Fortran 库参考

Sun™ Studio 8

Sun Microsystems, Inc.
www.sun.com

部件号码 817-5798-10
2004 年 4 月，修订版 A

如对本文档有任何意见，请将电子邮件发送到：http://www.sun.com/hwdocs/feedback

请回收

Adobe PostScript

目录

表

开始之前必须了解的事项

《*Fortran 库参考*》介绍了 Sun™ Studio Fortran 库中的内函数和例行程序。该参考手册的读者是具有 Fortran 语言和 Solaris™ 操作环境实践知识的程序员。

该指南的读者是具有 Fortran 语言实践知识以及想了解如何有效地使用 Sun Fortran 编译器的科研人员、工程师和程序员。同时，假设这些人员总的来说熟悉 Solaris 操作环境或 UNIX®。

同时提供的《*Fortran 编程指南*》中讨论了 Solaris 操作环境中的 Fortran 编程问题，包括输入/输出、应用程序开发、库的创建和使用、程序分析、移植、优化和并行化。

印刷惯例

表 P-1 字样惯例

字样	含义	示例
`AaBbCc123`	命令、文件和目录的名称；计算机屏幕输出	编辑您的 `.login` 文件。 使用 `ls -a` 列出所有文件。 `% You have mail.`
`AaBbCc123`	您键入的内容，与计算机屏幕输出对比时	`% su` `Password:`
AaBbCc123	书名、新字或术语、要强调的字	阅读 《*用户指南*》第 6 章。 这些称为*类*选项。 您*必须*是超级用户才能这样做。
AaBbCc123	命令行占位符文字，将用实际名称或值替代	要删除文件，请键入 **`rm`** *filename*。

■ 在特别需要留空白的地方，符号 Δ 代表空格：

```
ΔΔ36.001
```

■ FORTRAN 77 标准使用比较旧的惯例，拼写名称时使用大写字母 “FORTRAN”。而当前惯例使用的是小写字母：“Fortran 95”

■ 对联机手册页的引用以主题名称和节编号表示。例如，对库例程 GETENV 的引用将显示为 getenv(3F)，表示访问该手册页的命令为：man -s 3F getenv。

表 P-2 代码惯例

代码符号	含义	表示法	代码示例
[ ]	方括号包含可选的参数。	O[*n*]	O4, O
{ }	花括号包含一个必需选项的一组选项。	d{y|n}	dy
|	“管道”或“条形”符号分隔参数，只能选择其中一个参数。	B{dynamic|static}	Bstatic
:	冒号类似逗号，有时用来分隔参数。	R*dir*[:*dir*]	R/local/libs:/U/a
…	省略号表示数列中的省略。	xinline=*fl*[,…*fn*]	xinline=alpha,dos

Shell 提示符

Shell	提示符
C shell	*machine-name*%
C shell 超级用户	*machine-name*#
Bourne shell 和 Korn shell	$
Bourne shell 和 Korn shell 的超级用户	#

访问 Sun Studio 软件和手册页

编译器和工具及其手册页未安装到标准的 /usr/bin/ 和 /usr/share/man 目录中。要访问编译器和工具，必须正确设置 PATH 环境变量（请参见第 xi 页上的“访问编译器和工具”）。要访问手册页，必须正确设置 MANPATH 环境变量（请参见第 xii 页上的“访问手册页”）。

有关 PATH 变量的详细信息，请参见 csh(1)、 sh(1) 和 ksh(1) 手册页。有关 MANPATH 变量的详细信息，请参见 man(1) 手册页。有关设置 PATH 变量和 MANPATH 变量以访问此发行版的详细信息，请参见安装指南或与系统管理员联系。

注 – 本节所含信息假定您的 Sun Studio 编译器和工具安装在 /opt 目录中。如果软件未安装在 /opt 目录中，请询问系统管理员以了解系统中的等价路径。

访问编译器和工具

按照下面的步骤确定您是否需要更改您的 PATH 变量以访问编译器和工具。

▼ 确定您是否需要设置您的 PATH 环境变量

1. **通过在命令提示符后面键入以下内容，显示 PATH 变量的当前值。**

```
% echo $PATH
```

2. **检查输出以查找包含 /opt/SUNWspro/bin/ 的路径的字符串。**

   如果您找到路径，您的 PATH 变量已设置为可访问编译器和工具。如果您没有找到路径，请按照下面过程中的说明设置 PATH 环境变量。

▼ 要设置您的 PATH 环境变量以便能够访问编译器和工具

1. **如果您正在使用 C shell，请编辑您的起始 .cshrc 文件。如果您正在使用 Bourne shell 或 Korn shell，请编辑您的起始 .profile 文件。**

2. **将以下内容增加到您的 PATH 环境变量中。如果安装了 Sun ONE Studio 软件或 Forte Developer 软件，请在其安装路径前面增加以下路径。**

   /opt/SUNWspro/bin

访问手册页

按照下列步骤确定您是否需要更改您的 MANPATH 变量以访问手册页。

▼ 确定您是否需要设置您的 MANPATH 环境变量

1. **通过在命令提示符后面键入以下内容，请求 dbx 手册页。**

```
% man dbx
```

2. **检查输出 （如果有）。**

   如果无法找到 dbx(1) 手册页，或者显示的手册页不适用于所安装软件的当前版本，请按照下面过程中的说明设置 MANPATH 环境变量。

▼ 设置您的 MANPATH 环境变量以便能够访问手册页

1. **如果您正在使用 C shell，请编辑您的起始 .cshrc 文件。如果您正在使用 Bourne shell 或 Korn shell，请编辑您的起始 .profile 文件。**

2. **将以下内容增加到您的 MANPATH 环境变量中。**

   /opt/SUNWspro/man

访问集成开发环境

Sun Studio 8 集成开发环境 (IDE) 提供用于创建、编辑、生成、调试和分析 C、 C++ 或 Fortran 应用程序性能的模块。

IDE 需要 Sun Studio 8 的核心平台组件。如果核心平台组件未安装，或者安装到以下某个位置，您必须将 SPRO_NETBEANS_HOME 环境变量设置为核心平台组件所安装的位置 (*installation_directory*/netbeans/3.5R)：

- 缺省安装目录 /opt/netbeans/3.5R
- Sun Studio 8 的编译器和工具组件的同一位置 （例如，编译器和工具组件安装在 /foo/SUNWspro 中，核心平台组件安装在 /foo/netbeans/3.5R 中）

用于启动 IDE 的命令是 sunstudio。有关此命令的详细信息，参见 sunstudio(1) 手册页。

访问编译器和工具文档

您可以从以下位置访问该文档：

- 该文档可通过随软件安装在您的本地系统或网络以下位置的文档索引获得 `file:/opt/SUNWspro/docs/index.html`。

  如果软件未安装在 `/opt` 目录中，请询问系统管理员以了解系统中的等价路径。

- 大多数手册可从 `docs.sun.com`sm web 站点获得。以下手册只能通过安装的软件获得：
  - *标准 C++ 库类参考*
  - *标准 C++ 库用户指南*
  - *Tools.h++ 类库参考*
  - *Tools.h++ 用户指南*
- 发行说明可从 `docs.sun.com` web 站点获得。
- 在 IDE 中， IDE 的所有组件的联机帮助可通过 [帮助] 菜单获得，也可通过许多窗口和对话框上的 [帮助] 按钮获得。

`docs.sun.com` web 站点 (`http://docs.sun.com`) 使您能够通过国际互联网阅读、打印和购买 Sun Microsystems 手册。如果您无法找到某本手册，请查看随软件安装在您的本地系统或网络中的文档索引。

注 – Sun 对于本文档中提到的第三方 web 站点的可用性不负任何责任，并且对从此类站点或资源中获得的任何内容、广告、产品或其它资料不作任何保证或担保。对于因为使用任何从此类站点或资源中获得的内容、商品或服务而造成或宣称造成的直接或间接损害或损失， Sun 概不负责。

可访问格式的文档

该文档以可访问格式提供，残疾人用户可通过辅助技术进行阅读。您可以找到下表所述文档的可访问版本。如果软件未安装在 /opt 目录中，请询问系统管理员以了解系统中的等价路径。

文档类型	可访问版本的格式和位置
手册 （第三方手册除外）	HTML，在 http://docs.sun.com 上
第三方手册： • *标准 C++ 库类参考* • *标准 C++ 库用户指南* • *Tools.h++ 类库参考* • *Tools.h++ 用户指南*	HTML，在安装的软件中，通过位于 file:/opt/SUNWspro/docs/index.html 的文档索引访问
自述文件和手册页	HTML，在安装的软件中，通过位于 file:/opt/SUNWspro/docs/index.html 的文档索引访问
联机帮助	HTML，可通过 IDE 中的 [帮助] 菜单访问
发行说明	HTML，在 http://docs.sun.com 上

相关的编译器和工具文档

下表描述了可从 `file:/opt/SUNWspro/docs/index.html` 和 `http://docs.sun.com` 获得的相关文档。如果软件未安装在 `/opt` 目录中，请询问系统管理员以了解系统中的等价路径。

文档标题	描述
Fortran 用户指南	介绍 `f95` 编译器的编译时运行环境以及命令行选项。同时，还包括将传统的 `f77` 程序迁移到 `f95` 的指示。
Fortran 编程指南	介绍如何针对 Solaris 环境、输入/输出、库、性能、调试和并行处理编写有效的 Fortran 代码。
OpenMP API 用户指南	简要介绍 OpenMP 多重处理 API，并且详细介绍了 Forte 开发者如何实现 OpenMP 多重处理 API。
数值计算指南	描述关于浮点计算的数值准确性的问题。

访问相关的 Solaris 文档

下表描述了可通过 `docs.sun.com` web 站点获得的相关文档。

文档集合	文档标题	描述
Solaris 参考手册集合	参见手册页部分的标题。	提供有关 Solaris 操作环境的信息。
Solaris 软件开发者集合	*链接程序和库指南*	描述了 Solaris 链接编辑器和运行时链接程序的操作。
Solaris 软件开发者集合	*多线程编程指南*	包括 POSIX 和 Solaris 线程 API、使用同步对象编程、编译多线程程序以及查找多线程程序的工具。

开发者资源

访问 http://developers.sun.com/prodtech/cc 以查找以下经常更新的资源：

- 关于编程技巧和最佳实践的文章
- 包含简短编程提示的知识库
- 编译器和工具组件的文档以及随软件安装的文档的校正
- 有关支持级别的信息
- 用户论坛
- 可下载的代码样例
- 新技术预览

您可以在 http://developers.sun.com 上找到适用于开发者的更多资源。

联系 Sun 技术支持

如果您有关于本产品的技术问题，而本文档中未涉及，请访问：

http://www.sun.com/service/contacting

发送您的意见

Sun 乐于改进文档并欢迎您发表意见和建议。请通过以下电子邮件地址将您的意见反馈给 Sun：

docfeedback@sun.com

请在您的电子邮件主题行中包含您的文档的部件号码 (817-5798-10)。

第 1 章

Fortran 库例程

本章介绍 Fortran 库例程。

本章介绍的所有例程在手册库的第 3F 节中均有对应的手册页。例如， **man -s 3F access** 将显示库例程 access 的手册页项。

本章不介绍标准的 Fortran 95 内例程。有关内例程的信息，请参见相关的 Fortran 95 标准文档。

有关可通过 Fortran 和 C 语言调用的其它数学例程，请参见《*数值计算指南*》。这些包括 libm 和 libsunmath 中的标准数学库例程 （请参见 Intro(3M)）、这些库的优化版本、 SPARC 向量数学库、 libmvec 以及其它库例程。

有关 f95 编译器实现的 Fortran 77 和 VMS 内函数的详细信息，请参见第 3 章。

1.1 数据类型的考虑事项

除非另有说明，此处列出的函数例程不是内函数例程。这意味着函数返回的数据类型可能与函数名称的隐式类型处理相冲突，需要用户进行显式类型声明。例如， getpid() 返回的数据类型为 INTEGER*4，它需要 INTEGER*4 getpid 声明，以便能够正确地处理结果。（如果没有显式类型处理，缺省情况是假设结果为 REAL，这是因为函数名称以 g 开头。）需要指出的是，显式类型语句出现在这些例程的函数摘要中。

请注意， IMPLICIT 语句以及 -dbl 和 -xtypemap 编译器选项也会更改参数的数据类型处理以及返回值的处理。在调用这些库例程时，如果应有的数据类型与实际的数据类型不相符，可能会导致无法预料的情况发生。选项 -xtypemap 和 -dbl 将数据类型为 INTEGER 的函数提升至 INTEGER*8，将 REAL 函数提升至 REAL*8，并且将 DOUBLE 函数提升至 REAL*16。要防止出现这些问题，库调用中出现的函数名和变量必须按照其应有的大小进行显式类型处理，如下所示：

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

在该示例中，显式类型处理可使得在使用 -xtypemap 和 -dbl 编译器选项时，库调用中不会发生任何数据类型提升。如果没有显式类型处理，这些选项可能会产生未预料的结果。有关这些选项的详细信息，请参阅 《*Fortran 用户指南*》和 f95(1) 手册页。

Fortran 95 编译器 f95 提供了包含文件 system.inc，它为大多数非内在的库例程定义了接口。应包含该文件，以确保所调用的函数及其参数的类型正确，尤其在使用 -xtypemap 更改了缺省的数据类型时。

```
include 'system.inc'
integer(4) mypid
mypid = getpid()
print *, mypid
```

您可以使用 Fortran 编译器的全局程序检查选项 -Xlist，捕捉与库调用造成的类型不匹配有关的许多问题。在 《*Fortran 用户指南*》、《*Fortran 编程指南*》和 f95(1) 手册页中介绍了 f95 编译器如何执行全局程序检查。

1.2 64 位环境

编译在 64 位操作环境下运行的程序（也就是说，使用 `-xarch=v9`、`v9a` 或 `v9b` 编译并在运行 64 位 Solaris 操作环境的 SPARC 平台上运行可执行文件）会更改某些函数的返回值。通常是一些连接标准的系统级例程的函数，例如 `malloc`(3F)（参见第 1-57 页上的 “`malloc`, `malloc64`, `realloc`, `free`: 分配/重新分配/释放内存”），并且可能会根据环境采用或返回 32 位或 64 位值。为保证代码在 32 位环境与 64 位环境之间的可移植性，提供的这些例程的 64 位版本始终采用并（或）返回 64 位值。下表指出适用于 64 位环境的库例程：

表 1-1 64 位环境的库例程

函数	描述
`malloc64`	分配内存并返回一个指针
`fseeko64`	重新确定大文件的位置
`ftello64`	确定大文件的位置
`stat64`, `fstat64`, `lstat64`	确定文件的状态
`time64`, `ctime64`, `gmtime64`, `ltime64`	分解系统时间，转换为字符
`qsort64`	将数组元素排序

1.3 Fortran 数学函数

下面的函数和子例程是 Fortran 数学库的一部分。它们适用于使用 `f95` 编译的所有程序。这些例程属于非内例程，它们采用专用的数据类型作为参数，并返回相同的数据类型。非内例程必须在引用它们的例程中进行声明。

许多这样的例程都是 “包装器”，即与 C 语言库中例程的 Fortran 接口，同样它们也不是标准的 Fortran 例程。它们包括 IEEE 推荐的支持函数以及专用化的随机数生成器。有关这些库的更多信息，参见 《*数值计算指南*》以及手册页 `libm_single`(3F)、`libm_double`(3F) 和 `libm_quadruple`(3F)。

1.3.1 单精度函数

这些子程序是单精度数学函数和子例程。

通常，以下函数提供了对单精度数学函数的访问，单精度函数与标准的 Fortran 通用内函数*不*对应 — 数据类型由常见的数据类型处理规则确定。

只要保留缺省的类型，这些函数就不需要使用 REAL 语句进行显式类型处理。（以 “r” 开头的名称表示 REAL，以 “i” 开头的名称表示 INTEGER。）

有关这些例程的详细信息，参见 C 数学库手册页 (3M)。例如，要了解 r_acos(x) 的信息，参见 acos(3M) 手册页。

表 1-2 单精度数学函数

函数名称	返回类型	描述
r_acos(x)	REAL	反余弦
r_acosd(x)	REAL	--
r_acosh(x)	REAL	反双曲余弦
r_acosp(x)	REAL	--
r_acospi(x)	REAL	--
r_atan(x)	REAL	反正切
r_atand(x)	REAL	--
r_atanh(x)	REAL	反双曲正切
r_atanp(x)	REAL	--
r_atanpi(x)	REAL	--
r_asin(x)	REAL	反正弦
r_asind(x)	REAL	--
r_asinh(x)	REAL	反双曲正弦
r_asinp(x)	REAL	--
r_asinpi(x)	REAL	--
r_atan2((y, x)	REAL	反正切
r_atan2d(y, x)	REAL	--
r_atan2pi(y, x)	REAL	--
r_cbrt(x)	REAL	立方根
r_ceil(x)	REAL	上舍入函数
r_copysign(x, y)	REAL	--
r_cos(x)	REAL	余弦
r_cosd(x)	REAL	--
r_cosh(x)	REAL	双曲余弦
r_cosp(x)	REAL	--
r_cospi(x)	REAL	--

