数 = ',statb(13)
end
```

ファイル状態を格納する配列の詳細

INTEGE*4 型の配列 *statb* に返される情報の意味は、stat(2) での構造体 *stat* の説明と同じです。

予備の値は含まれません。順序は以下のとおりです。

statb(1)	*i* ノードが存在するデバイス
statb(2)	この *i* ノードの番号
statb(3)	保護
statb(4)	ファイルへのハードリンクの数
statb(5)	所有者のユーザー識別子
statb(6)	所有者のグループ識別子

statb(7)	i ノードに対応するデバイスのデバイスタイプ
statb(8)	ファイルの合計サイズ
statb(9)	ファイルの最終アクセス時刻
statb(10)	ファイルの最終更新時刻
statb(11)	ファイルの最終状態変更時刻
statb(12)	ファイルシステム入出力操作の最適ブロックサイズ
statb(13)	実際に割り当てられているブロックの数

参照 : stat(2)、access(3F)、perror(3F)、time(3F)

注 – パス名を <sys/param.h> で定義されている MAXPATHLEN より長くすることはできません。

stat64、lstat64、fstat64: ファイルの状態の取得

stat、lstat、fstat の 64 ビット 「ロングファイル」(Solaris 2.6 と 7) バージョンです。これらのルーチンは、要素数 13 の配列 statb を INTEGER*8 で宣言しなければならない点を除いて、非 64 ビットルーチンと同じです。

system : システムコマンドの実行

関数は、次のように呼び出します。

INTEGER*4 system *status* = system(*string*)			
string	CHARACTER**n*	入力	実行するコマンドを含む文字列
戻り値	INTEGER*4	出力	実行されたシェルの終了状態。 この値の説明については wait(2) を参照

例 : system()

```
character*8 string / 'ls s*' /
INTEGER*4 status, system
status = system( string )
if ( status .ne. 0 ) stop 'system: エラー'
end
```

関数 system は、あたかもコマンドとしてキー入力されたかのように、ユーザーのシェルに文字列 *string* を渡します。

注 – *string* の長さは 1,024 文字を超えることができません。

---

system が環境変数 SHELL を見つけた場合、System はコマンドインタプリタ (シェル) として SHELL の値を使用し、それ以外は sh(l) を使用します。

現プロセスはコマンドが終了するまで待機します。

歴史的には、cc と f77 はそれぞれ異なった前提で発展してきました。

- cc が system を呼び出す場合、シェルとして必ず Bourne シェルが呼び出されます。
- f77 が system を呼び出す場合、どのシェルが呼び出されるかは環境変数 SHELL により決まります。

system 関数は開いているファイルをすべてフラッシュします。

- 出力ファイルに関しては、バッファは実ファイルにフラッシュされます。
- 入力ファイルに関しては、ポインタの位置が予測不能になります。

参照 : execve(2)、wait(2)、system(3)

sysytem() はマルチスレッド対応ではありません。マルチスレッドプログラム、または並列プログラムからは呼び出さないでください。

time、ctime、ltime、gmtime：システム時間の取得

これらのルーチンは以下の関数を実行します。

time	標準バージョン：システム時間を整数値 (0 GMT 1970/1/1 を起点とした秒数) として読み取る VMS バージョン：システム時間を文字 (hh:mm:ss) として読み取る
ctime	システム時間を ASCII 文字列に変換する
ltime	システム時間を現地時間の月、日などに分解する
gmtime	システム時間を GMT の月、日などに分解する

time：システム時間の取得

time() には、標準バージョンと VMS バージョンの 2 つのバージョンがあります。f77 のコマンド行オプション -lV77 を指定した場合は VMS バージョンが呼び出され、それ以外の場合には標準バージョンが呼び出されます。(ライブラリルーチンの VMS バージョンは、f77 では -lv77 ライブラリオプションを指定すると使えますが、f95 では使用できません。)

標準関数は、次のように呼び出します。

INTEGER*4 time *n* = time()			標準バージョン
戻り値	INTEGER*4	出力	0:0:0 GMT 1970/1/1 を起点とした秒数
	INTEGER*8	出力	64 ビット環境では、time は INTEGER*8 の値を戻す。

関数 time ( ) は、グリニッジ時間で 1970 年 1 月 1 日 00 時 00 分 00 秒からの時間を秒数で示す整数を返します。これはオペレーティングシステム時計の値です。

例 : time() — オペレーティングシステムに付属する標準バージョン