表 1-2　单精度数学函数 *（续）*

函数名称	返回类型	描述
r_erf(x)	REAL	误差函数
r_erfc(x)	REAL	--
r_expm1(x)	REAL	(e**x)-1
r_floor(x)	REAL	基底
r_hypot(x, y)	REAL	斜边
r_infinity( )	REAL	--
r_j0(x)	REAL	贝塞尔
r_j1(x)	REAL	--
r_jn(x)	REAL	--
ir_finite(x)	INTEGER	--
ir_fp_class(x)	INTEGER	--
ir_ilogb(x)	INTEGER	--
ir_irint(x)	INTEGER	--
ir_isinf(x)	INTEGER	--
ir_isnan(x)	INTEGER	--
ir_isnormal(x)	INTEGER	--
ir_issubnormal(x)	INTEGER	--
ir_iszero(x)	INTEGER	--
ir_signbit(x)	INTEGER	--
r_addran()	REAL	随机
r_addrans(x, p, l, u)	subroutineR	数
r_lcran()	EAL	生成器
r_lcrans(x, p, l, u)	subroutine	
r_shufrans(x, p, l, u)	subroutine	
r_lgamma(x)	REAL	对数伽玛
r_logb(x)	REAL	--
r_log1p(x)	REAL	--
r_log2(x)	REAL	--

表 1-2　单精度数学函数 *（续）*

函数名称	返回类型	描述
r_max_normal()	REAL	
r_max_subnormal()	REAL	
r_min_normal()	REAL	
r_min_subnormal()	REAL	
r_nextafter(x, y)	REAL	
r_quiet_nan(n)	REAL	
r_remainder(x, y)	REAL	
r_rint(x)	REAL	
r_scalb(x, y)	REAL	
r_scalbn(x, n)	REAL	
r_signaling_nan(n)	REAL	
r_significand(x)	REAL	
r_sin(x)	REAL	正弦
r_sind(x)	REAL	--
r_sinh(x)	REAL	双曲正弦
r_sinp(x)	REAL	--
r_sinpi(x)	REAL	--
r_sincos(x, s, c)	subroutine	正弦和余弦
r_sincosd(x, s, c)	subroutine	--
r_sincosp(x, s, c)	subroutine	--
r_sincospi(x, s, c)	subroutine	--
r_tan(x)	REAL	正切
r_tand(x)	REAL	--
r_tanh(x)	REAL	双曲正切
r_tanp(x)	REAL	--
r_tanpi(x)	REAL	--
r_y0(x)	REAL	贝塞尔
r_y1(x)	REAL	--
r_yn(n, x)	REAL	--

- 变量 c、l、p、s、u、x 和 y 属于类型 REAL。
- 如果某个 IMPLICIT 语句实际上将以 “r” 开头的名称声明为其它数据类型，则将这些函数的类型显式声明为 REAL。
- sind(x)、asind(x) 等函数采用*度数*，而不是*弧度*。

参见：intro(3M) 以及 《*数值计算指南*》。

1.3.2 双精度函数

以下子程序为双精度数学函数和子例程。

通常，这些函数与标准的 Fortran 通用内函数*不*对应 — 数据类型由常用的数据类型处理规则决定。

这些 DOUBLE PRECISION 函数需要出现在 DOUBLE PRECISION 语句中。

有关详细信息，请参阅 C 库手册页。 d_acos(x) 的手册页为 acos(3M)。

表 1-3 双精度数学函数

函数名称	返回类型	描述
d_acos(x)	DOUBLE PRECISION	反余弦
d_acosd(x)	DOUBLE PRECISION	--
d_acosh(x)	DOUBLE PRECISION	反双曲余弦
d_acosp(x)	DOUBLE PRECISION	--
d_acospi(x)	DOUBLE PRECISION	--
d_atan(x)	DOUBLE PRECISION	反正切
d_atand(x)	DOUBLE PRECISION	--
d_atanh(x)	DOUBLE PRECISION	反双曲正切
d_atanp(x)	DOUBLE PRECISION	--
d_atanpi(x)	DOUBLE PRECISION	--
d_asin(x)	DOUBLE PRECISION	反正弦
d_asind(x)	DOUBLE PRECISION	--
d_asinh(x)	DOUBLE PRECISION	反双曲正弦
d_asinp(x)	DOUBLE PRECISION	--
d_asinpi(x)	DOUBLE PRECISION	--
d_atan2((y, x)	DOUBLE PRECISION	反正切
d_atan2d(y, x)	DOUBLE PRECISION	--
d_atan2pi(y, x)	DOUBLE PRECISION	--
d_cbrt(x)	DOUBLE PRECISION	立方根
d_ceil(x)	DOUBLE PRECISION	上舍入函数
d_copysign(x, x)	DOUBLE PRECISION	--
d_cos(x)	DOUBLE PRECISION	余弦
d_cosd(x)	DOUBLE PRECISION	--
d_cosh(x)	DOUBLE PRECISION	双曲余弦
d_cosp(x)	DOUBLE PRECISION	--
d_cospi(x)	DOUBLE PRECISION	--

表 **1-3** 双精度数学函数 *（续）*

函数名称	返回类型	描述
d_erf(x)	DOUBLE PRECISION	误差函数
d_erfc(x)	DOUBLE PRECISION	--
d_expm1(x)	DOUBLE PRECISION	(e**x)-1
d_floor(x)	DOUBLE PRECISION	基底
d_hypot(x, y)	DOUBLE PRECISION	斜边
d_infinity( )	DOUBLE PRECISION	--
d_j0(x)	DOUBLE PRECISION	贝塞尔
d_j1(x)	DOUBLE PRECISION	--
d_jn(x)	DOUBLE PRECISION	--
id_finite(x)	INTEGER	
id_fp_class(x)	INTEGER	
id_ilogb(x)	INTEGER	
id_irint(x)	INTEGER	
id_isinf(x)	INTEGER	
id_isnan(x)	INTEGER	
id_isnormal(x)	INTEGER	
id_issubnormal(x)	INTEGER	
id_iszero(x)	INTEGER	
id_signbit(x)	INTEGER	
d_addran()	DOUBLE PRECISION	随机
d_addrans(x, p, l, u)	subroutine	数
d_lcran()	DOUBLE PRECISION	生成器
d_lcrans(x, p, l, u)	subroutine	
d_shufrans(x, p, l,u)	subroutine	
d_lgamma(x)	DOUBLE PRECISION	对数伽玛
d_logb(x)	DOUBLE PRECISION	--
d_log1p(x)	DOUBLE PRECISION	--
d_log2(x)	DOUBLE PRECISION	--

表 1-3　双精度数学函数 *（续）*

函数名称	返回类型	描述
d_max_normal()	DOUBLE PRECISION	
d_max_subnormal()	DOUBLE PRECISION	
d_min_normal()	DOUBLE PRECISION	
d_min_subnormal()	DOUBLE PRECISION	
d_nextafter(x, y)	DOUBLE PRECISION	
d_quiet_nan(n)	DOUBLE PRECISION	
d_remainder(x, y)	DOUBLE PRECISION	
d_rint(x)	DOUBLE PRECISION	
d_scalb(x, y)	DOUBLE PRECISION	
d_scalbn(x, n)	DOUBLE PRECISION	
d_signaling_nan(n)	DOUBLE PRECISION	
d_significand(x)	DOUBLE PRECISION	
d_sin(x)	DOUBLE PRECISION	正弦
d_sind(x)	DOUBLE PRECISION	--
d_sinh(x)	DOUBLE PRECISION	双曲正弦
d_sinp(x)	DOUBLE PRECISION	--
d_sinpi(x)	DOUBLE PRECISION	--
d_sincos(x, s, c)	subroutine	正弦和余弦
d_sincosd(x, s, c)	subroutine	--
d_sincosp(x, s, c)	subroutine	--
d_sincospi(x, s, c)	subroutine	
d_tan(x)	DOUBLE PRECISION	正切
d_tand(x)	DOUBLE PRECISION	--
d_tanh(x)	DOUBLE PRECISION	双曲正切
d_tanp(x)	DOUBLE PRECISION	--
d_tanpi(x)	DOUBLE PRECISION	--
d_y0(x)	DOUBLE PRECISION	贝塞尔
d_y1(x)	DOUBLE PRECISION	--
d_yn(n, x)	DOUBLE PRECISION	--

- 变量 c、 l、 p、 s、 u、 x 和 y 的类型为 DOUBLE PRECISION。
- 在 DOUBLE PRECISION 语句中或者通过适当的 IMPLICIT 语句显式声明这些函数的类型。
- sind(x)、 asind(x) 等函数采用*度数*，而不是*弧度*。

参见：intro(3M) 以及 《*数值计算指南*》。

1.3.3 四倍精度函数

这些子程序为**四倍**精度 (REAL*16) 数学函数和子例程。

通常，这些函数与标准的通用内函数*不*对应，而数据类型由常用的数据类型处理规则决定。

四倍精度函数必须出现在 REAL*16 语句中。

表 1-4 四倍精度 libm 函数

函数名称	返回类型
q_copysign(x, y)	REAL*16
q_fabs(x)	REAL*16
q_fmod(x)	REAL*16
q_infinity( )	REAL*16
iq_finite(x)	INTEGER
iq_fp_class(x)	INTEGER
iq_ilogb(x)	INTEGER
iq_isinf(x)	INTEGER
iq_isnan(x)	INTEGER
iq_isnormal(x)	INTEGER
iq_issubnormal(x)	INTEGER
iq_iszero(x)	INTEGER
iq_signbit(x)	INTEGER
q_max_normal()	REAL*16
q_max_subnormal()	REAL*16
q_min_normal()	REAL*16
q_min_subnormal()	REAL*16
q_nextafter(x, y)	REAL*16
q_quiet_nan(n)	REAL*16
q_remainder(x, y)	REAL*16
q_scalbn(x, n)	REAL*16
q_signaling_nan(n)	REAL*16

- 变量 c、l、p、s、u、x 和 y 属于四倍精度类型。
- 通过 REAL*16 语句或适当的 IMPLICIT 语句对这些函数进行显式类型处理。
- sind(x)、asind(x) 等函数采用*度数*，而不是*弧度*。

如果您需要使用其它任何四倍精度 libm 函数，请在调用语句前面加上 $PRAGMA C(*fcn*)。有关详细信息，请参见《*Fortran 编程指南*》中介绍 C–Fortran 接口的章节。

1.4 abort: 终止和写入信息转储文件

该子例程的调用方式如下所示：

```
call abort
```

abort 刷新 I/O 缓冲区，然后终止进程，可能会在当前的目录中产生 core 文件内存转储。有关限制或抑制信息转储的信息，请参见 limit(1)。

1.5 access: 检查文件权限或存在性

该函数的调用方式如下所示：

INTEGER*4 access status = access (*name, mode*)			
name	**字符**	输入	文件名
mode	**字符**	输入	权限
返回值	INTEGER*4	输出	*status*=0：正常 *status*>0：错误代码

access 确定您是否能够使用由 *mode* 指定的权限访问文件 *name*。如果使用由 *mode* 指定的权限访问文件成功， access 返回零。有关错误代码的含义，请参见 gerror(3F)。

将 *mode* 设置为 r、 w 和 x 中的一个或多个值，可以是任何顺序或任意组合，或者为空白，其中 r、 w 和 x 具有以下含义：

'r'	测试是否有读取权限
'w'	测试是否有写入权限
'x'	测试是否有执行权限
' '	测试文件是否存在

示例 1：测试是否有读/写权限：

```
      INTEGER*4  access, status
      status = access ( 'taccess.data', 'rw' )
      if ( status .eq. 0 ) write(*,*) "ok"
      if ( status .ne. 0 ) write(*,*) 'cannot read/write', status
```

示例 2：测试文件是否存在：

```
      INTEGER*4  access, status
      status = access ( 'taccess.data', ' ' )    ! blank mode
      if ( status .eq. 0 ) write(*,*) "file exists"
      if ( status .ne. 0 ) write(*,*) 'no such file', status
```

1.6 alarm: 在指定的时间后调用子例程

该函数的调用方式如下所示：

INTEGER*4 alarm n = alarm (*time*, *sbrtn*)			
time	INTEGER*4	输入	等待的秒数 （0= 不调用）
sbrtn	例程名称	输入	要执行的子程序必须列在外部语句中。
返回值	INTEGER*4	输出	最后一次报警的剩余时间

示例：alarm — 等待 9 秒钟，然后调用 sbrtn：

```
      integer*4 alarm, time / 1 /
      common / alarmcom / i
      external sbrtn
      i = 9
      write(*,*) i
      nseconds =  alarm ( time, sbrtn )
      do n = 1,100000        ! Wait until alarm activates sbrtn.
       r = n           ! (any calculations that take enough time)
       x=sqrt(r)
      end do
      write(*,*) i
      end
      subroutine sbrtn
      common / alarmcom / i
      i = 3                      ! Do no I/O in this routine.
      return
      end
```

参见：alarm(3C)、 sleep(3F) 和 signal(3F)。注意以下限制条件：

- 子例程无法将自身的名称传给 alarm。
- alarm 例程生成的信号可能会干扰 I/O，因此调用的子例程 *sbrtn* 本身决不能执行任何 I/O 操作。
- 从并行化或多线程的 Fortran 程序中调用 alarm() 可能会产生未预料的结果。

1.7 bit: 位函数：and, or, ..., bit, setbit, ...

定义如下：

and(*word1, word2*)	计算其参数的按位操作 *and*。
or(*word1, word2*)	计算其参数的按位操作 *inclusive or*。
xor(*word1, word2*)	计算其参数的按位操作 *exclusive or*。
not(*word*)	返回其参数的按位操作 *complement*。
lshift(*word, nbits*)	逻辑左移，但没有结尾的循环进位。

`rshift(` *word, nbits* `)`	带符号扩展的算术右移。
`call bis(` *bitnum*`,` *word* `)`	将 *word* 中的位 *bitnum* 设置为 1。
`call bic(` *bitnum*`,` *word* `)`	将 *word* 中的位 *bitnum* 清除为 0。
`bit(` *bitnum*`,` *word* `)`	测试 *word* 中的位 *bitnum*，如果位是 1，返回 `.true`。如果位是 0，返回 `.false`。
`call setbit(`*bitnum*`,`*word*`,`*state*`)`	如果 *state* 为非零，将 *word* 中的位 *bitnum* 设置为 1，否则将其清除清除为零。

MIL-STD-1753 的交替外部版本为：

`iand(` *m*`,` *n* `)`	计算其参数的按位 “*与*”。
`ior(` *m*`,` *n* `)`	计算其参数的按位操作 *inclusive or*。
`ieor(` *m*`,` *n* `)`	计算其参数的按位操作 *exclusive or*。
`ishft(` *m*`,` *k* `)`	为不带结尾循环进位的逻辑移位（如果 *k*>0 则为左移，如果 *k*<0 则为右移）。
`ishftc(` *m*`,` *k*`,` *ic* `)`	循环移位：*m* 最右边的 *ic* 位循环左移 *k* 个位置。
`ibits(` *m*`,` *i*`,` *len* `)`	提取位：从 *m* 中的 *i* 位开始提取 *len* 位。
`ibset(` *m*`,` *i* `)`	设置位：返回值等于字 *m*，并且位数 *i* 设置为 1。
`ibclr(` *m*`,` *i* `)`	清除位：返回值等于字 *m*，并且位数 *i* 设置为 0。
`btest(` *m*`,` *i* `)`	测试 *m* 中的位 *i*；如果位是 1，返回 `.true`，如果位是 0，返回 `.false`。

有关操作位字段的其它函数的信息，请参见第 1-60 页上的 “`mvbits`: 移动位字段”以及第 2 章和第 3 章。

1.7.1 用法：`and`, `or`, `xor`, `not`, `rshift`, `lshift`

对于内函数：

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* 和 *nbits* 是整型输入参数。这些函数是编译器内联扩展的内函数。返回的数据类型是第一个参数的数据类型。

对于 *nbits* 的合理值不做任何测试。

示例：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)',
     1         6x 'not(4)'/4o12.11)
        end
demo% f95 tandornot.f
demo% a.out
    and(7,4)     or(7,4)    xor(7,4)      not(4)
 00000000004 00000000007 00000000003 37777777773
demo%
```

示例：lshift, rshift：

```
demo% cat tlrshift.f
       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% f95 tlrshift.f
demo% a.out
 lshift(7,1) rshift(4,1)
 00000000016 00000000002
demo%
```

1.7.2 用法：bic, bis, bit, setbit

对于子例程和函数

```
call bic( bitnum, word )
call bis( bitnum, word )
call setbit( bitnum, word, state )
LOGICAL bit
x = bit( bitnum, word )
```

bitnum、*state* 和 *word* 为 INTEGER*4 输入参数。函数 bit() 返回逻辑值。

对位进行编号，使位 0 成为最低有效位，而位 31 是最高有效位。

bic、bis 和 setbit 是外部子例程，bit 是外部函数。

示例 3：bic, bis, setbit, bit：

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

1.8 chdir: 更改缺省目录

该函数的调用方式如下所示：

INTEGER*4 chdir *n* = chdir(*dirname*)			
dirname	字符	输入	目录名称
返回值	INTEGER*4	输出	*n*=0：正常， *n*>0：错误代码

示例：chdir — 将 cwd 更改为 MyDir：

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

参见：chdir(2)、 cd(1) 和 gerror(3F) 中的错误代码解释。

路径名称的长度不能超过 <sys/param.h> 中规定的 MAXPATHLEN 值。路径可以是相对路径或绝对路径。

使用该函数可能会导致按单元查询失败。

某些 Fortran 文件操作会根据文件名重新打开文件。在执行输入/输出时，使用 chdir 可能使运行时系统不能跟踪使用相对路径名创建的文件，包括使用打开语句创建但没有文件名的文件。

1.9 chmod: 更改文件的模式

该函数的调用方式如下所示：

INTEGER*4 chmod *n* = chmod(*name*, *mode*)			
name	**字符**	输入	路径名
mode	**字符**	输入	*chmod*(1) 可以识别的任意字符 例如 o-w， 444 等等。
返回值	INTEGER*4	输出	*n* = 0：OK；*n*>0：系统错误编号

示例：chmod — 为 MyFile 增加写入权限：

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

参见：chmod(1) 和 gerror(3F) 中的错误代码解释。

路径名称的长度不能超过 <sys/param.h> 中规定的 MAXPATHLEN 值。路径可以是相对路径或绝对路径。

1.10 date: 获取以字符串表示的当前日期

注 – 由于该例程只返回两位数值的年份，因此它不是“2000 年安全”的例程。在 1999 年 12 月 31 日之后，使用该例程输出计算日期差异的程序可能无法正常工作。使用此 date() 例程的程序在第一次调用例程时会显示运行时警告消息，从而向用户发出报警。参见可用作替换例程的 date_and_time()。

该子例程的调用方式如下所示：

call date(*c*)			
c	**字符** *9	输出	变量、数组、数组元素或字符子串

返回字符串 *c* 的格式为 *dd-mmm-yy*，其中 *dd* 表示两位数的日期， *mmm* 表示三个字母的月份缩写， *yy* 表示两位数的年份 （不是 2000 年安全！）。

示例：date：

```
demo% cat dat1.f
* dat1.f -- Get the date as a character string.
        character c*9
        call date ( c )
        write(*,"(' The date today is: ', A9 )" ) c
        end
demo% f95 dat1.f
demo% a.out
Computing time differences using the 2 digit year from subroutine
       date is not safe after year 2000.
 The date today is: 9-Jan-02
demo%
```

参见 idate() 和 date_and_time()。

1.11 date_and_time: 获取日期和时间

这是 Fortran 95 内例程，能够安全进入 2000 年。

date_and_time 子例程返回日期和实时时钟的数据，且返回的是本地时间以及本地时间与通用协调时间 (UTC) （也称为格林威治标准时间， GMT）之间的时差。

date_and_time() 子例程的调用方式如下所示：

call date_and_time(*date, time, zone, values*)			
date	字符 *8	输出	以 CCYYMMDD 格式表示的日期，其中 CCYY 表示四位数的年份，MM 表示两位数的月份，而 DD 表示一个月中两位数的天数。例如：19980709
time	字符 *10	输出	以 hhmmss.sss 格式表示的当前时间，其中 hh 表示小时， mm 表示分数，ss.sss 表示秒和毫秒。
zone	字符 *5	输出	与 UTC 的时差，以小时数和分钟数表示并且采用 hhmm 格式。
values	INTEGER*4 VALUES(8)	输出	下面介绍的 8 个元素组成的整数数组。

INTEGER*4 *values* 数组中返回的 8 个值为：

VALUES(1)	以 4 位整数表示的年份，例如：1998。
VALUES(2)	以从 1 到 12 的整数表示的月份。
VALUES(3)	以从 1 到 31 的整数表示的一个月中的天数。
VALUES(4)	以分钟数表示的与 UTC 的时差。
VALUES(5)	以从 1 到 23 的整数表示的一天中的小时数。
VALUES(6)	以从 1 到 59 的整数表示的一个小时中的分钟数。
VALUES(7)	以从 0 到 60 的整数表示的一分钟中的秒数。
VALUES(8)	范围为从 0 至 999 的毫秒数。

date_and_time 的用法示例：

```
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_time    array values:'
       print *,'year=',date_time(1)
       print *,'month_of_year=',date_time(2)
       print *,'day_of_month=',date_time(3)
       print *,'time difference in minutes=',date_time(4)
       print *,'hour of day=',date_time(5)
       print *,'minutes of hour=',date_time(6)
       print *,'seconds of minute=',date_time(7)
       print *,'milliseconds of second=',date_time(8)
       print *, 'DATE=',b(1)
       print *, 'TIME=',b(2)
       print *, 'ZONE=',b(3)
       end
```

2000 年 2 月 16 日在美国加利福尼亚的计算机上运行该例程时，输出结果如下所示：

```
 date_time    array values:
 year= 2000
 month_of_year= 2
 day_of_month= 16
 time difference in minutes= -420
 hour of day= 11
 minutes of hour= 49
 seconds of minute= 29
 milliseconds of second= 236
 DATE=20000216
 TIME=114929.236
 ZONE=-0700
```

1.12 dtime, etime: 经过的执行时间

这两个函数都返回经过的时间值 （或返回 -1.0 的误差指示）。返回的时间以秒数表示。

Fortran 95 使用的 dtime 和 etime 版本在缺省情况下使用系统的低分辨率时钟，其分辨率是百分之一秒。然而，如果在 Sun OS™ 操作系统公用程序 ptime(1) (/usr/proc/bin/ptime) 下运行程序，则会使用高分辨率时钟。

1.12.1 dtime: 上次 dtime 调用后经过的时间

对于 dtime，经过的时间为：

- 第一次调用：自开始执行起经过的时间
- 后续调用：自上一次调用 dtime 起经过的时间
- 单处理器：CPU 占用的时间
- 多处理器：所有 CPU 的时间总和，该数据不是很有用，可以使用 etime。

注 – 在并行化的循环中调用 dtime 会给出不确定的结果，这是因为参与循环的所有线程共用经过时间计数器。

该函数的调用方式如下所示：

e = dtime(*tarray*)			
tarray	real(2)	输出	*e*= -1.0：错误：*tarray* 值未定义 *e*≠ -1.0：如果没有错误，则为 *tarray(1)* 中的用户时间。如果没有错误，则为 *tarray(2)* 中的系统时间
返回值	real	输出	*e*= -1.0：错误 *e*≠ -1.0：*tarray(1)* 和 *tarray(2)* 的总和

示例：dtime()，单处理器：

```
demo% cat tdtime.f
       real e, dtime, t(2)
       print *, 'elapsed:', e, ', user:', t(1), ', sys:', t(2)
       do i = 1, 10000
        k=k+1
       end do
       e = dtime( t )
       print *, 'elapsed:', e, ', user:', t(1), ', sys:', t(2)
       end
demo% f95 tdtime.f
demo% a.out
elapsed: 0.0E+0 , user: 0.0E+0 , sys: 0.0E+0
 elapsed: 0.03 , user: 0.01 , sys: 0.02
demo%
```

1.12.2 etime: 自开始执行起经过的时间

对于 etime，经过的时间为：

- 单处理器执行 — 调用进程的 CPU 时间
- 多处理器执行 — 处理程序时墙上时钟的时间

如果 PARALLEL 或 OMP_NUM_THREADS 环境变量被定义为大于 1 的某个整数值，运行时库确定程序在多处理器模式下执行。

该函数的调用方式如下所示：

e = etime(*tarray*)			
tarray	real(2)	输出	*e*= -1.0：错误：*tarray* 值未定义。 *e*≠ -1.0：单处理器：*tarray(1)* 中的用户时间。*tarray(2)* 中的系统时间。 多处理器：*tarray(1)* 中的墙上时钟时间，*tarray(2)* 中的时间为 0.0。
返回值	real	输出	*e*= -1.0：错误 *e*≠ -1.0：*tarray(1)* 和 *tarray(2)* 的总和

注意初次调用 etime 的时间不准确。它只是使系统时钟开始运转。不要使用初次调用 etime 返回的值。

示例：etime()，单处理器：

```
demo% cat tetime.f
       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% f95 tetime.f
demo% a.out
elapsed: 0.02 , user: 0.01 , sys: 0.01
demo%
```

参见 times(2) 和 《*Fortran 编程指南*》。

1.13 exit: 终止进程并设置状态

该子例程的调用方式如下所示：

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

示例：exit()：

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

exit 刷新并关闭进程中的所有文件，如果它在执行 wait，则会通知父进程。

status 的低序 8 位适用于父进程。这 8 个位向左移动 8 个位，其它所有位为零。（因此，*status* 应该位于 256 - 65280 的范围中。）该调用从不返回任何值。

C 函数 exit 可能在最终的系统 ‘exit’ 之前执行清除操作。

不带参数调用 exit 会导致出现编译时间警告消息，并且自动提供零作为参数。参见：exit(2)、 fork(2)、 fork(3F)、 wait(2) 和 wait(3F)。

1.14 fdate: 以 ASCII 字符串返回日期和时间

该子例程或函数的调用方式如下所示：

call fdate(*string*)		
string	字符 *24	输出

或者：

字符 fdate*24 *string* = fdate()			如果用作函数，调用例程必须定义 fdate 的类型和大小。
返回值	**字符** *24	输出	

示例 1：fdate 用作子例程：

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

输出：

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

示例 2：fdate 用作函数，输出结果相同：

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

参见：ctime(3)、 time(3F) 和 idate(3F)。

1.15 flush: 刷新逻辑单元的输出

该函数的调用方式如下所示：

INTEGER*4 flush *n* = flush(*lunit*)			
lunit	INTEGER*4	输入	逻辑单元
返回值	INTEGER*4	输出	*n* = 0 没有错误 *n* > 0 错误编号

flush 函数将逻辑单元 lunit 缓冲区的内容刷新到相关的文件中。对于逻辑单元 0 和逻辑单元 6，在两者都与控制台终端关联时，该函数特别有用。如果遇到错误，该函数返回正错误编号，否则返回零。

参见 fclose(3S)。

1.16 fork: 创建当前进程的副本

该函数的调用方式如下所示：

INTEGER*4 fork *n* = fork()			
返回值	INTEGER*4	输出	*n*>0：*n*= 副本的进程 ID *n*<0， *n*= 系统错误代码

fork 函数创建正在调用的进程副本。两个进程之间的唯一区别在于返回给其中某个进程 （称为*父*进程）的值是副本的进程 ID。副本通常称为*子*进程。返回给子进程的值将为零。

为了避免外部文件中的输入/输出缓冲区内容重复，在产生派生进程之前，将会刷新所有已开放供写入的逻辑单元。

示例：fork()：

```
      INTEGER*4 fork, pid
      pid = fork()
      if(pid.lt.0) stop 'fork error'
      if(pid.gt.0) then
             print *, 'I am the parent'
      else
             print *, 'I am the child'
      endif
```

由于没有一种令人满意的方式能够保留整个 exec 例程的开放逻辑单元，因此尚未提供对应的 exec 例程。然而，可以使用 system(3F) 执行 fork/exec 的常用函数。参见：fork(2)、 wait(3F)、 kill(3F)、 system(3F) 和 perror(3F)。

1.17 fseek, ftell: 确定文件的位置以及重新确定文件的位置

fseek 和 ftell 是允许重新确定文件位置的进程。 ftell 返回文件的当前位置，反映已与文件开头偏移很多字节。在程序的以后某个时间， fseek 可以使用此保存的偏移值，将文件重新放置到同样的位置以便读取。

1.17.1 fseek: 重新确定逻辑单元中文件的位置

该函数的调用方式如下所示：

INTEGER*4 fseek *n* = fseek(*lunit*, *offset*, *from*)			
lunit	INTEGER*4	输入	开放的逻辑单元
offset	INTEGER*4 *或* INTEGER*8	输入	相对于 *from* 指定位置的偏移值 （以字节数表示）
	在使用 -xarch=v9 选项编译用于 64 位环境 （例如 Solaris 7 或 8）的程序时，需要提供 INTEGER*8 偏移值。如果提供了文字常量，它必须是 64 位常量，例如：100_8		
from	INTEGER*4	输入	0= 文件开头 1= 当前位置 2= 文件结尾
返回值	INTEGER*4	输出	*n*=0：OK；*n*>0：系统错误代码

注 – 对于以后的文件，在调用 fseek 函数后面加上输出操作 （例如 WRITE）会导致 fseek 位置之后的所有数据记录被删除，并且被换成新的数据记录 （以及文件结尾标记）。只有直接访问文件才能将记录重新写入到位。

示例：fseek() — 重新确定 MyFile 的位置，使其与开始位置偏移两个字节：

```
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 error'
end
```

示例：在 64 位环境中使用 -xarch=v9 编译的同样示例：

```
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
```

1.17.2 ftell: 返回文件的当前位置

该函数的调用方式如下所示：

INTEGER*4 ftell *n* = ftell(*lunit*)			
lunit	INTEGER*4	输入	开放的逻辑单元
返回值	INTEGER*4	输出	*n*>=0：*n*= 与文件开头偏移的字节数 *n*<0：*n*= 系统错误代码

示例：ftell()：

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

示例：在 64 位环境中使用 -xarch=v9 编译的同样示例：

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

参见 fseek(3S) 和 perror(3F)；同时参见 fseeko64(3F) ftello64(3F)。

1.18 fseeko64, ftello64: 确定大文件的位置以及重新确定大文件的位置

fseeko64 和 ftello64 是 fseek 和 ftell 的“大文件”版本。它们采用并返回 INTEGER*8 文件位置偏移值。（“大文件”指大于 2 千兆字节的文件，因此字节位置必须以 64 位整数表示。）使用这些版本确定或重新确定大文件的位置。

1.18.1 fseeko64: 重新确定逻辑单元中文件的位置

该函数的调用方式如下所示：

INTEGER fseeko64 *n* = fseeko64(*lunit*, *offset64*, *from*)			
lunit	INTEGER*4	输入	开放的逻辑单元
offset64	INTEGER*8	输入	相对于 *from* 指定位置的 64 位偏移值（以字节数表示）
from	INTEGER*4	输入	0= 文件开头 1= 当前位置 2= 文件结尾
返回值	INTEGER*4	输出	*n*=0：OK；*n*>0：系统错误代码

注 – 对于以后的文件，在调用 fseeko64 函数后面加上输出操作 （例如 WRITE）会导致 fseek 位置之后的所有数据记录被删除，并且被换成新的数据记录 （以及文件结尾标记）。只有直接访问文件才能将记录重新写入到位。

示例：fseeko64() — 重新确定 MyFile 的位置，使其与开始位置偏移两个字节：

```
      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 error'
      end
```

1.18.2 ftello64: 返回文件的当前位置

该函数的调用方式如下所示：

INTEGER*8 ftello64 *n* = ftello64(*lunit*)			
lunit	INTEGER*4	输入	开放的逻辑单元
返回值	INTEGER*8	输出	*n*≥0：*n*= 与文件开头偏移的字节数 *n*<0：*n*= 系统错误代码

示例：ftello64()：

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

1.19 getarg, iargc: 获取命令行参数

命令行中的 getarg 和 iargc 访问参数（在命令行预处理程序扩展之后）。

1.19.1 getarg: 获取命令行参数

该子例程的调用方式如下所示：

call getarg(*k*, *arg*)			
k	INTEGER*4	输入	参数索引（0= 第一个 = 命令名称）
arg	字符 **n*	输出	第 *k* 个参数
n	INTEGER*4	*arg* 的大小	大得足以容纳最长的参数

1.19.2 iargc: 获取命令行参数的数量

该函数的调用方式如下所示：

m = iargc()			
返回值	INTEGER*4	输出	命令行中参数的数量

示例：iargc 和 getarg，获取参数的数量以及每一个参数：

```
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% f95 yarg.f
demo% a.out *.f
1 first.f
2 yarg.f
```

参见 execve(2) 和 getenv(3F)。

1.20 getc, fgetc: 获取下一个字符

getc 和 fgetc 从输入流中获取下一个字符。不要将这些例程的调用与相同逻辑单元中正常的 Fortran I/O 混在一起。

1.20.1 getc: 从 stdin 中获取下一个字符

该函数的调用方式如下所示：

INTEGER*4 getc *status* = getc(*char*)			
char	**字符**	输出	下一个字符
返回值	INTEGER*4	输出	*status*=0：OK *status*=-1：文件结尾 *status*>0：系统错误代码或 f77 I/O 错误代码

示例：getc 获取键盘上的每一个字符；注意 Control-D (^D)：

```
      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
```

编译之后，运行以上源代码的样例如下：

```
demo% a.out
ab              Program reads letters typed in
0 141           Program outputs status and octal value of the characters entered
0 142               141 represents 'a', 142 is 'b'
0 012               012 represents the RETURN key
^D                  terminated by a CONTROL-D.
-1 377          Next attempt to read returns CONTROL-D
demo%
```

对于任何逻辑单元，不要将正常的 Fortran 输入与 `getc()` 混在一起。

1.20.2 `fgetc`: 获取指定逻辑单元中的下一个字符

该函数的调用方式如下所示：

`INTEGER*4 fgetc`
status = `fgetc(` *lunit*, *char* `)`

lunit	`INTEGER*4`	输入	逻辑单元
char	字符	输出	下一个字符
返回值	`INTEGER*4`	输出	*status*=-1：文件结尾 *status*>0：系统错误代码或 `f77` I/O 错误代码

示例：`fgetc` 获取 `tfgetc.data` 中的每一个字符；注意换行 (Octal 012)：

```
      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
```

编译之后，运行以上源代码的样例如下：