```
    INTEGER*4  n, time
    n = time()
    write(*,*)  '1970/1/1  0 時 GMT  からの秒数  =  ',  n
end
demo% f77 -silent ttime.f
demo% a.out
 1970/1/1  0 時 GMT  からの秒数  =  913240205
demo%
```

time の VMS バージョンは、現在のシステム時間を文字列として取得するサブルーチンです。

VMS サブルーチンは、次のように呼び出します。

call time(*t*)			
t	CHARACTER*8	出力	*hh*:*mm*:*ss* 形式の時間 *hh*、*mm*、*ss* は 2 桁の数字 *hh* は時 *mm* は分 *ss* は秒

例 : time(t)、VMS バージョン、ctime — システム時間を ASCII に変換します。

```
    character  t*8
    call time( t )
    write(*, "(' 現在の時刻は、 ', A8 )")  t
    end
demo% f77 -silent ttimeV.f -lV77
demo% a.out
現在の時刻は、 08:14:13
demo%
```

ctime : システム時間の文字への変換

関数 ctime は、システム時間 *stime* を変換して、24 文字の ASCII 文字列として返します。

関数は、次のように呼び出します。

CHARACTER ctime*24 *string* = ctime(*stime*)			
stime	INTEGER*4	入力	time ( ) で読み取ったシステム時間 (標準バージョン)
戻り値	CHARACTER*24	出力	文字列に変換されたシステム時間。ctime と *string* は CHARACTER*24 として型宣言する

以下に ctime が戻す値の書式を示します。詳細については、ctime(3C) のマニュアルページを参照してください。

例 : ctime()

```
     character*24 ctime, string
     INTEGER*4  n, time
     n = time()
     string = ctime( n )
     write(*,*) 'ctime: ', string
     end
demo% f77 -silent tctime.f
demo% a.out
 ctime: Wed Dec  9 13:50:05 1998
demo%
```

ltime : システム時間の月、日など (現地時間) への分解

このルーチンはシステム時間を現地時間の月、日などに分解します。

サブルーチンは、次のように呼び出します。

call ltime(*stime, tarray*)			
stime	INTEGER*4	入力	time() で読み取ったシステム時間 (標準バージョン)
tarray	INTEGER*4(9)	出力	現地時間の日、月、年、... に分解されたシステム時間

tarray の要素の意味については、次の「gmtime : システム時間の月、日など (GMT) への分解」を参照してください。

例 : ltime()

```
    integer*4  stime, tarray(9), time
    stime = time()
    call ltime( stime, tarray )
    write(*,*) 'ltime: ', tarray
    end
demo% f77 -silent tltime.f
demo% a.out
 ltime: 25 49 10 12 7 91 1 223 1
demo%
```

gmtime : システム時間の月、日など (GMT) への分解

このルーチンはシステム時間を GMT の月、日などに分解します。

サブルーチン

call gmtime(*stime, tarray*)			
stime	INTEGER*4	入力	time() で読み取ったシステム時間 (標準バージョン)
tarray	INTEGER*4(9)	出力	GMT の日、月、年、... に分解されたシステム時間

例 : gmtime