```
demo% cat tfgetc.data
ab
yz
demo% a.out
0 141        'a' read
0 142        'b' read
0 012        linefeed read
0 171        'y' read
0 172        'z' read
0 012        linefeed read
-1 012       CONTROL-D read
demo%
```

对于任何逻辑单元，不要将正常的 Fortran 输入与 `fgetc()` 混在一起。

参见：`getc`(3S)、 `intro`(2) 和 `perror`(3F)。

1.21 getcwd: 获取当前工作目录的路径

该函数的调用方式如下所示：

`INTEGER*4 getcwd` *status* = `getcwd(` *dirname* `)`			
dirname	字符 **n*	输出 返回当前目录的路径	当前工作目录的路径名称。 *n* 的长度必须能够满足最长路径名的要求。
返回值	`INTEGER*4`	输出	*status*=0：OK *status*>0：错误代码

示例：`getcwd`：

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

参见：chdir(3F)、 perror(3F) 和 getwd(3)。

注意：路径名称的长度不能超过 <sys/param.h> 中规定的 MAXPATHLEN 值。

1.22 getenv: 获取环境变量的值

该子例程的调用方式如下所示：

call getenv(*ename*, *evalue*)			
ename	字符 *n*	输入	寻找的环境变量名称
evalue	字符 *n*	输出	找到的环境变量值，如果不成功，则为空白。

ename 和 *evalue* 的长度必须能够足以容纳相应的字符串。

如果 *evalue* 太短而不能容纳整个字符串值，字符串将被截断以便能够容纳 *evalue*。

getenv 子例程搜索环境列表中是否有格式为 *ename=evalue* 的字符串，如果存在这样的字符串，则在 *evalue* 中返回值；否则在 *evalue* 中填上空白。

示例：使用 getenv() 打印 $SHELL 的值：

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

参见：*execve*(2) 和 *environ*(5)。

1.23 getfd: 获取外部单元编号的文件描述符

该函数的调用方式如下所示：

INTEGER*4 getfd *fildes* = getfd(*unitn*)			
unitn	INTEGER*4	输入	外部单元编号
返回值	INTEGER*4 *或* INTEGER*8	输出	如果文件已连接，返回文件描述符。如果文件未连接，返回 -1。在编译用于 64 位环境时，返回 INTEGER*8 结果。

示例：getfd()：

```
      INTEGER*4 fildes, getfd, unitn/1/
      open( unitn, file='tgetfd.data' )
      fildes = getfd( unitn )
      if ( fildes .eq. -1 ) stop 'getfd: file not connected'
      write(*,*) 'file descriptor = ', fildes
      end
```

参见 open(2)。

1.24 getfilep: 获取外部单元编号的文件指针

函数为：

irtn = *c_read*(getfilep(*unitn*), *inbyte*, 1)			
c_read	C 函数	输入	用户自己的 C 函数。参见示例。
unitn	INTEGER*4	输入	外部单元编号。
getfilep	INTEGER*4 *或* INTEGER*8	返回值	如果文件已连接，则返回文件指针；如果文件未连接，返回 -1。在编译用于 64 位环境时，返回 INTEGER*8 值。

该函数用于将标准的 Fortran I/O 与 C I/O 混合在一起。这样的混合不可移植，并且不保证能够用于以后的操作系统或 Fortran 版本。建议不要使用该函数，并且没有提供直接的接口。您必须创建自己的 C 例程，才能使用 `getfilep` 返回的值。下面显示 C 例程的样例。

示例：Fortran 通过将 `getfilep` 传给 C 函数，从而使用 getfilep：

```
demo% cat tgetfilepF.f

        character*1  inbyte
        integer*4    c_read,  getfilep, unitn / 5 /
        external     getfilep
        write(*,'(a,$)') 'What is the digit? '

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

        write(*,9)  inbyte
   9    format('The digit read by C is ', a )
        end
```

实际使用 `getfilep` 的 C 函数样例：

```
demo% cat 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 ) ;
}
```

下面是编译-生成-运行该函数的样例：

```
demo% cc -c tgetfilepC.c
demo% f95 tgetfilepC.o tgetfilepF.f
demo% a.out
What is the digit? 3
The digit read by C is 3
demo%
```

有关更多信息，请阅读 《*Fortran 编程指南*》中介绍 C-Fortran 接口的章节。参见 `open`(2)。

1.25 getlog: 获取用户的登录名称

该子例程的调用方式如下所示：

call getlog(*name*)			
name	字符 *n	输出	返回用户的登录名称，如果进程脱离终端正在运行，则所有内容都为空白。*n* 应该大得足以容纳最长的名称。

示例：getlog：

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

参见 getlogin(3)。

1.26 getpid: 获取进程 ID

该函数的调用方式如下所示：

INTEGER*4 getpid *pid* = getpid()			
返回值	INTEGER*4	输出	当前进程的进程 ID。

示例：getpid：

```
        INTEGER*4 getpid, pid
        pid = getpid()
        write(*,*) 'process id = ', pid
        end
```

参见 getpid(2)。

1.27 getuid, getgid: 获取进程的用户 ID 或组 ID

getuid 和 getgid 分别获取进程的用户或组 ID。

1.27.1 getuid: 获取进程的用户 ID

该函数的调用方式如下所示：

INTEGER*4 getuid *uid* = getuid()			
返回值	INTEGER*4	输出	进程的用户 ID

1.27.2 getgid: 获取进程的组 ID

该函数的调用方式如下所示：

INTEGER*4 getgid gid = getgid()			
返回值	INTEGER*4	输出	进程的组 ID

示例：getuid() 和 getpid()：

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

参见：getuid(2)。

1.28 hostnm: 获取当前主机的名称

该函数的调用方式如下所示：

INTEGER*4 hostnm *status* = hostnm(*name*)			
name	**字符** ******n*	输出	当前主机系统的名称。 *n* 必须大得足以能够容纳主机名称。
返回值	INTEGER*4	输出	*status*=0：OK *status*>0：错误

示例：hostnm()：

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

参见 gethostname(2)。

1.29 idate: 返回当前日期

idate 将当前系统日期放入一个整数数组中：天数、月份和年份。

该子例程的调用方式如下所示：

call idate(*iarray*)			*Standard Version*
iarray	INTEGER*4	输出	三元素数组：天数、月份和年份。

示例：idate （标准版本）：

```
demo% cat tidate.f
       INTEGER*4 iarray(3)
       call idate( iarray )
       write(*, "(' The date is: ',3i5)" )  iarray
       end
demo% f95 tidate.f
demo% a.out
 The date is: 10 8 1998
demo%
```

1.30 ieee_flags,ieee_handler,sigfpe: IEEE 算术

这些子程序提供了在 Fortran 程序中完全利用 ANSI/IEEE 标准 754-1985 算术所需的模式和状态。它们与函数 ieee_flags(3M)、ieee_handler(3M) 和 sigfpe(3) 密切对应。

下面是汇总表：

表 1-5 IEEE 算术支持例程

ieeer = ieee_flags(*action*,*mode*,*in*,*out*) ieeer = ieee_handler(*action*,*exception*,*hdl*) ieeer = sigfpe(*code*, *hdl*)		
action	字符	输入
code	sigfpe_code_type	输入
mode	字符	输入
in	字符	输入
exception	字符	输入
hdl	sigfpe_handler_type	输入
out	字符	输出
返回值	INTEGER*4	输出

有关如何有策略地使用这些函数的详细信息，参见 Sun 的 《*数值计算指南*》。

如果您使用 sigfpe，必须在浮点状态寄存器中自己设置对应的陷阱-启用-掩码位。详细信息位于 SPARC 体系结构手册中。 libm 函数 ieee_handler 为您设置这些陷阱-启用-掩码位。

mode 和 *exception* 接受的字符关键字取决于 *action* 的值。

表 1-6 ieee_flags (*action,mode,in,out*) 参数和操作

action = 'clearall'	*mode*， *in*， *out*，未使用；返回 0	
action = 'clear' 清除 *mode*， *in* *out* 未使用；返回 0	*mode* = 'direction'	
	mode = 'exception'	*in* = 'inexact' *或* 'division' *或* 'underflow' *或* 'overflow' *或* 'invalid' *或* 'all' *或* 'common'
action = 'set' 设置浮点 *mode*， *in* *out* 未使用；返回 0	*mode* = 'direction'	*in* = 'nearest' *或* 'tozero' *或* 'positive' *或* 'negative'
	mode = 'exception'	*in* = 'inexact' *或* 'division' *或* 'underflow' *或* 'overflow' *或* 'invalid' *或* 'all' *或* 'common'
action = 'get' 测试 *mode* 设置 *in*， *out* 可能为空白或者要测试的某个设置返回当前的设置，具体视 *mode* 还是 'not available' 而定。如果 *mode* = 'exception'，函数返回 0 或当前的异常标志。	*mode* = 'direction'	*out* = 'nearest' *或* 'tozero' *或* 'positive' *或* 'negative'
	mode = 'exception'	*out* = 'inexact' *或* 'division' *或* 'underflow' *或* 'overflow' *或* 'invalid' *或* 'all' *或* 'common'

表 1-7 ieee_handler (*action,in,out*) 参数

action = 'clear' 清除 *in* 的用户异常处理； *out* 未使用	*in* = 'inexact' *或* 'division' *或* 'underflow' *或* 'overflow' *或* 'invalid' *或* 'all' *或* 'common'
action = 'set' 设置 *in* 的用户处理异常； *out* 是处理程序例程的地址，或者是 floating point.h 中定义的 SIGFPE_DEFAULT、 SIGFPE_ABORT 或 SIGFPE_IGNORE。	*in* = 'inexact' *或* 'division' *或* 'underflow' *或* 'overflow' *或* 'invalid' *或* 'all' *或* 'common'

示例 1：将舍入方向设置为向零舍入，除非硬件不支持要求的舍入模式；

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

示例 2：将舍入方向清理为缺省方向 （向最近的值舍入）：

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

示例 3：清除所有由于产生异常而出现的位：

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

示例 4：按如下所示检测溢出异常：

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

以上代码将 out 设置为 overflow 并且将 ieeer 设置为 25 （该值视平台而定）。类似的编码检测到异常，例如 invalid 或 inexact。

示例 5：hand1.f，编写并使用符号处理程序：

```
      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('Exception at hex address ', z8 )
      end
```

将 address 和 function hand 的声明更改为 INTEGER*8，以便在 64 位 SPARC V9 环境 (-xarch=v9) 中启用示例 5。

参见 《*数值计算指南*》。参见：floatingpoint(3)、 signal(3)、 sigfpe(3)、 floatingpoint(3F)、 ieee_flags(3M) 和 ieee_handler(3M)。

1.30.1 floatingpoint.h: Fortran IEEE 定义

头文件 floatingpoint.h 定义了根据 ANSI/IEEE 标准 754-1985 实现标准的浮点所使用的常量和类型。

如下所示在 Fortran 95 源程序中包含该文件：

```
#include "floatingpoint.h"
```

使用该包含文件需要在 Fortran 编译之前预处理。如果引用该包含文件的源文件名的扩展名为 .F、 .F90 或 .F95，将自动预处理该文件。

IEEE 舍入模式：

fp_direction_type	IEEE 舍入方向模式的类型。枚举的顺序随硬件而变化。

SIGFPE 处理：

sigfpe_code_type	SIGFPE 代码的类型。
sigfpe_handler_type	调用以处理特定 SIGFPE 代码并且用户可自定义的 SIGFPE 异常处理程序的类型。
SIGFPE_DEFAULT	表示缺省 SIGFPE 异常处理的宏：收到缺省的结果让 IEEE 异常继续出现以及由于其它 SIGFPE 代码的原因而终止 IEEE 异常。
SIGFPE_IGNORE	表示交替的 SIGFPE 异常处理的宏，即忽略异常并继续执行。
SIGFPE_ABORT	表示交替的 SIGFPE 异常处理的宏，即终止信息转储。

IEEE 异常处理：

N_IEEE_EXCEPTION	不相同的 IEEE 浮点异常数。
fp_exception_type	N_IEEE_EXCEPTION 异常的类型。每一个异常都有指定的位编号。
fp_exception_field_type	该类型应该至少能够容纳 N_IEEE_EXCEPTION 个位，它与 fp_exception_type 规定的 IEEE 例外数相对应。因此， fp_inexact 对应于重要性最低的位，fp_invalid 对应于第五个不太重要的位。一些操作可以设置多个异常。

IEEE 分类：

fp_class_type	IEEE 浮点值和符号分类的列表。

参阅 《*数值计算指南*》。参见 ieee_environment(3F)。

1.31 index,rindex,lnblnk: 子串的索引或长度

这些函数通过字符串搜索：

index(*a1*,*a2*)	字符串 *a1* 中第一次出现的字符串 *a2* 的索引。
rindex(*a1*,*a2*)	字符串 *a1* 中最后一次出现的字符串 *a2* 的索引。
lnblnk(*a1*)	字符串 *a1* 中最后一个非空白字符串的索引

index 有以下几种形式：

1.31.1 index: 字符串中第一次出现子串

索引是通过以下方式调用的内函数：

n = index(*a1*, *a2*)			
a1	**字符**	输入	主字符串
a2	**字符**	输入	子串
返回值	INTEGER	输出	*n*>0：字符串 *a1* 中第一次出现的 *a2* 的索引。 *n*=0：*a2* 不出现在 *a1* 中。

如果 index() 被声明为 INTEGER*8，在编译用于 64 位环境时它将返回 INTEGER*8 值，并且字符变量 *a1* 是一个非常大的字符串 （大于 2 千兆字节）。

1.31.2 rindex: 字符串中最后一次出现子串

该函数的调用方式如下所示：

INTEGER*4 rindex *n* = rindex(*a1*, *a2*)			
a1	字符	输入	主字符串
a2	字符	输入	子串
返回值	INTEGER*4 *或* INTEGER*8	输出	*n*>0：*a1* 中最后一次出现的 *a2* 的索引 *n*=0：*a2* 不出现在 *a1* 中。在 64 位环境中返回 INTEGER*8。

1.31.3 lnblnk: 字符串最后一个非空白字符串

该函数的调用方式如下所示：

n = lnblnk(*a1*)			
a1	字符	输入	字符串
返回值	INTEGER*4 *或* INTEGER*8	输出	*n*>0：*a1* 中最后一个非空白字符串的索引 *n*=0：*a1* 全部不为空白在 64 位环境中返回 INTEGER*8。

示例：index(),rindex(),lnblnk()：

```
demo% cat tindex.f
*                         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% f95 tindex.f
demo% a.out
24 21     <- declen is 24  because intrinsic len() returns the declared length of  s
1 13
```

注 – 编译用于在 64 位环境下运行的程序必须声明 index、rindex 和 lnblnk（以及他们的接收变量） INTEGER*8，以便处理非常大的字符串。

1.32 inmax: 返回最大正整数

该函数的调用方式如下所示：

m = inmax()			
返回值	INTEGER*4	输出	最大正整数

示例：inmax：

```
demo% cat tinmax.f
       INTEGER*4 inmax, m
       m = inmax()
       write(*,*) m
       end
demo% f95 tinmax.f
demo% a.out
   2147483647
demo%
```

参见 libm_single(3F) 和 libm_double(3F)。参见第 3 章中介绍的非标准 FORTRAN 77 内函数 ephuge()。

1.33 itime: 当前时间

itime 将当前系统时间放入整数数组中：小时、分钟和秒。该子例程的调用方式如下所示：

call itime(*iarray*)			
iarray	INTEGER*4	输出	三元素数组： *iarray*(1) = 小时 *iarray*(2) = 分钟 *iarray*(3) = 秒

示例：itime：

```
demo% cat titime.f
       INTEGER*4 iarray(3)
       call itime( iarray )
       write (*, "(' The time is: ',3i5)" )  iarray
       end
demo% f95 titime.f
demo% a.out
 The time is: 15 42 35
```

参见 time(3F)、 ctime(3F) 和 fdate(3F)。

1.34 kill: 将信号发给进程

该函数的调用方式如下所示：

status = kill(*pid*, *signum*)			
pid	INTEGER*4	输入	某个用户进程的进程 ID。
signum	INTEGER*4	输入	有效的信号编号。参见 *signal*(3)。
返回值	INTEGER*4	输出	*status*=0：OK *status*>0：错误代码

示例 （片段）：使用 kill() 发送消息：

```
        INTEGER*4 kill, pid, signum
*     …
        status = kill( pid, signum )
        if ( status .ne. 0 ) stop 'kill: error'
        write(*,*) 'Sent signal ', signum, ' to process ', pid
        end
```

函数将信号 *signum* 和整数信号编号发给进程 *pid*。有效的信号编号列在 C 包含文件 /usr/include/sys/signal.h 中。

参见：kill(2)、 signal(3)、 signal(3F)、 fork(3F) 和 perror(3F)。

1.35 link, symlnk: 链接到现有的文件

link 创建到现有文件的链接。 symlink 创建到现有文件的符号链接。

该函数的调用方式如下所示：

status = link(*name1*, *name2*)			
INTEGER*4 symlnk *status* = symlnk(*name1*, *name2*)			
name1	字符 *n	输入	现有文件的路径名称
name2	字符 *n	输入	链接到文件的路径名称， *name1*。 *name2* 不能已经存在。
返回值	INTEGER*4	输出	*status*=0：OK *status*>0：系统错误代码

1.35.1 link: 创建到现有文件的链接

示例 1：link：创建到文件 tlink.db.data.1 的链接 data1：

```
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: error'
        end
demo% f95 tlink.f
demo% ls -l data1
data1 not found
demo% a.out
demo% ls -l data1
-rw-rw-r-- 2 generic 2 Aug 11 08:50 data1
demo%
```

1.35.2 symlnk: 创建到现有文件的符号链接

示例 2：symlnk：创建到文件 tlink.db.data.1 的符号链接 data1：