```
    integer*4  stime, tarray(9), time
    stime = time()
    call gmtime( stime, tarray )
    write(*,*) 'gmtime: ', tarray
    end
demo% f77 -silent tgmtime.f
demo% a.out
 gmtime:   12  44  19  18  5  94  6  168  0
demo%
```

ctime(3V) からの tarray() の値：以下は添字の値、単位、範囲を表しています

1	秒 (0 - 61)
2	分 (0 - 59)
3	時間 (0 - 23)
4	日 (1 - 31)
5	月 (0 - 11)
6	年 - 1900
7	曜日 (日曜日は 0)
8	年間通しの日数 (0 - 365)
9	夏時間、有効であれば 1

これらの値は C ライブラリルーチン ctime(3C) で定義されており、59 秒を超える秒数をシステムが返す理由について説明します。

参照：idate(3F)、fdate(3F)

ctime64、gmtime64、ltime64：64 ビット環境用のシステム時間のルーチン

これらは、ルーチン ctime、gmtime、ltime に対応したバージョンで、64 ビット環境との互換性を備えるために用意されたものです。ctime64、gmtime64、ltime64 は、入力変数 *stime* を INTEGER*8 にする必要があること以外は、ctime、gmtime、ltime と同じものです。

このルーチンを、INTEGER*8 の *stime* とともに 32 ビット環境で使用した場合、*stime* の値が INTEGER*4 の範囲を超えていると、ctime64 の戻り値はすべてアスタリスクになり、同時に gmtime と ltime の tarray 配列には -1 が充填されます。

topen、tclose、tread、...、tstate：テープ入出力

(FORTRAN 77 のみ) これらのルーチンを使用すると、磁気テープを操作できます。

topen	デバイス名をテープの論理装置に結合する
tclose	ファイル終了マークを書き込み、テープデバイスチャネルを閉じ、*tlu* (テープ論理装置) との結合を切り離す
tread	テープから次の物理レコードを読み出してバッファに入れる
twrite	バッファから次の物理レコードを取り出してテープに書き込む
trewin	テープを最初のデータファイルの先頭に巻き戻す
tskipf	ファイルとレコードまたはそのどちらかを前方へスキップし、ファイル終了状態をリセットする
tstate	テープ入出力チャネルの論理状態を決定する

1 つの装置では、これらの関数を標準 Fortran 入出力と混在させて使用しないでください。

最初に topen() を使用して、指定されたデバイスに対してテープ論理装置 *tlu* を開く必要があります。その後で、指定された *tlu* に関する他の操作をすべて行います。*tlu* は通常の Fortran 論理装置とはまったく関係ありません。

これらの関数は、使用する前に INTEGER*4 型宣言文で名前を宣言しておかなければならないので注意してください。

topen：デバイスとテープ論理装置との結合

INTEGER*4 topen *n* = topen(*tlu*, *devnam*, *islabeled*)			
tlu	INTEGER*4	入力	0 ～ 7 の範囲のテープ論理装置
devnam	CHARACTER	入力	デバイス名。例 : '/dev/rst0'
islabeled	LOGICAL	入力	真 = テープはラベル付き 注: ラベルはテープ上の最初のファイル
戻り値	INTEGER*4	出力	*n*=0: 正常 *n*<0: エラー

この関数はテープを駆動するものではありません。詳細については perror(3f) を参照してください。

例 : topen() ― 1/4 インチのテープファイルを開きます。

```
    CHARACTER devnam*9 / '/dev/rst0' /
    INTEGER*4 n / 0 /, tlu / 1 /, topen
    LOGICAL islabeled / .false. /
    n = topen( tlu, devnam, islabeled )
    IF ( n .LT. 0 ) STOP "topen: 開けません"
    WRITE(*,'("topen ok:", 2I3, 1X, A10)') n, tlu,  devnam
    END
```

出力は次のようになります。

```
topen ok: 0 1 /dev/rst0
```

tclose：ファイル終了マークを書き込み、テープチャネルを閉じ、*tlu* を切り離す

この関数は次のように呼び出します。

INTEGER*4 tclose *n* = tclose (*tlu*)			
tlu	INTEGER*4	入力	0 ～ 7 の範囲のテープ論理装置
n	INTEGER*4	戻り値	*n*=0: 正常 *n*<0: エラー

注意 – tclose() は装置のポインタの現在位置の直後にファイル終了マークを置き、装置を閉じます。したがって、trewin() でテープを巻き戻してから tclose() を使用した場合、その装置の内容はすべて失われます。

例 : tclose() — 開いている 1/4 インチのテープファイルを閉じます。