```
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: error'
       end
demo% f95 tsymlnk.f
demo% ls -l data1
data1 not found
demo% a.out
demo% ls -l data1
lrwxrwxrwx 1 generic 15 Aug 11 11:09 data1 -> tlink.db.data.1
demo%
```

参见：link(2)、 symlink(2)、 perror(3F) 和 unlink(3F)。

注意：路径名称的长度不能超过 <sys/param.h> 中规定的 MAXPATHLEN 值。

1.36 loc: 返回对象的地址

该内函数的调用方式如下所示：

k = loc(*arg*)			
arg	任意类型	输入	变量或数组
返回值	INTEGER*4 *或* INTEGER*8	输出	*arg* 的地址
	在使用 -xarch=v9 编译在 64 位环境下运行的程序时返回 INTEGER*8。参见下面的说明。		

示例：loc：

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

注 – 编译以在 64 位环境下运行的程序应该将接收 loc() 函数输出的变量声明为 INTEGER*8。

1.37 long, short: 整型对象转换

long 和 short 处理 INTEGER*4 与 INTEGER*2 之间的整型对象转换，并且在子程序调用列表中特别有用。

1.37.1 long: 将短整型转换为长整型

该函数的调用方式如下所示：

call *ExpecLong*(long(*int2*))		
int2	INTEGER*2	输入
返回值	INTEGER*4	输出

1.37.2 short: 将长整型转换为短整型

函数为：

INTEGER*2 short call *ExpecShort*(short(*int4*))		
int4	INTEGER*4	输入
返回值	INTEGER*2	输出

示例 （片段）：long() 和 short()：

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

ExpecLong 是应该需要 *long* (INTEGER*4) 整型参数的用户程序调用的一些子例程。同样， *ExpecShort* 应该需要 *short* (INTEGER*2) 整型参数。

如果常量用于调用库例程并且使用 -i2 选项编译代码，则 long 非常有用。

在长对象必须作为短整型传送的类似上下文环境下， short 非常有用。将整型传递给幅度太大的短整型虽然不会导致错误，但是会导致出现未预料的行为。

1.38 longjmp, isetjmp: 返回至 isetjmp 设置的位置

isetjmp 为 longjmp 设置位置； longjmp 返回到该位置。

1.38.1 isetjmp: 为 longjmp 设置位置

该内函数的调用方式如下所示：

ival = isetjmp(*env*)			
env	INTEGER*4	输出	*env* 是由 12 个元素组成的整数数组。在 64 位环境中，它必须声明为 INTEGER*8。
返回值	INTEGER*4	输出	如果显式调用 isetjmp， *ival* = 0。 如果通过 longjmp 调用 isetjmp， *ival* ≠ 0。

1.38.2 longjmp: 返回至 isetjmp 设置的位置

该子例程的调用方式如下所示：

call longjmp(*env*, *ival*)			
env	INTEGER*4	输入	*env* 是由 isetjmp 初始化的 12 个词组成的整数数组。 在 64 位环境中，它必须声明为 INTEGER*8。
ival	INTEGER*4	输出	如果显式调用 isetjmp， *ival* = 0。 如果通过 longjmp 调用 isetjmp，*ival* ≠ 0。

1.38.3 描述

isetjmp 和 longjmp 例程用于处理在程序的低级别例程中遇到的错误和中断。它们属于 f95 内函数。

这些例程只有在迫不得已的情况下才应该使用。它们需要受规范约束，并且不可移植。请阅读手册页和 setjmp(3V) 以了解错误及其它详细信息。

isetjmp 将堆栈环境保存在 *env* 中。它还会保存寄存器环境。

longjmp 恢复上一次调用 isetjmp 保存的环境，并且以继续执行的这种方式返回值，就好象调用 isetjmp 刚刚返回值 *ival*。

如果未调用 longjmp，从 isetjmp 中返回的整数表达式 *ival* 为零，如果调用了 longjmp，则返回的整数表达式不为零。

示例：使用 isetjmp 和 longjmp 的代码片段：

```
        INTEGER*4  env(12)
        common /jmpblk/ env
        j = isetjmp( env )
        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 )
        return
        end
```

1.38.4 限制条件

- 您必须调用 isetjmp，然后才能调用 longjmp。
- isetjmp 和 longjmp 的 *env* 整数数组变量长度至少必须为 12 个元素。
- 您必须将 *env* 变量以常规方式或作为参数，从调用 isetjmp 的例程传递至调用 longjmp 的例程。
- longjmp 尝试清理堆栈。必须从较低的调用级别中调用 longjmp，而不是从 isetjmp 中调用。
- 将 isetjmp 作为属于过程名称的参数传递并不起作用。

参见 setjmp(3V)。

1.39 malloc, malloc64, realloc, free: 分配/重新分配/释放内存

函数 malloc()、malloc64() 和 realloc() 分配内存块并返回块的起始地址。返回值可以用于设置 INTEGER 或 Cray 式样的 POINTER 变量。realloc() 根据新的大小重新分配现有的内存块。free() 释放 malloc()、malloc64() 或 realloc() 分配的内存块。

注 – 这些例程在 f95 中作为内函数，在 f77 作为外函数。除非您要使用自己的版本，否则，它们不应该出现在 Fortran 95 程序的类型声明或 EXTERNAL 语句中。realloc() 例程只能用于 f95。

符合标准的 Fortran 95 程序应该将 ALLOCATE 和 DEALLOCATE 语句用于 ALLOCATABLE 数组，以便执行动态的内存管理，并且不能直接调用 malloc/realloc/free。

传统的 Fortran 77 程序可能使用 malloc()/malloc64() 为 Cray 式样的 POINTER 变量赋值，POINTER 变量的数据表示法与 INTEGER 变量的数据表示法相同。Cray 式样的 POINTER 变量用在 f95 程序中，以便支持从 Fortran 77 中移植程序。

1.39.1 分配内存：malloc, malloc64

malloc() 函数的调用方式如下所示：

<table>
<tr><td colspan="4">k = malloc(n)</td></tr>
<tr><td>n</td><td>INTEGER</td><td>输入</td><td>内存的字节数</td></tr>
<tr><td rowspan="2">返回值</td><td>INTEGER
(Cray
POINTER)</td><td>输出</td><td>k>0：k= 分配的内存块起始位置的地址
k=0：错误</td></tr>
<tr><td colspan="3">在使用 -xarch=v9 编译用于 64 位环境的程序时，返回 INTEGER*8 指针值。参见下面的说明。</td></tr>
</table>

注 – 该函数在 Fortran 95 中属于内函数，在 Fortran 77 中属于外函数。编译用于在 64 位环境下运行的 Fortran 77 程序将 `malloc()` 函数和接收其输出的变量声明为 `INTEGER*8`。提供的 `malloc64`(3F) 能够在 32 位环境与 64 位环境之间移植程序。

k = malloc64(*n*)			
n	INTEGER*8	输入	内存的字节数
返回值	INTEGER*8 *(Cray* POINTER*)*	输出	*k*>0：*k*= 分配的内存块起始位置的地址 *k*=0：错误

这些函数分配内存区域，并返回该区域起始位置的地址。（在 64 位环境中，返回的字节地址可能超出 `INTEGER*4` 数值范围 — 接收变量必须声明为 `INTEGER*8` 以免内存地址被截断。）内存区域没有以任何方式初始化，因此，不能假设内存已经预设为某个值，尤其不能假设为零！

示例：使用 `malloc()` 的代码片断：

```
      parameter (NX=1000)
      integer ( p2X, X )
      real*4 X(1)
      …
      p2X = malloc( NX*4 )
      if ( p2X .eq. 0 ) stop 'malloc: cannot allocate'
      do 11 i=1,NX
11         X(i) = 0.
      …
      end
```

在上面的示例中，我们获得了 4,000 字节的内存， `p2X` 指向该内存，并且内存已初始化为零。

1.39.2 重新分配内存：realloc

realloc() f95 内函数的调用方式如下所示：

k = realloc(*ptr*, *n*)			
ptr	INTEGER	输入	现有内存块的指针。（上一次调用 malloc() 或 realloc() 返回的值）。
n	INTEGER	输入	请求的新内存块大小 （以字节数表示）。
返回值	INTEGER *(Cray* POINTER*)*	输出	*k*>0：*k*= 分配的新内存块起始位置的地址 *k*=0：错误
	在使用 -xarch=v9 编译用于 64 位环境的程序时，返回 INTEGER*8 指针值。参见下面的说明。		

realloc() 函数将 ptr 指向的内存块大小更改为 n 个字节，并且返回 （可能已移动的）新内存块的指针。内存块的内容保存不变，新大小和旧大小中最小的值为内存块的大小。

如果 ptr 为零，realloc() 的行为与 malloc() 的行为相同，并且分配大小为 n 个字节的新内存块。

如果 n 为零并且 ptr 不为零，指向的内存块可以用于进一步分配，并且只有在终止应用程序时内存才返回给系统。

示例：使用 malloc() 和 realloc() 以及 Cray 式样的 POINTER 变量：

```
      PARAMETER (nsize=100001)
      POINTER (p2space,space)
      REAL*4 space(1)

      p2space = malloc(4*nsize)
      if(p2space .eq. 0) STOP 'malloc: cannot allocate space'
      ...
      p2space = realloc(p2space, 9*4*nsize)
      if(p2space .eq. 0) STOP 'realloc: cannot reallocate space'
      ...
      CALL free(p2space)
      ...
```

注意 realloc() 只适用于 f95。

1.39.3 free: 释放 Malloc 分配的内存

该子例程的调用方式如下所示：

call free (*ptr*)		
ptr	*Cray* POINTER	输入

free 释放以前由 malloc 和 realloc() 分配的内存区域。内存区域返回给内存管理器；用户程序不能再使用该内存区域。

示例：free()：

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

1.40 mvbits: 移动位字段

该子例程的调用方式如下所示：

call mvbits(*src*, *ini1*, *nbits*, *des*, *ini2*)			
src	INTEGER*4	输入	来源
ini1	INTEGER*4	输入	来源中初始位的位置
nbits	INTEGER*4	输入	要移动的位数
des	INTEGER*4	输出	目标
ini2	INTEGER*4	输入	目标中初始位的位置

示例：mvbits：

```
demo% cat mvb1.f
* mvb1.f -- From src, initial bit 0, move 3 bits to des, initial
*           bit 3.
*    src    des
* 543210 543210 ← Bit numbers
* 000111 000001 ← Values before move
* 000111 111001 ← Values after move
       INTEGER*4 src, ini1, nbits, des, ini2
       data src, ini1, nbits, des, ini2
     1    / 7,    0,     3,   1,    3 /
       call mvbits ( src, ini1, nbits, des, ini2 )
       write (*,"(5o3)") src, ini1, nbits, des, ini2
       end
demo% f95 mvb1.f
demo% a.out
  7  0  3 71  3
demo%
```

注意以下几点：

- 位的编号是从 0 到 31，按重要性从低到高排列。
- mvbits 只更改 *des* 位置中的位 *ini2* 到位 *ini2*+*nbits*-1，它不更改 *src* 位置中的位。
- 限制条件为：
  - *ini1* + *nbits* ≥ 32
  - *ini2* + *nbits* ≤ 32

1.41 perror, gerror, ierrno: 获取系统错误消息

这些例程执行以下函数：

perror	将消息打印到 Fortran 逻辑单元 0 stderr。
gerror	获取 （上一次检测的系统错误的）系统错误消息。
ierrno	获取上一次检测的系统错误的错误编号。

1.41.1 perror: 将消息打印到逻辑单元 0 stderr

该子例程的调用方式如下所示：

call perror(*string*)			
string	字符 **n*	输入	消息。它写在上一次检测的系统错误的标准错误消息前面。

示例 1：

```
    call perror( "file is for formatted I/O" )
```

1.41.2 gerror: 获取上一次检测的系统错误的消息

该子例程或函数的调用方式如下所示：

call gerror(*string*)			
string	字符 **n*	输出	上一次检测的系统错误的消息

示例 2：gerror() 用作子例程：

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

示例 3：gerror() 用作函数；未使用 *string*：

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

1.41.3 ierrno: 获取上一次检测的系统错误的编号

该函数的调用方式如下所示：

n = ierrno()			
返回值	INTEGER*4	输出	上一次检测的系统错误的编号

该数值只有在真正出现错误时才更新。可能会生成此类错误的大多数例程和 I/O 语句在调用之后返回错误代码；该值能够比较可靠地反映导致出现错误状况的原因。

示例 4：ierrno()：

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

参见 intro(2) 和 perror(3)。

注意：

- 调用 perror 时 *string* 的长度不能超过 127 个字符。
- gerror 返回的字符串长度由调用的程序决定。
- 《*Fortran 用户指南*》中列出了 f95 的运行时 I/O 错误代码。

1.42 putc, fputc: 将字符写入逻辑单元

putc 通常将控制终端输出写入逻辑单元 6。

fputc 写入逻辑单元。

这些函数绕过正常的 Fortran I/O，将字符写入与 Fortran 逻辑单元相关的文件中。

不要将正常的 Fortran 输出与相同单元中这些函数的输出混在一起。

注意：要写入任何特殊的 \ 换码符，例如，新行 '\n'，需要使用 -f77=backslash FORTRAN 77 兼容选项进行编译。

1.42.1 putc: 写入逻辑单元 6

该函数的调用方式如下所示：

INTEGER*4 putc *status* = putc(*char*)			
char	**字符**	输入	要写入单元的字符
返回值	INTEGER*4	输出	*status*=0：OK *status*>0：系统错误代码

示例：putc()：

```
demo% cat tputc.f
       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% f95 -f77=backslash tputc.f
demo% a.out
OK by putc
demo%
```

1.42.2 fputc: 写入指定的逻辑单元

该函数的调用方式如下所示：

INTEGER*4 fputc *status* = fputc(*lunit*,*char*)			
lunit	INTEGER*4	输入	要写入的单元
char	**字符**	输入	要写入单元的字符
返回值	INTEGER*4	输出	*status*=0：OK *status*>0：系统错误代码

示例：fputc()：

```
demo% cat tfputc.f
       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% f95 -f77=backslash tfputc.f
demo% a.out
demo% cat tfputc.data
OK by fputc
demo%
```

参见 putc(3S)、 intro(2) 和 perror(3F)。

1.43 qsort,qsort64: 对一维数组的元素进行排序

该子例程的调用方式如下所示：

call qsort(*array*, *len*, *isize*, *compar*) *（续）* call qsort64(*array*, *len8*, *isize8*, *compar*)			
array	array	输入	包含要排序的元素。
len	INTEGER*4	输入	数组中的元素数量。
len8	INTEGER*8	输入	数组中的元素数量。

isize	INTEGER*4	输入	元素的大小，通常： 4 代表整数或实数 8 代表双精度或复数 16 代表双复数 字符数组的字符对象长度
isize8	INTEGER*8	输入	元素的大小，通常： 4_8 代表整数或实数 8_8 代表双精度或复数 16_8 代表双复数 字符数组的字符对象长度
compar	函数名	输入	用户提供的 INTEGER*2 函数名称。 确定排序的顺序。 compar(*arg1,arg2*)

在 64 位环境中将 qsort64 用于大于 2 千兆字节的数组。确保指定数组长度 *len8* 和元素大小 *isize8*，作为 INTEGER*8 数据。使用 Fortran 95 式样的常量来显式指定 INTEGER*8 常量。

compar(*arg1, arg2*) 参数是 *array* 的元素，它返回以下值：

负数	*arg1* 排在 *arg2* 的前面
零	*arg1* 与 *arg2* 相当
正数	*arg1* 排在 *arg2* 的后面

例如：

```
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/,
     1           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% f95 tqsort.f
demo% a.out
  0 1 2 3 4 5 6 7 8 9
```

1.44 ran: 生成介于 0 和 1 之间的随机号

反复调用 ran 会生成均匀分布的随机号序列。参见 lcrans(3m)。

r = ran(*i*)			
i	INTEGER*4	输入	变量或数组元素
r	REAL	输出	变量或数组元素

示例：ran：

```
demo% cat ran1.f
* ran1.f -- Generate random numbers.
       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% f95 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%
```

注意以下几点：

- 该范围包括 0.0，但不包括 1.0。
- 该算法是一种倍增叠合型通用随机号生成器。
- 通常，在执行调用程序期间，将会设置*一次* i 的值。
- i 的初始值应该是大的奇整数。
- 每次调用 RAN 会获取序列中的下一个随机号。
- 要在每次运行程序时获取不同的随机号序列，必须将每次运行的参数设置为不同的初始值。
- RAN 使用该参数来存储根据以下算法计算出来的下一个随机号的值：

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

- SEED 包含 32 位数，高序位 24 位被转换为浮点，并且返回该值。

1.45 rand, drand, irand: 返回随机值

rand 返回 0.0 到 1.0 范围中的实数值。

drand 返回 0.0 到 1.0 范围中的双精度值。

irand 返回 0 到 2147483647 范围中的正整数。

这些函数使用 random(3) 来生成随机号序列。这三个函数共用同一个 256 字节的状态数组。这些函数的唯一优点是它们可以广泛地用于 UNIX 系统。要获得更好的随机号生成器，请比较 lcrans、addrans 和 shufrans。参见 random(3) 和《*数值计算指南*》。

i = irand(*k*) *r* = rand(*k*) *d* = drand(*k*)			
k	INTEGER*4	输入	*k*=0：获取序列中的下一个随机号 *k*=1：重新开始序列，返回第一个编号 *k*>0：用作新序列的种子，返回第一个编号
rand	REAL*4	输出	
drand	REAL*8	输出	
irand	INTEGER*4	输出	

示例：irand()：