```
    CHARACTER devnam*9 / '/dev/rst0' /
    INTEGER*4 n / 0 /, tlu / 1 /, tclose, topen
    LOGICAL islabeled / .false. /
    n = topen( tlu, devnam, islabeled )
    n = tclose( tlu )
    IF ( n .LT. 0 ) STOP "tclose: 閉じられません"
    WRITE(*, '("tclose ok:", 2I3, 1X, A10)')  n, tlu,  devnam
    END
```

出力は次のようになります。

```
tclose ok: 0 1  /dev/rst0
```

twrite：次の物理レコードのテープへの書き込み

この関数は次のように呼び出します。

INTEGER*4 twrite *n* = twrite(*tlu*, *buffer*)			
tlu	INTEGER*4	入力	0 ～ 7 の範囲のテープ論理装置
buffer	CHARACTER	入力	512 の倍数のサイズでなければならない
n	INTEGER*4	戻り値	*n*>0: 正常。*n* は書き込まれたバイト数 *n*=0: テープの終わり *n*<0: エラー

物理レコード長が *buffer* のサイズになります。

例 : twrite() ― 2 つのレコードからなるファイルを書き込みます。

```
    CHARACTER devnam*9 / '/dev/rst0' /, rec1*512 / "abcd" /,
&       rec2*512 / "wxyz" /
    INTEGER*4 n / 0 /, tlu / 1 /, tclose, topen, twrite
    LOGICAL islabeled / .false. /
    n = topen( tlu, devnam, islabeled )
    IF ( n .LT. 0 ) STOP "topen: 開けません"
    n = twrite( tlu, rec1 )
    IF ( n .LT. 0 ) STOP "twrite: 書けません(1)"
    n = twrite( tlu, rec2 )
    IF ( n .LT. 0 ) STOP "twrite: 書けません(2)"
    WRITE(*, '("twrite ok:", 2I4, 1X, A10)')  n, tlu, devnam
    END
```

出力は次のようになります。

```
twrite ok: 512 1 /dev/rst0
```

tread：テープからの次の物理レコードの読み取り

この関数は次のように呼び出します。

INTEGER*4 tread *n* = tread(*tlu*, *buffer*)			
tlu	INTEGER*4	入力	0 ～ 7 の範囲のテープ論理装置
buffer	CHARACTER	入力	サイズは 512 の倍数で、読み取られる物理レコードの最大サイズが入るだけの大きさでなければならない
n	INTEGER*4	戻り値	*n*>0: 正常。*n* は読み取られたバイト数 *n*<0: エラー *n*=0: ファイル終了マーク

テープのポインタがファイル終了マークまたはテープ終了マークの位置にある場合、tread は値を返しますが、テープの読み取りは行いません。

例： tread() ― 前の例で書き込まれたファイルの最初のレコードを読み取ります。

```
    CHARACTER devnam*9 / '/dev/rst0' /, onerec*512 / " " /
    INTEGER*4 n / 0 /, tlu / 1 /, topen, tread
    LOGICAL islabeled / .false. /
    n = topen( tlu, devnam, islabeled )
    IF ( n .LT. 0 ) STOP "topen: 開けません"
    n = tread( tlu, onerec )
    IF ( n .LT. 0 ) STOP "tread: 読めません"
    WRITE(*,'("tread ok:", 2I4, 1X, A10)')  n, tlu,  devnam
    WRITE(*,'( A4)')  onerec
    END
```

出力は次のようになります。

```
tread ok: 512 1 /dev/rst0
abcd
```

trewin：最初のデータファイルの先頭へのテープの巻き戻し

この関数は次のように呼び出します。

INTEGER*4 trewin *n* = trewin (*tlu*)			
tlu	INTEGER*4	入力	0 ～ 7 の範囲のテープ論理装置
n	INTEGER*4	戻り値	*n*=0: 正常 *n*<0: エラー

テープにラベルが付けられている場合は、巻き戻し後にラベルはスキップされます。

例 1： trewin() — 典型的な使用例 (該当部分のみ)