```
demo% cat trand.f
       integer*4 v(5), iflag/0/
       do i = 1, 5
        v(i) = irand( iflag )
       end do
       write(*,*) v
       end
demo% f95 trand.f
demo% a.out
   2078917053 143302914 1027100827 1953210302 755253631
demo%
```

1.46 rename: 重命名文件

该函数的调用方式如下所示：

INTEGER*4 rename *status* = rename(*from*, *to*)			
from	字符 *n	输入	现有文件的路径名称
to	字符 *n	输入	文件的新路径名称
返回值	INTEGER*4	输出	*status*=0：OK *status*>0：系统错误代码

如果 *to* 指定的文件已经存在，则 *from* 和 *to* 必须属于相同的文件类型，并且必须位于相同的文件系统中。如果 *to* 已经存在，请先将其删除。

示例：rename() — 将文件 trename.old 重命名为 trename.new

```
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: error'
       end
demo% f95 trename.f
demo% ls trename*
trename.f trename.old
demo% a.out
demo% ls trename*
trename.f trename.new
demo%
```

参见 rename(2) 和 perror(3F)。

注意：路径名称的长度不能超过 <sys/param.h> 中规定的 MAXPATHLEN 值。

1.47 secnds: 获取以秒数表示的系统时间并减去参数

t = secnds(*t0*)			
t0	REAL	输入	常量、变量或数组元素
返回值	REAL	输出	自子夜起的秒数减去 *t0*

示例：secnds：

```
demo% cat sec1.f
       real elapsed, t0, t1, x, y
       t0 = 0.0
       t1 = secnds( t0 )
       y = 0.1
       do i = 1, 10000
        x = asin( y )
       end do
       elapsed = secnds( t1 )
       write ( *, 1 ) elapsed
  1    format ( ' 10000 arcsines: ', f12.6, ' sec' )
       end
demo% f95 sec1.f
demo% a.out
 10000 arcsines:     0.009064 sec
demo%
```

请注意：

- SECNDS 返回的值精确到 0.01 秒。
- 该值为系统时间，即从子夜起的秒数，并且它应该正确跨越午夜。
- 对于在一天快要结束时的短时间间隔，一些精度可能会丢失。

1.48 set_io_err_handler, get_io_err_handler: 设置并获取 I/O 错误处理程序

只要在指定的输入逻辑单元中检测到错误， set_io_err_handler() 就声明要调用的用户定义的例程。

get_io_err_handler() 返回当前声明的错误处理例程的地址。

这些例程为模块子例程，只有在 USE SUN_IO_HANDLERS 出现在调用例程中时，才能访问这些例程。

USE SUN_IO_HANDLERS call set_io_err_handler(*iu, subr_name, istat*)			
iu	INTEGER*8	输入	逻辑单元编号
subr_name	EXTERNAL	输入	用户提供的错误处理程序子例程的名称。
istat	INTEGER*4	输出	返回状态。

USE SUN_IO_HANDLERS call get_io_err_handler(*iu, subr_pointer, istat*)			
iu	INTEGER*8	输入	逻辑单元编号
subr_pointer	POINTER	输出	当前声明的处理程序例程的地址。
istat	INTEGER*4	输出	返回状态。

SET_IO_ERR_HANDLER 设置用户提供的子例程 *subr_name*，在出现输出错误时，该子例程用作逻辑单元 *iu* 的 I/O 错误处理程序。对于格式化的文件，*iu* 必须是连接的 Fortran 逻辑单元。如果有错误， *istat* 将设置为非零值，否则设置为零。

例如，在已经打开逻辑单元 *iu* 之前调用 SET_IO_ERR_HANDLER， *istat* 将被设置为 1001 （“非法的单元”）。如果 *subr_name* 为 NULL，用户错误处理将被关闭并且程序恢复到缺省的 Fortran 错误处理。

使用 GET_IO_ERR_HANDLER 来获取当前用作该逻辑单元错误处理程序的函数的地址。例如，调用 GET_IO_ERR_HANDLER 来保存当前的 I/O，然后再切换到另一个处理程序例程。以后您可以通过保存的值恢复错误处理程序。

subr_name 是用户提供的例程名称，用于处理逻辑单元 *iu* 的 I/O 错误。运行时 I/O 库将所有相关的信息传给 *subr_name*，使该例程可以诊断问题并且有可能修复错误，然后再继续运行。

用户提供的错误处理程序例程的接口如下所示：

```
SUBROUTINE SUB_NAME(UNIT, SRC_FILE, SRC_LINE, DATA_FILE, FILE_POS,
                    CURR_BUFF, CURR_ITEM, CORR_CHAR, CORR_ACTION )
INTENT (IN) UNIT, SRC_FILE, SRC_LINE, DATA_FILE
INTENT (IN) FILE_POS, CURR_BUFF, CURR_ITEM
INTENT (OUT) CORR_CHAR, CORR_ACTION
```

UNIT	INTEGER*8	输入	报告错误的输入文件的逻辑单元编号
SRC_FILE	字符 *(*)	输入	引起输入操作的 Fortran 源文件名称。
SRC_LINE	INTEGER*8	输入	有错误的输入操作的 SRC_FILE 中的行号。
DATA_FILE	字符 *(*)	输入	正在读取的数据文件名称。只有文件是打开的外部文件时才适用。如果名称不可用，（比方说逻辑单元 5）DATA_FILE 将被设置为零长度的字符数据项。
FILE_POS	INTEGER*8	输入	在输入文件中的当前位置 （以字节数表示）。只有知道 DATA_FILE 的名称时，才能定义位置。
CURR_BUFF	字符 *(*)	输入	包含输入记录中剩余数据的字符串。错误的输入字符是字符串中的第一个字符。

CURR_ITEM	INTEGER*8	输入	检测到错误时记录中已经读取的输入项数，包括当前的输入项。例如： READ(12,10)L,(ARR(I),I=1,L) 如果在这种情况下 CURR_ITEM 的值为 15，表示在读取 ARR 的第 14 个元素时出现错误， L 是第一项，ARR(1) 是第二项，以此类推。
CORR_CHAR	**字符**	输出	处理程序返回并且由用户提供的更正字符。只有在 CORR_ACTION 不为零时，才使用该值。如果 CORR_CHAR 是无效的字符，将会再次调用处理程序，直到返回有效的字符。这导致无止境循环，用户需要防止出现这种情形。
CORR_ACTION	INTEGER	输出	指定 I/O 库要采取的更正措施。如果值为零，不需要采取特殊措施，库将恢复到其缺省的错误处理例程。值为 1 使 CORR_CHAR 返回到 I/O 错误处理例程。

局限性

I/O 处理程序只能将一个字符换成另一个字符。它不能将一个字符换成多个字符。

错误恢复算法只能修复当前读取的错误字符，而不能修复在其它上下文环境中已经解释为有效字符的错误字符。例如，在进行列表控制的读取时，如果输入“1.234509.8765”，而正确的输入应该是 “1.2345 9.8765”， I/O 库将在第二阶段遇到错误，因为它不是有效的数字。但是，当时不可能返回并将 “0”更改为空白。

当前，这种错误处理功能不适用于由名称列表控制的输入。在进行由名称列表控制的输入时，如果出现错误，则不会调用指定的 I/O 错误处理程序。

只能为外部文件而不是内部文件设置 I/O 错误处理程序，这是因为没有与内部文件关联的逻辑单元。

调用 I/O 错误处理程序只能用于语法错误，不能用于系统错误或语义错误 （例如溢出的输入值）。

如果用户提供的 I/O 错误处理程序不断向 I/O 库提供错误的字符，导致反复调用用户提供的 I/O 错误处理程序，就有可能出现无止境的循环。如果在同一个文件位置反复出现错误，错误处理程序应该自行终止运行。解决这种问题的一种方法就是将 CORR_ACTION 设置为 0，采用缺省的错误路径。然后， I/O 库继续进行正确的错误处理。

1.49 sh: 快速执行 sh 命令

该函数的调用方式如下所示：

INTEGER*4 sh *status* = sh(*string*)			
string	字符 *n	输入	包含要执行的命令的字符串
返回值	INTEGER*4	输出	执行的 shell 的退出状态。有关该值的解释，请参见 *wait*(2)。

示例：sh()：

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

函数 sh 将 *string* 传给 sh shell 作为输入，就好象已经作为命令输入该字符串。

当前进程将等到命令终止。

派生的进程刷新所有打开的文件：

- 对于输出文件，缓冲区将刷新到实际文件中。
- 对于输入文件，无法预见指针的位置。

sh() 函数不能安全地用于多线程程序。不要从多线程或并行化程序中调用该函数。

参见：execve(2)、 wait(2) 和 system(3)。

注意：*string* 不能超过 1,024 个字符。

1.50 signal: 更改信号的操作

该函数的调用方式如下所示：

INTEGER*4 signal *或* INTEGER*8 signal *n* = signal(*signum*, *proc*, *flag*)			
signum	INTEGER*4	输入	信号编号，参见 *signal*(3)
proc	例程名称	输入	处理例程的用户信号名称；必须在外部语句中。
flag	INTEGER*4	输入	*flag* < 0：使用 *proc* 作为信号处理程序 *flag* ≥ 0：忽略 *proc*；传送 *flag* 作为操作： *flag* = 0：使用缺省的操作。 *flag* = 1：忽略该信号
返回值	INTEGER*4	输出	*n*=-1：系统错误 *n*>0：上一个操作的定义 *n*>1：*n*= 本来应该调用的例程地址 *n*<-1：如果 *signum* 是有效的信号编号：*n*= 本来应该调用的例程地址。如果 *signum* 并*不是*有效的信号编号：*n* 为错误编号。
	INTEGER*8		在 64 位环境中， signal 和接收其输出的变量必须声明为 INTEGER*8。

如果调用了 *proc*，则向 proc 传送信号编号，作为整数参数。

如果进程引发信号，缺省的操作通常是清理并终止。信号处理例程提供了捕捉特定异常或中断以供特殊处理的功能。

返回的值以后可以用于调用 signal，以便恢复以前的操作定义。

即使没有错误，您也有可能会获得负返回值。事实上，如果您将*有效的*信号编号传给 signal()，并且您获得的返回值小于 -1，则表示正常。

floatingpoint.h 定义 *proc* 值 SIGFPE_DEFAULT、 SIGFPE_IGNORE 和 SIGFPE_ABORT。参见第 1-44 页上的 “floatingpoint.h: Fortran IEEE 定义”。

在 64 位环境中， signal 以及接收其输出的变量必须声明为 INTEGER*8，以免可能返回的地址被截断。

参见 kill(1)、 signal(3) 和 kill(3F) 以及 《*数值计算指南*》。

1.51 sleep: 一段时间暂停执行

该子例程的调用方式如下所示：

call sleep(*itime*)			
itime	INTEGER*4	输入	要休止的秒数

由于系统计时粒度影响，实际时间最多比 *itime* 少 1 秒钟。

示例：sleep()：

```
        INTEGER*4 time / 5 /
        write(*,*) 'Start'
        call sleep( time )
        write(*,*) 'End'
        end
```

参见 sleep(3)。

1.52 stat, lstat, fstat: 获取文件状态

这些函数返回以下信息：

- 设备，
- 索引节点编号，
- 保护，
- 硬链接数，
- 用户 ID，
- 组 ID，
- 设备类型，
- 大小，
- 访问时间，
- 修改时间，
- 状态更改时间，
- 最佳的块大小，
- 分配的块

stat 和 lstat 都是按文件名查询。 fstat 按逻辑单元查询。

1.52.1 stat: 按文件名获取文件状态

该函数的调用方式如下所示：

INTEGER*4 stat *ierr* = stat (*name*, *statb*)			
name	**字符** **n*	输入	文件的名称
statb	INTEGER*4	输出	文件的状态结构， 13-元素数组
返回值	INTEGER*4	输出	*ierr*=0：OK *ierr*>0：错误代码

示例 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: error'
      write(*,*)'UID of owner = ',statb(5),',
     1   blocks = ',statb(13)
      end
```

1.52.2 fstat: 按逻辑单元获取文件状态

函数

INTEGER*4 fstat *ierr* = fstat (*lunit*, *statb*)			
lunit	INTEGER*4	输入	逻辑单元编号
statb	INTEGER*4	输出	文件的状态：13-元素数组
返回值	INTEGER*4	输出	*ierr*=0：OK *ierr*>0：错误代码

的调用方式如下所示：

示例 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: error'
      write(*,*)'UID of owner = ',statb(5),',
     1      blocks = ',statb(13)
      end
```

1.52.3 lstat: 按文件名获取文件状态

该函数的调用方式如下所示：

ierr = lstat (*name*, *statb*)			
name	**字符** **n*	输入	文件名
statb	INTEGER*4	输出	文件夹的状态数组， 13 个元素
返回值	INTEGER*4	输出	*ierr*=0：OK *ierr*>0：错误代码

示例 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: error'
      write(*,*)'UID of owner = ',statb(5),',
     1   blocks = ',statb(13)
      end
```

1.52.4 文件状态数组的详细信息

INTEGER*4 数组 *statb* 中返回的信息含义在 stat(2) 下面的 *stat* 结构中已经作了介绍。备用值不包括在内。顺序如下表所示：

statb(1)	索引节点所在的设备
statb(2)	该索引节点的编号
statb(3)	保护
statb(4)	文件的硬链接数
statb(5)	属主的用户 ID
statb(6)	属主的组 ID
statb(7)	属于设备的索引节点的设备类型
statb(8)	文件的总大小
statb(9)	上次访问文件的时间
statb(10)	上次修改文件的时间
statb(11)	上次更改文件状态的时间
statb(12)	文件系统 I/O 操作的最佳块大小
statb(13)	分配的实际块数

参见 stat(2)、 access(3F)、 perror(3F) 和 time(3F)。

注意：路径名称的长度不能超过 <sys/param.h> 中规定的 MAXPATHLEN 值。

1.53 stat64, lstat64, fstat64: 获取文件状态

stat、lstat 和 fstat 的 64 位“长文件”版本。除了 13 元素数组 *statb* 必须声明为 INTEGER*8 之外，这些例程与非 64 位例程相同。

1.54 system: 执行系统命令

该函数的调用方式如下所示：

INTEGER*4 system *status* = system(*string*)			
string	字符 *n	输入	包含要执行的命令的字符串
返回值	INTEGER*4	输出	执行的 shell 的退出状态。有关该值的解释，请参见 *wait*(2)。

示例：system()：

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

函数 system 将 *string* 作为输入传送给 shell，如同将该字符串作为命令输入。
注意：*string* 不能超过 1024 个字符。

如果 system 找到环境变量 SHELL，则 system 将 SHELL 的值用作命令解释程序 (shell)；否则它使用 sh(1)。

当前进程将等到命令终止。

传统上， cc 开发时使用的是不同的假设：

- 如果 cc 调用 system， shell 始终是 Bourne shell。

system 函数刷新所有打开的文件：

- 对于输出文件，缓冲区将刷新到实际文件中。
- 对于输入文件，无法预见指针的位置。

参见：execve(2)、 wait(2) 和 system(3)。

system() 函数不能安全地用于多线程程序。不要从多线程或并行化程序中调用该函数。

1.55 time, ctime, ltime, gmtime: 获取系统时间

这些例程具有以下函数：

time	标准版本：获取以整数表示的系统时间 （从 70 年 1 月 1 日 GMT 0 时至今的秒数） VMS 版本：获取以字符表示的系统时间 (hh:mm:ss)
ctime	将系统时间转换为 ASCII 字符。
ltime	将系统时间分解成当地时间的月份、日期等等。
gmtime	将系统时间分解成 GMT 时间的月份、日期等等。

1.55.1 time: 获取系统时间

time() 函数的调用方式如下所示：

INTEGER*4 time *或* INTEGER*8 *n* = time() *标准版本*			
返回值	INTEGER*4	输出	自 70 年 1 月 1 日 GMT 0:0:0 时起至今的时间 （以秒数为单位）
	INTEGER*8	输出	在 64 位环境中， time 返回 INTEGER*8 值。

函数 time() 返回自 1970 年 1 月 1 日 GMT 00:00:00 起至今的时间整数，单位是秒数。这是操作系统时钟的值。

示例：time()，操作系统的标准版本：

```
demo% cat ttime.f
        INTEGER*4  n, time
        n = time()
        write(*,*) 'Seconds since 0 1/1/70 GMT = ', n
        end
demo% f95 ttime.f
demo% a.out
 Seconds since 0 1/1/70 GMT =   913240205
demo%
```

1.55.2 ctime: 将系统时间转换为字符

函数 ctime 转换系统时间 *stime*，并返回 24 个字符组成的 ASCII 字符串。

该函数的调用方式如下所示：

字符 ctime*24 *string* = ctime(*stime*)			
stime	INTEGER*4	输入	time() 的系统时间 （标准版本）
返回值	**字符** *24	输出	作为字符串的系统时间。将 ctime 和 *string* 声明为**字符** *24。

下面的例子中显示了 ctime 返回值的格式。在**手册**页 ctime(3C) 中对此作了说明。

示例：ctime()：

```
demo% cat tctime.f
        character*24 ctime, string
        INTEGER*4  n, time
        n = time()
        string = ctime( n )
        write(*,*) 'ctime: ', string
        end
demo% f95 tctime.f
demo% a.out
 ctime: Wed Dec  9 13:50:05 1998
demo%
```

1.55.3 ltime: 将系统时间分解成月份、日期等等 （当地时间）

将系统时间分解成当地时区的月份、日期等等。

该子例程的调用方式如下所示：

call ltime(*stime*, *tarray*)			
stime	INTEGER*4	输入	time() 的系统时间 （标准版本）
tarray	INTEGER*4(9)	输出	系统时间 （对本地时区），分解为年份、月份、日期等等

有关 tarray 中元素的含义，请参见下面一节。

示例：ltime()：

```
demo% cat tltime.f
        integer*4  stime, tarray(9), time
        stime = time()
        call ltime( stime, tarray )
        write(*,*) 'ltime: ', tarray
        end
demo% f95 tltime.f
demo% a.out
 ltime: 25 49 10 12 7 91 1 223 1
demo%
```

1.55.4 gmtime: 将系统时间分解成月份、日期等等 (GMT)

该例程将系统时间分解成 GMT 时间的月份、日期等等。

此子例程为：

call gmtime(*stime*, *tarray*)			
stime	INTEGER*4	输入	time() 的系统时间 （标准版本）
tarray	INTEGER*4(9)	输出	系统时间， GMT，作为年份、月份、日期等等

示例：gmtime：

```
demo% cat tgmtime.f
        integer*4  stime, tarray(9), time
        stime = time()
        call gmtime( stime, tarray )
        write(*,*) 'gmtime: ', tarray
        end
demo% f95t tgmtime.f
demo% a.out
 gmtime:   12  44  19  18  5  94  6  168  0
demo%
```

下面是 ltime 和 gmtime 的 tarray() 值：索引、单位和范围：

1	秒 (0 - 61)	6	年份－ 1900
2	分钟 (0 - 59)	7	星期 （星期日 = 0）
3	小时 (0 - 23)	8	一年中的天数 (0 - 365)
4	一个月中的天数 (1 - 31)	9	夏令时，如果实行夏令时，范围为 1。
5	自一月起的月份 (0 - 11)		

这些值通过 C 库例程 ctime(3C) 定义，它解释了为什么系统返回的秒数*大于* 59。参见：idate(3F) 和 fdate(3F)。

1.55.5 ctime64, gmtime64, ltime64: 64 位环境的系统时间例程

这些是对应的例程 ctime、gmtime 和 ltime 版本，它们能够在 64 位环境中具有可移植性。除了输入变量 *stime* 必须是 INTEGER*8 之外，它们与这些例程相同。

在 32 位环境中与 INTEGER*8 *stime* 一起使用时，如果 *stime* 的值超出 INTEGER*4 范围， ctime64 全部返回星号，而 gmtime 和 ltime 在 tarray 数组中填入 -1。

1.56 ttynam, isatty: 获取终端端口的名称

ttynam 和 isatty 处理终端端口名称。

1.56.1 ttynam: 获取终端端口的名称

函数 ttynam 为与逻辑单元 *lunit* 相关的终端设备返回填充了空白的路径名称。

该函数的调用方式如下所示：

字符 ttynam*24 *name* = ttynam(*lunit*)			
lunit	INTEGER*4	输入	逻辑单元
返回值	**字符** **n*	输出	如果返回非空白字符串：*name*= *lunit* 中设备的路径名称。大小 *n* 必须足够大以便能够容纳最长的路径。 如果返回空白字符串 （全部为空白）：*lunit* 不与目录 /dev 中的终端设备关联。

1.56.2 isatty: 该单元是否为终端？

函数 isatty 根据逻辑单元 *lunit* 是否为终端设备返回 True 或 False。

该函数的调用方式如下所示：

terminal = isatty(*lunit*)			
lunit	INTEGER*4	输入	逻辑单元
返回值	LOGICAL*4	输出	*terminal*=true：它是终端设备 *terminal*=false：它*不是*终端设备

示例：确定 *lunit* 是否为 tty：

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

输出为：

```
 terminal = T, name = "/dev/ttyp1   "
```

1.57 unlink: 删除文件

该函数的调用方式如下所示：

INTEGER*4 unlink *n* = unlink (*patnam*)			
patnam	**字符** **n*	输入	文件名
返回值	INTEGER*4	输出	*n*=0：OK *n*>0：错误

函数 unlink 删除路径名称 *patnam* 指定的文件。如果这是文件的最后一个链接，文件的内容将会丢失。

示例：unlink() — 删除 tunlink.data 文件：

```
demo% cat tunlink.f
       call unlink( 'tunlink.data' )
       end
demo% f95 tunlink.f
demo% ls tunl*
tunlink.f tunlink.data
demo% a.out
demo% ls tunl*
tunlink.f
```

参见：unlink(2)、 link(3F) 和 perror(3F)。注意：路径名称的长度不能超过 <sys/param.h> 中规定的 MAXPATHLEN 值。

1.58 wait: 等待终止进程

函数为：

INTEGER*4 wait *n* = wait(*status*)			
status	INTEGER*4	输出	子进程的终止状态
返回值	INTEGER*4	输出	*n*>0：子进程的进程 ID。 *n*<0：*n*= 系统错误代码；参见 wait(2)。

wait 暂停调用程序，直到收到信号或者某个子进程终止。如果自上一次执行 wait 函数后任何子进程已终止，则立即返回子进程 ID。如果没有子进程，则会立即返回错误代码。

示例：使用 wait() 的代码片断：

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

参见：wait(2)、 signal(3F)、 kill(3F) 和 perror(3F)。

第 2 章

Fortran 95 内函数

本章列出了 f95 编译器能够识别的内函数名称。

2.1 标准 Fortran 95 的通用内函数

本节根据通用的 Fortran 95 内函数在 Fortran 95 标准中出现时的功能性将这些函数分组。

显示的参数为在使用关键字形式时可以用作参数关键字的名称，如 `cmplx(Y=B, KIND=M, X=A)` 中所示。

有关这些通用内过程的详细说明，请查阅 Fortran 95 标准。

2.1.1 参数存在查询函数

通用内函数名称	描述
PRESENCE	存在参数

2.1.2 数值函数

通用内函数名称	描述
ABS (A)	绝对值
AIMAG (Z)	复数的虚部
AINT (A [, KIND])	整数截尾
ANINT (A [, KIND])	最近的整数
CEILING (A [, KIND])	大于或等于数值的最小整数
CMPLX (X [, Y, KIND])	转换为复数类型
CONJG (Z)	共轭复数
DBLE (A)	转换为双精度实数类型
DIM (X, Y)	正差数
DPROD (X, Y)	双精度实数乘积
FLOOR (A [, KIND])	小于或等于数值的最大整数
INT (A [, KIND])	转换为整数类型
MAX (A1, A2 [, A3,...])	最大值
MIN (A1, A2 [, A3,...])	最小值
MOD (A, P)	余数函数
MODULO (A, P)	模数函数
NINT (A [, KIND])	最近的整数
REAL (A [, KIND])	转换为实数类型
SIGN (A, B)	符号传输

2.1.3 数学函数

通用内函数名称	描述
ACOS (X)	反余弦
ASIN (X)	反正弦
ATAN (X)	反正切

通用内函数名称	描述
ATAN2 (Y, X)	反正切
COS (X)	余弦
COSH (X)	双曲余弦
EXP (X)	指数
LOG (X)	自然对数
LOG10 (X)	常用对数 （10 为基数）
SIN (X)	正弦
SINH (X)	双曲正弦
SQRT (X)	平方根
TAN (X)	正切
TANH (X)	双曲正切

2.1.4 字符函数

通用内函数名称	描述
ACHAR (I)	ASCII 排序序列中给定位置的字符
ADJUSTL (STRING)	齐左调整
ADJUSTR (STRING)	齐右调整
CHAR (I [, KIND])	处理器整理序列中某个位置的字符
IACHAR (C)	ASCII 整理序列中某个字符的位置
ICHAR (C)	处理器整理序列中某个字符的位置
INDEX (STRING, SUBSTRING [, BACK])	子串的起始位置
LEN_TRIM (STRING)	不包含结尾空白字符的字符串长度
LGE (STRING_A, STRING_B)	词法上大于或等于
LGT (STRING_A, STRING_B)	词法上大于
LLE (STRING_A, STRING_B)	词法上小于或等于
LLT (STRING_A, STRING_B)	词法上小于
REPEAT (STRING, NCOPIES)	重复并置

通用内函数名称	描述
SCAN (STRING, SET [, BACK])	扫描字符串以查找集中的某个字符
TRIM (STRING)	删除结尾的空白字符
VERIFY (STRING, SET [, BACK])	检验字符串中的字符集

2.1.5 字符查询函数

通用内函数名称	描述
LEN (STRING)	字符实体的长度

2.1.6 种类函数

通用内函数名称	描述
KIND (X)	种类类型参数值
SELECTED_INT_KIND (R)	指定范围的整数种类类型参数
SELECTED_REAL_KIND ([P, R])	指定精度和范围的实数种类类型参数值

2.1.7 逻辑函数

通用内函数名称	描述
LOGICAL (L [, KIND])	在种类类型参数不相同的逻辑类型对象之间转换

2.1.8 数值查询函数

通用内函数名称	描述
DIGITS (X)	模型的有效数字数
EPSILON (X)	与此相比几乎可以忽略的数值
HUGE (X)	模型中最大的数值
MAXEXPONENT (X)	模型的最大指数
MINEXPONENT (X)	模型的最小指数
PRECISION (X)	十进制精度
RADIX (X)	模型的基数
RANGE (X)	十进制指数范围
TINY (X)	模型中最小的正数

2.1.9 位查询函数

通用内函数名称	描述
BIT_SIZE (I)	模型中的位数

2.1.10 位操作函数

通用内函数名称	描述
BTEST (I, POS)	位测试
IAND (I, J)	逻辑 AND
IBCLR (I, POS)	清除位
IBITS (I, POS, LEN)	位提取
IBSET (I, POS)	设置位
IEOR (I, J)	互斥 OR

通用内函数名称	描述
IOR (I, J)	包容 OR
ISHFT (I, SHIFT)	逻辑移位
ISHFTC (I, SHIFT [, SIZE])	循环移位
NOT (I)	逻辑补充

2.1.11 传送函数

通用内函数名称	描述
TRANSFER (SOURCE, MOLD [, SIZE])	处理第一个参数，就好象它与第二个参数属于同一种类型

2.1.12 浮点处理函数

通用内函数名称	描述
EXPONENT (X)	型号的指数部分
FRACTION (X)	数值的小数部分
NEAREST (X, S)	指定的方向最近的不同处理器
RRSPACING (X)	接近指定数值的型号相对间隔的倒数
SCALE (X, I)	实数乘以基数得出整数幂
SET_EXPONENT (X, I)	设置数值的指数部分
SPACING (X)	接近指定数值的型号的绝对间隔

2.1.13 向量和矩阵乘法函数

通用内函数名称	描述
DOT_PRODUCT (VECTOR_A, VECTOR_B)	两个一级数组的点乘积
MATMUL (MATRIX_A, MATRIX_B)	矩阵乘法

2.1.14 约简数组函数

通用内函数名称	描述
ALL (MASK [, DIM])	如果所有的值为 True 则为 True。
ANY (MASK [, DIM])	如果任意值为 True 则为 True。
COUNT (MASK [, DIM])	数组中 True 元素数
MAXVAL (ARRAY, DIM [, MASK]) **或** MAXVAL (ARRAY [, MASK])	数组中的最大值
MINVAL (ARRAY, DIM [, MASK]) **或** MINVAL (ARRAY [, MASK])	数组中的最小值
PRODUCT (ARRAY, DIM [, MASK]) **或** PRODUCT (ARRAY [, MASK])	数组元素的乘积
SUM (ARRAY, DIM [, MASK]) **或** SUM (ARRAY [, MASK])	数组元素的求和

2.1.15 数组查询函数

通用内函数名称	描述
ALLOCATED (ARRAY)	数组分配状态
LBOUND (ARRAY [, DIM])	数组的维数下界
SHAPE (SOURCE)	数组或标量的形式
SIZE (ARRAY [, DIM])	数组中的元素总数
UBOUND (ARRAY [, DIM])	数组的维数上界

2.1.16 数组构造函数

通用内函数名称	描述
MERGE (TSOURCE, FSOURCE, MASK)	在屏蔽下合并
PACK (ARRAY, MASK [, VECTOR])	在屏蔽下将数组压缩为一级数组
SPREAD (SOURCE, DIM, NCOPIES)	增加维数以复制数组
UNPACK (VECTOR, MASK, FIELD)	在屏蔽下将一级数组解压缩为数组

2.1.17 数组整形函数

通用内函数名称	描述
RESHAPE (SOURCE, SHAPE[, PAD, ORDER])	数组整形

2.1.18 数组处理函数

通用内函数名称	描述
CSHIFT (ARRAY, SHIFT [, DIM])	循环移位
EOSHIFT (ARRAY, SHIFT [, BOUNDARY, DIM]	结束移位
TRANSPOSE (MATRIX)	调换两级数组

2.1.19 数组位置函数

通用内函数名称	描述
MAXLOC (ARRAY, DIM [, MASK]) 或 MAXLOC (ARRAY [, MASK])	数组中最大值的位置
MINLOC (ARRAY, DIM [, MASK]) 或 MINLOC (ARRAY [, MASK])	数组中最小值的位置

2.1.20 指针关联状态函数

通用内函数名称	描述
ASSOCIATED (POINTER [, TARGET])	关联状态查询或比较
NULL ([MOLD])	返回分离的指针

2.1.21 内子例程

通用内函数名称	描述
CPU_TIME (TIME)	获取处理器的时间
DATE_AND_TIME ([DATE, TIME, ZONE, VALUES])	获取日期和时间
MVBITS (FROM, FROMPOS, LEN, TO, TOPOS)	将位从一个整数复制到另一个整数
RANDOM_NUMBER (HARVEST)	返回伪随机数值
RANDOM_SEED ([SIZE, PUT, GET])	初始化或重新启动伪随机数据产生器
SYSTEM_CLOCK ([COUNT, COUNT_RATE, COUNT_MAX])	从系统时钟中获取数据

2.1.22 内函数的专用名称

表 2-1 Fortran 95 内函数的专用名称和通用名称

专用名称	通用名称	参数类型
ABS (A)	ABS (A)	缺省实数
ACOS (X)	ACOS (X)	缺省实数
AIMAG (Z)	AIMAG (Z)	缺省复数
AINT (A)	AINT (A)	缺省实数
ALOG (X)	LOG (X)	缺省实数

表 **2-1** Fortran 95 内函数的专用名称和通用名称 *（续）*

	专用名称	通用名称	参数类型
	ALOG10 (X)	LOG10 (X)	缺省实数
#	AMAX0 (A1, A2 [, A3,...])	REAL (MAX (A1, A2 [, A3,...]))	缺省整数
#	AMAX1 (A1, A2 [, A3,...])	MAX (A1, A2 [, A3,...])	缺省实数
#	AMIN0 (A1, A2 [, A3,...])	REAL (MIN (A1, A2 [, A3,...]))	缺省整数
#	AMIN1 (A1, A2 [, A3,...])	MIN (A1, A2 [, A3,...])	缺省实数
	AMOD (A, P)	MOD (A, P)	缺省实数
	ANINT (A)	ANINT (A)	缺省实数
	ASIN (X)	ASIN (X)	缺省实数
	ATAN (X)	ATAN (X)	缺省实数
	ATAN2 (Y, X)	ATAN2 (Y, X)	缺省实数
	CABS (A)	ABS (A)	缺省复数
	CCOS (X)	COS (X)	缺省复数
	CEXP (X)	EXP (X)	缺省复数
#	CHAR (I)	CHAR (I)	缺省整数
	CLOG (X)	LOG (X)	缺省复数
	CONJG (Z)	CONJG (Z)	缺省复数
	COS (X)	COS (X)	缺省实数
	COSH (X)	COSH (X)	缺省实数
	CSIN (X)	SIN (X)	缺省复数
	CSQRT (X)	SQRT (X)	缺省复数
	DABS (A)	ABS (A)	双精度
	DACOS (X)	ACOS (X)	双精度
	DASIN (X)	ASIN (X)	双精度
	DATAN (X)	ATAN (X)	双精度
	DATAN2 (Y, X)	ATAN2 (Y, X)	双精度
	DCOS (X)	COS (X)	双精度
	DCOSH (X)	COSH (X)	双精度
	DDIM (X, Y)	DIM (X, Y)	双精度
	DEXP (X)	EXP (X)	双精度

表 2-1 Fortran 95 内函数的专用名称和通用名称 *（续）*

	专用名称	通用名称	参数类型
	DIM (X, Y)	DIM (X, Y)	缺省实数
	DINT (A)	AINT (A)	双精度
	DLOG (X)	LOG (X)	双精度
	DLOG10 (X)	LOG10 (X)	双精度
#	DMAX1 (A1, A2 [, A3,...])	MAX (A1, A2 [, A3,...])	双精度
#	DMIN1 (A1, A2 [, A3,...])	MIN (A1, A2 [, A3,...])	双精度
	DMOD (A, P)	MOD (A, P)	双精度
	DNINT (A)	ANINT (A)	双精度
	DPROD (X, Y)	DPROD (X, Y)	缺省实数
	DSIGN (A, B)	SIGN (A, B)	双精度
	DSIN (X)	SIN (X)	双精度
	DSINH (X)	SINH (X)	双精度
	DSQRT (X)	SQRT (X)	双精度
	DTAN (X)	TAN (X)	双精度
	DTANH (X)	TANH (X)	双精度
	EXP (X)	EXP (X)	缺省实数
#	FLOAT (A)	REAL (A)	缺省整数
	IABS (A)	ABS (A)	缺省整数
#	ICHAR (C)	ICHAR (C)	缺省字符
	IDIM (X, Y)	DIM (X, Y)	缺省整数
#	IDINT (A)	INT (A)	双精度
	IDNINT (A)	NINT (A)	双精度
#	IFIX (A)	INT (A)	缺省实数
	INDEX (STRING, SUBSTRING)	INDEX (STRING, SUBSTRING)	缺省字符
#	INT (A)	INT (A)	缺省实数
	ISIGN (A, B)	SIGN (A, B)	缺省整数
	LEN (STRING)	LEN (STRING)	缺省字符
#	LGE (STRING_A, STRING_B)	LGE (STRING_A, STRING_B)	缺省字符
#	LGT (STRING_A, STRING_B)	LGT (STRING_A, STRING_B)	缺省字符