```
    CHARACTER devnam*9 / '/dev/rst0' /
    INTEGER*4 n /0/, tlu /1/, tclose, topen, tread, trewin
    …
    n = trewin( tlu )
    IF ( n .LT. 0 ) STOP "trewin: 巻き戻しできません"
    WRITE(*, '("trewin ok:", 2I4, 1X, A10)') n, tlu, devnam
    …
    END
```

例 2 : trewin() — 2 つのレコードからなるファイルで 3 レコードの読み取りを試み、巻き戻して、1 レコードを読み取ります。

```
CHARACTER devnam*9 / '/dev/rst0' /, onerec*512 / " " /
INTEGER*4 n / 0 /, r, tlu / 1 /, topen, tread, trewin
LOGICAL islabeled / .false. /
n = topen( tlu, devnam, islabeled )
IF ( n .LT. 0 ) STOP "topen: 開けません"
DO r = 1, 3
   n = tread( tlu, onerec )
   WRITE(*,'(1X, I2, 1X, A4)')  r, onerec
END DO
n = trewin( tlu )
IF ( n .LT. 0 ) STOP "trewin: 巻き戻しできません"
WRITE(*, '("trewin ok:" 2I4, 1X, A10)')  n, tlu, devnam
n = tread( tlu, onerec )
IF ( n .LT. 0 ) STOP "tread: 巻き戻し後、読めません"
WRITE(*,'(A4)')  onerec
END
```

出力は次のようになります。

```
1 abcd
2 wxyz
3 wxyz
trewin ok: 0 1 /dev/rst0
abcd
```

tskipf : ファイルとレコードのスキップ、ファイル終了状態のリセット

この関数は次のように呼び出します。

INTEGER*4 tskipf *n* = tskipf(*tlu*, *nf*, *nr*)			
tlu	INTEGER*4	入力	0 ～ 7 の範囲のテープ論理装置

INTEGER*4 tskipf *n* = tskipf(*tlu*, *nf*, *nr*)			
nf	INTEGER*4	入力	最初にスキップするファイル終了マークの数
nr	INTEGER*4	入力	ファイルスキップ後にスキップする物理レコード数
n	INTEGER*4	戻り値	*n*=0: 正常 *n*<0: エラー

この関数は後方へのスキップは行いません。

tskipf は、まず最初に前方の *nf* 個のファイル終了マークをスキップし、次に前方の *nr* 個の物理レコードをスキップします。現在のファイルのポインタがファイル終了マークの位置にある場合、そのファイルもスキップするファイルの 1 個として数えられます。tskipf は、ファイル終了状態をリセットする効果もあります。次に説明する tstate と比較してください。

tskipf() の例: 典型的な使用例の該当部分のみを示します。4 個のファイルと 1 個のレコードをスキップします。

```
    INTEGER*4 nfiles / 4 /, nrecords / 1 /, tskipf, tlu / 1 /
    …
    n = tskipf( tlu, nfiles, nrecords )
    IF ( n .LT. 0 ) STOP "tskipf: スキップできません"
    …
```

tstate と比較してください。

tstate：テープ入出力チャネルの論理状態の読み取り

この関数は次のように呼び出します。

INTEGER*4 tstate *n* = tstate(*tlu*, *fileno*, *recno*, *errf*, *eoff*, *eotf*, *tcsr*)			
tlu	INTEGER*4	入力	0 ～ 7 の範囲のテープ論理装置
fileno	INTEGER*4	出力	現在のファイル番号
recno	INTEGER*4	出力	現在のレコード番号

INTEGER*4 tstate *n* = tstate(*tlu, fileno, recno, errf, eoff, eotf, tcsr*)			
errf	LOGICAL	出力	真 = エラーが発生した
eoff	LOGICAL	出力	真 = ファイルポインタの現在位置はファイル終了マーク
eotf	LOGICAL	出力	真 = テープが論理テープ終了マークに達した
tcsr	INTEGER*4	出力	真 = デバイスでハードエラーが発生した。*tcsr* にはテープドライブ制御状態レジスタの値が入る。ソフトウェアエラーの場合は *tcsr* にはゼロが返される。この状態レジスタに返される値はテープドライブのメーカーとサイズにより大きく異なる。

詳細については st(4s) を参照してください。

eoff が真である間は *tlu* からの読み取りはできません。tskipf() を使用してファイル 1 とレコード 0 をスキップすることにより、このファイル終了状態フラグを偽に設定することができます。

```
    n = tskipf( tlu, 1, 0).
```

このようにすると、後続の正しいレコードを読み取ることができます。

テープの終わり (EOT) は空のファイルにより示され、しばしば二重の EOF マークとして表されます。EOT より後を読むことはできませんが、EOT より後に書くことはできます。

例 : 2 つのレコードからなるファイルを 3 つ書き込みます。

```
    CHARACTER devnam*10 / '/dev/nrst0' /,
&                 f0rec1*512 / "eins" /, f0rec2*512 / "zwei" /,
&                 f1rec1*512 / "ichi" /, f1rec2*512 / "ni__" /,
&                 f2rec1*512 / "un__" /, f2rec2*512 / "deux" /
    INTEGER*4 n / 0 /, tlu / 1 /, tclose, topen, trewin, twrite
    LOGICAL islabeled / .false. /
    n = topen( tlu, devnam, islabeled )
    n = trewin( tlu )
    n = twrite( tlu, f0rec1 )
    n = twrite( tlu, f0rec2 )
    n = tclose( tlu )
    n = topen( tlu, devnam, islabeled )
    n = twrite( tlu, f1rec1 )
    n = twrite( tlu, f1rec2 )
    n = tclose( tlu )
    n = topen( tlu, devnam, islabeled )
    n = twrite( tlu, f2rec1 )
    n = twrite( tlu, f2rec2 )
    n = tclose( tlu )
    END
```

次の例は、tstate() を使用して EOF をトラップしながら、すべてのファイルを書き込んでいます。

例 : 前の例で書き込んだ 3 つのファイルのすべてのレコードを読み取るループの中で tstate() を使用しています。

```
    CHARACTER devnam*10 / '/dev/nrst0' /, onerec*512 / " " /
    INTEGER*4 f, n / 0 /, tlu / 1 /, tcsr, topen, tread,
&   trewin, tskipf, tstate
    LOGICAL errf, eoff, eotf, islabeled / .false. /
    n = topen( tlu, devnam, islabeled )
    n = tstate( tlu, fn, rn, errf, eoff, eotf, tcsr )
    WRITE(*,1)  'open:', fn, rn, errf, eoff, eotf, tcsr
1   FORMAT(1X, A10, 2I2, 1X, 1L, 1X, 1L,1X, 1L, 1X, I2 )
2   FORMAT(1X, A10,1X,A4,1X,2I2,1X,1L,1X,1L,1X,1L,1X,I2)
    n = trewin( tlu )
    n = tstate( tlu, fn, rn, errf, eoff, eotf, tcsr )
    WRITE(*,1)  'rewind:', fn, rn, errf, eoff, eotf, tcsr
    DO f = 1, 3
       eoff = .false.
       DO WHILE ( .NOT. eoff )
          n = tread( tlu, onerec )
          n = tstate( tlu, fn, rn, errf, eoff, eotf, tcsr )
          IF (.NOT. eoff) WRITE(*,2) 'read:', onerec,
&        fn, rn, errf, eoff, eotf, tcsr
       END DO
       n = tskipf( tlu, 1, 0 )
       n = tstate( tlu, fn, rn, errf, eoff, eotf, tcsr )
       WRITE(*,1)  'tskip: ', fn, rn, errf, eoff, eotf, tcsr
    END DO
    END
```

出力は以下のようになります。

```
open: 0 0 F F F 0
rewind: 0 0 F F F 0
read: eins 0 1 F F F 0
read: zwei 0 2 F F F 0
tskip: 1 0 F F F 0
read: ichi 1 1 F F F 0
read: ni__ 1 2 F F F 0
tskip: 2 0 F F F 0
read: un__ 2 1 F F F 0
read: deux 2 2 F F F 0
tskip: 3 0 F F F 0
```

EOF と EOT についてまとめると以下のようになります。

- テープのポインタが EOF または EOT の位置にある場合
  - `tread()` は何もせずに戻ります。テープの読み取りは行いません。
  - `tskipf(tlu,1,0)` が成功すると、EOF 状態フラグを偽にリセットして復帰しますが、テープのポインタを進めません。
- `twrite()` が成功すると、EOF と EOT 状態フラグを偽にリセットします。
- `tclose()` が成功すると、すべてのフラグを偽にリセットします。
- `tclose()` は次のようなデータの切り捨てを行います。

`tclose()` は装置のポインタの現在位置の直後に EOF マークを置き、装置を閉じます。したがって、`trewin()`でテープを巻き戻してから `tclose()` により装置を閉じた場合、内容はすべて失われます。`tclose()` のこの動作は Berkeley コードから引き継がれたものです。

参照 : `ioctl`(2)、`mtio`(4s)、`perror`(3f)、`read`(2)、`st`(4s)、`write`(2)

ttynam、isatty : 端末ポートの名前の読み取り

`ttynam` と `isatty` は、端末ポート名に関する処理を行います。

ttynam：端末ポートの名前の読み取り

関数 ttynam は論理装置 lunit に結合されている端末デバイスのパス名に空白を詰めて返します。

関数は、次のように呼び出します。

character ttynam*24 *name* = ttynam(*lunit*)			
lunit	INTEGER*4	入力	論理装置
戻り値	CHARACTER*n	出力	*name* が空白でない場合: *name* は *lunit* 上のデバイスのパス名 *name* が空の文字列 (すべて空白): *lunit* はディレクトリ /dev の中の端末デバイスと結合されていない

isatty：装置が端末であるかどうかの確認

関数は、次のように呼び出します。

terminal = isatty(*lunit*)			
lunit	INTEGER*4	入力	論理装置
戻り値	LOGICAL	出力	*terminal* = 真: 端末デバイスである *terminal* = 偽: 端末デバイスではない

例 : *lunit* が tty であるかどうかを確認します。

```
character*12 name, ttynam
INTEGER*4 lunit /5/
logical isatty, terminal
terminal = isatty( lunit )
name = ttynam( lunit )
write(*,*) '端末 = ', terminal, ', 名前 = "', name, '"'
end
```

出力は次のように表示されます。

```
端末 = T, 名前 = "/dev/ttyp1  "
```

unlink：ファイルの削除

関数は、次のように呼び出します。

INTEGER*4 unlink *n* = unlink(*patnam*)			
patnam	CHARACTER**n*	入力	ファイル名
戻り値	INTEGER*4	出力	*n*=0: 正常 *n*>0: エラー

関数 unlink は、パス名 *patnam* で指定されたファイルを削除します。

これがこのファイルに対する最後のリンクである場合、ファイルの内容はすべて失われます。

例：unlink() — tunlink.data ファイルを削除します。

```
    call unlink( 'tunlink.data' )
    end
demo% f77 -silent tunlink.f
demo% ls tunl*
tunlink.f tunlink.data
demo% a.out
demo% ls tunl*
tunlink.f
demo%
```

参照：unlink(2)、link(3F)、perror(3F)

注 – パス名を <sys/param.h> で定義されている MAXPATHLEN より長くすることはできません。

wait : プロセス終了の待機

関数は、次のように呼び出します。

INTEGER*4 wait *n* = wait(*status*)			
status	INTEGER*4	出力	子プロセスの終了状態
戻り値	INTEGER*4	出力	*n*>0: 子プロセスのプロセス識別子 *n*<0: *n* は (システムエラーコード) wait(2) を参照

wait は、シグナルを受信するか、またはその子プロセスの 1 つが終了するまで呼び出し元のプロセスを保留にします。最後の wait が呼び出された後で、子プロセスのどれかが終了した場合、wait はただちに戻ります。子プロセスがない場合、wait はエラーコードを伴ってただちに戻ります。

例 : wait() を使用したコード部分

```
    INTEGER*4 n, status, wait
    …
    n = wait( status )
    if ( n .lt. 0 ) stop 'wait: エラー'
    …
    end
```

参照 : wait(2)、signal(3F)、kill(3F)、perror(3F)

索引

E

F

G

H

I

K

L

M

N

O

S

T

U

W

X

あ

い

え

お

か