表 **2-1** Fortran 95 内函数的专用名称和通用名称 *（续）*

	专用名称	通用名称	参数类型
#	LLE (STRING_A, STRING_B)	LLE (STRING_A, STRING_B)	缺省字符
#	LLT (STRING_A, STRING_B)	LLT (STRING_A, STRING_B)	缺省字符
#	MAX0 (A1, A2 [, A3,...])	MAX (A1, A2 [, A3,...])	缺省整数
#	MAX1 (A1, A2 [, A3,...])	INT (MAX (A1, A2 [, A3,...]))	缺省实数
#	MIN0 (A1, A2 [, A3,...])	MIN (A1, A2 [, A3,...])	缺省整数
#	MIN1 (A1, A2 [, A3,...])	INT (MIN (A1, A2 [, A3,...]))	缺省实数
	MOD (A, P)	MOD (A, P)	缺省整数
	NINT (A)	NINT (A)	缺省实数
#	REAL (A)	REAL (A)	缺省整数
	SIGN (A, B)	SIGN (A, B)	缺省实数
	SIN (X)	SIN (X)	缺省实数
	SINH (X)	SINH (X)	缺省实数
#	SNGL (A)	REAL (A)	双精度
	SQRT (X)	SQRT (X)	缺省实数
	TAN (X)	TAN (X)	缺省实数
	TANH (X)	TANH (X)	缺省实数

标有 # 号的函数不能用作实际参数。
“双精度”表示双精度实数。

2.2 Fortran 2000 模块例程

Fortran 2000 标准草案提供了一组内模块，它们定义了支持 IEEE 算术以及与 C 语言的互操作性而需要具备的一些特点。这些模块定义了新的函数和子例程，并且通过 Sun Studio 8 Fortran 95 编译器实现。

2.2.1 IEEE 算术和异常模块

Fortran 2000 标准草案内模块 `IEEE_EXCEPTIONS`、 `IEEE_ARITHMETIC` 和 `IEEE_FEATURES` 支持建议的语言标准中的新特点，从而支持 IEEE 算术和 IEEE 异常处理。

标准草案定义了一组查询函数、基本函数、种类函数、基本子例程和非基本子例程。下表中列出了这些函数和子例程。

要访问这些函数和子例程，调用例程必须包括

```
USE, INTRINSIC :: IEEE_ARITHMETIC, IEEE_EXCEPTIONS
```

有关详细信息，请参阅标准草案 (`http://www.j3-fortran.org`) 的第 14 章。

2.2.1.1 查询函数

模块 `IEEE_EXCEPTIONS` 包含以下查询函数。

函数	描述
`IEEE_SUPPORT_FLAG(FLAG[,X])`	查询处理器是否支持异常。
`IEEE_SUPPORT_HALTING(FLAG)`	查询处理器是否支持在出现异常后控制停止异常。

模块 `IEEE_ARITHMETIC` 包含以下查询函数。

函数	描述
`IEEE_SUPPORT_DATATYPE([X])`	查询处理器是否支持 IEEE 算术。
`IEEE_SUPPORT_DENORMAL([X])`	查询处理器是否支持非正常化的数值。
`IEEE_SUPPORT_DIVIDE([X])`	查询处理器是否支持根据 IEEE 标准规定的准确性进行分类。
`IEEE_SUPPORT_INF([X])`	查询处理器是否支持 IEEE 无穷大。

函数	描述
IEEE_SUPPORT_IO([X])	查询处理器是否在格式化输入/输出期间支持 IEEE 基本转换舍入。
IEEE_SUPPORT_NAN([X])	查询处理器是否支持 IEEE 非数值。
IEEE_SUPPORT_ROUNDING(VAL[,X])	查询处理器是否支持特定的舍入模式。
IEEE_SUPPORT_SQRT([X])	查询处理器是否支持 IEEE 平方根。
IEEE_SUPPORT_STANDARD([X])	查询处理器是否支持所有的 IEEE 功能。

2.2.1.2 基本函数

模块 IEEE_ARITHMETIC 包含实数 X 和 Y 的以下基本函数，并且这些函数的 IEEE_SUPPORT_DATATYPE(X) 和 IEEE_SUPPORT_DATATYPE(Y) 为 True。

函数	描述
IEEE_CLASS(X)	IEEE 类。
IEEE_COPY_SIGN(X,Y)	IEEE 复制符号函数。
IEEE_IS_FINITE(X)	确定值是否为有限值。
IEEE_IS_NAN(X)	确定值是否为 IEEE 非数值
IEEE_IS_NORMAL(X)	确定值是否正常。
IEEE_IS_NEGATIVE(X)	确定值是否为负数。
IEEE_LOGB(X)	采用 IEEE 浮点格式的无偏指数。
IEEE_NEXT_AFTER(X,Y)	按朝着 Y 的方向返回 X 的下一个可表示的相邻数。
IEEE_REM(X,Y)	IEEE REM 余数函数 X - Y*N，其中 N 是最接近 X/Y 精确值的整数。
IEEE_RINT(X)	根据当前的舍入模式舍入为整数值。
IEEE_SCALB(X,I)	返回 X*2**I
IEEE_UNORDERED(X,Y)	IEEE 无序函数。如果 X 或 Y 为 NaN 则为 True，否则为 False。
IEEE_VALUE(X,CLASS)	生成 IEEE 值。

2.2.1.3 种类函数

模块 IEEE_ARITHMETIC 包含以下转换函数：

函数	描述
IEEE_SELECTED_REAL_KIND([P,][R])	具有指定精度和范围的 IEEE 实数的种类类型参数值。

2.2.1.4 基本子例程

模块 IEEE_EXCEPTIONS 包含以下基本子例程。

子例程	描述
IEEE_GET_FLAG(FLAG,FLAG_VALUE)	获取异常标志。
IEEE_GET_HALTING_MODE(FLAG,HALTING)	获取异常的停止模式。

2.2.1.5 非基本子例程

模块 IEEE_EXCEPTIONS 包含以下非基本子例程。

子例程	描述
IEEE_GET_STATUS(STATUS_VALUE)	获取浮点环境的当前状态。
IEEE_SET_FLAG(FLAG,FLAG_VALUE)	设置异常标志。
IEEE_SET_HALTING_MODE(FLAG,HALTING)	控制异常持续或停止。
IEEE_SET_STATUS(STATUS_VALUE)	恢复浮点环境的状态。

模块 IEEE_ARITHMETIC 包含以下非基本子例程。

子例程	描述
IEEE_GET_ROUNDING_MODE(ROUND_VAL)	获取当前的 IEEE 舍入模式。
IEEE_SET_ROUNDING_MODE(ROUND_VAL)	设置当前的 IEEE 舍入模式。

2.2.2 C 绑定模块

Fortran 2000 标准草案提供了一种引用 C 语言过程的方式。 ISO_C_BINDING 模块将三种支持过程定义为内模块函数。访问这些函数需要在调用的例程中使用

```
USE, INTRINSIC :: ISO_C_BINDING, ONLY: C_LOC, C_PTR, C_ASSOCIATED
```

该模块中定义的过程为

函数	描述
C_LOC(X)	返回参数的 C 地址
C_ASSOCIATED(C_PTR_1 [, C_PTR_2])	表示 C_PTR_1 的关联状态，或者表示 C_PTR_1 和 C_PTR_2 是否与同一个实体关联。
C_F_POINTER(CPTR, FPTR [, SHAPE])	将指针与 C 指针的目标关联并指定其形式。

有关 ISO_C_BINDING 内模块的详细信息，请参阅位于 http://www.j3-fortran.org/. 中的 Fortran 2000 标准草案的第 15 章。

2.3 非标准 Fortran 95 内函数

以下函数被 f95 编译器视为内函数，但是不属于 Fortran 95 标准。

2.3.1 基本线性代数函数 (BLAS)

在使用 -xknown_lib=blas 编译时，编译器将对以下例程的调用识别为内函数并进行优化，从而链接到实现 Sun 性能库。编译器会忽略用户提供的这些例程版本。

表 2-2 BLAS 内函数

函数	描述
CAXPY	标量和向量的乘积并加上向量
DAXPY	
SAXPY	
ZAXPY	

表 2-2 BLAS 内函数 *（续）*

函数	描述
CCOPY	复制向量
DCOPY	
SCOPY	
ZCOPY	
CDOTC	点乘积（内部乘积）
CDOTU	
DDOT	
SDOT	
ZDOTC	
ZDOTU	
CSCAL	按比例缩放向量
DSCAL	
SSCAL	
ZSCAL	

有关这些例程的详细信息，请参见《*Sun Performance Library User's Guide*》。

2.3.2 区间运算内函数

下表列出了编译器在编译区间运算时 (`-xia`) 识别的内函数。有关详细信息，请参见《*Fortran 95 Interval Arithmetic 编程参考*》。

DINTERVAL	DIVIX	INF	INTERVAL
ISEMPTY	MAG	MID	MIG
NDIGITS	QINTERVAL	SINTERVAL	SUP
VDABS	VDACOS	VDASIN	VDATAN
VDATAN2	VDCEILING	VDCOS	VDCOSH
VDEXP	VDFLOOR	VDINF	VDINT
VDISEMPTY	VDLOG	VDLOG10	VDMAG
VDMID	VDMIG	VDMOD	VDNINT
VDSIGN	VDSIN	VDSINH	VDSQRT
VDSUP	VDTAN	VDTANH	VDWID
VQABS	VQCEILING	VQFLOOR	VQINF

VQINT	VQISEMPTY	VQMAG	VQMID
VQMIG	VQNINT	VQSUP	VQWID
VSABS	VSACOS	VSASIN	VSATAN
VSATAN2	VSCEILING	VSCOS	VSCOSH
VSEXP	VSFLOOR	VSINF	VSINT
VSISEMPTY	VSLOG	VSLOG10	VSMAG
VSMID	VSMIG	VSMOD	VSNINT
VSSIGN	VSSIN	VSSINH	VSSQRT
VSSUP	VSTAN	VSTANH	VSWID
WID			

2.3.3 其它供应商的内函数

`f95` 编译器能够识别包括 Cray Research, Inc. 在内的其它供应商的 Fortran 编译器定义的各种传统内函数。这些函数已经过时，应该尽量避免使用这些函数。

表 2-3 Cray CF90 和其它编译器的内函数

函数	参数	描述
CLOC	([C=]c)	获取字符对象的地址
COMPL	([I=]i)	逐位补充单词。使用 NOT(i)。
COT	([X=]x)	一般余切。（同样可以使用：DCOT, QCOT）
CSMG	([I=]i,[J=]j,[K=]k)	有条件的标量合并
DSHIFTL	([I=]i,[J=]j,[K=]k)	将双对象 i 和 j 向左移动 k 个位
DSHIFTR	([I=]i,[J=]j,[K=]k)	将双对象 i 和 j 向右移动 k 个位
EQV	([I=]i,[J=]j)	逻辑等价。使用 IOER(i,j)。
FCD	([I=]i,[J=]j)	构造字符指针
GETPOS	([I=]i)	获取文件位置
IBCHNG	([I=]i, [POS=]j)	更改单词中指定位的一般函数。
ISHA	([I=]i, [SHIFT=]j)	一般算术移位
ISHC	([I=]i, [SHIFT=]j)	一般循环移位
ISHL	([I=]i, [SHIFT=]j)	一般左移位

表 2-3 Cray CF90 和其它编译器的内函数 *（续）*

函数	参数	描述
LEADZ	([I=]i)	统计前导 0 位的数量
LENGTH	([I=]i)	返回成功传送的 Cray 单词数
LOC	([I=]i)	返回变量的地址 （参见第 1-52 页上的 “loc: 返回对象的地址”）
NEQV	([I=]i,[J=]j)	逻辑非等价。使用 IOER(i,j)。
POPCNT	([I=]i)	统计设为 1 的位数。
POPPAR	([I=]i)	计算位总体的奇偶校验
SHIFT	([I=]i,[J=]j)	循环式左移。使用 ISHFT(i,j) 或 ISHFTC(i,j,k)。
SHIFTA	([I=]i,[J=]j)	带符号扩展的算术移位。
SHIFTL	([I=]i,[J=]j)	补零式左移。使用 ISHFT(i,j) 或 ISHFTC(i,j,k)。
SHIFTR	([I=]i,[J=]j)	补零式右移。使用 ISHFT(i,j) 或 ISHFTC(i,j,k)。
TIMEF	()	返回自第一次调用后经过的时间
UNIT	([I=]i)	返回 BUFFERIN 或 BUFFEROUT 的状态
XOR	([I=]i,[J=]j)	逻辑互斥 OR。使用 IOER(i,j)。

有关 VMS Fortran 77 内函数的列表，参见第 3 章。

2.3.4 其它扩展

Fortran 95 编译器识别以下其它的内函数：

2.3.4.1 MPI_SIZEOF

MPI_SIZEOF(*x*, *size*, *error*)
以机器表示的字节数返回指定变量的大小 *x*。如果 *x* 是数组，它返回基本元素的大小，而不是整个数组的大小。

x	输入；任意类型的变量或数组
size	输出；整数；以字节数表示的大小 *x*
error	输出；整数；设置为如果检测到错误显示错误代码，否则为零

2.3.4.2 内存函数

内存的分配、重新分配和释放函数 `malloc()`、 `realloc()` 和 `free()` 作为 `f95` 内函数实现。有关详细信息，请参见第 1-57 页上的 “`malloc`, `malloc64`, `realloc`, `free`: 分配/重新分配/释放内存”。

第 3 章

FORTRAN 77 和 VMS 内函数

本章列出了 `f95` 接受的 FORTRAN 77 内函数设置，并且有助于将传统的 FORTRAN 77 程序迁移至 Fortran 95。

`f95` 将本章列出的所有 FORTRAN 77 和 VMS 函数以及前面一章列出的所有 Fortran 95 函数识别为内函数。作为从传统的 FORTRAN 77 程序迁移至 `f95` 的辅助方法，使用 `-f77=intrinsics` 编译时，编译器只会将 FORTRAN 77 和 VMS 函数识别为内函数，而不会将 Fortran 95 函数识别为内函数。

属于 Sun 扩展的 ANSI FORTRAN 77 标准的内函数标有 ✠ 符号。使用非标准内函数和库函数的程序可能无法移植到其它平台。

内函数在接受多种数据类型的参数时，有*通用*名称和*特定*名称。一般说来，*通用*名称返回的是与参数的数据类型相同的值。但是，也有一些例外情况，如类型转换函数（表 3-2）和查询函数 （表 3-7）。这些函数也可以通过函数的某个*特定*名称进行调用，以便处理特定的参数数据类型。

对于处理多个数据项的函数 （例如 `sign(a1,a2)`），所有的数据参数必须具有相同的类型。

下面的表按以下几方面列出 FORTRAN 77 内函数：

- 内函数 — 描述函数的作用
- 定义 — 数学定义
- 参数数量 — 函数接受的参数的数量
- 通用名称 — 函数的通用名称
- 专用名称 — 函数的专用名称
- 参数类型 — 与每个专用名称关联的数据类型
- 函数类型 — 针对具体参数类型所返回的数据类型

注 – 编译器选项 `-xtypemap` 会更改变量的缺省大小，并影响对内函数的引用。参见第 3-13 页上的 “备注”以及《*Fortran 用户指南*》中有关缺省大小和对齐的介绍。

3.1 算术和数学函数

本节详细介绍算术函数、类型转换函数、三角函数以及其它函数。其中，“a”代表单参数函数的参数，“a1”和 “a2”分别代表双参数函数的第一个参数和第二个参数，“ar ”和 “ai”分别代表函数的复数参数的实部和虚部。

3.1.1 算术函数

表 3-1 Fortran 77 算术函数

内函数	定义	参数数量	通用名称	专用名称	参数类型	函数类型
绝对值 *参见注释 (6)*。	|a| = $(ar^2+ai^2)^{1/2}$	1	ABS	IABS ABS DABS CABS QABS ✠ ZABS ✠ CDABS ✠ CQABS ✠	INTEGER REAL DOUBLE COMPLEX REAL*16 DOUBLE COMPLEX DOUBLE COMPLEX COMPLEX*32	INTEGER REAL DOUBLE REAL REAL*16 DOUBLE DOUBLE REAL*16
截断 *参见注释 (1)*。	int(a)	1	AINT	AINT DINT QINT ✠	REAL DOUBLE REAL*16	REAL DOUBLE REAL*16
最近的整数	int(a+.5) （如果 a ≥ 0） int(a-.5) （如果 a < 0）	1	ANINT	ANINT DNINT QNINT ✠	REAL DOUBLE REAL*16	REAL DOUBLE REAL*16
最近的整数	int(a+.5) （如果 a ≥ 0） int(a-.5) （如果 a < 0）	1	NINT	NINT IDNINT IQNINT ✠	REAL DOUBLE REAL*16	INTEGER INTEGER INTEGER
余数 *参见注释 (1)*。	a1-int(a1/a2)*a2	2	MOD	MOD AMOD DMOD QMOD ✠	INTEGER REAL DOUBLE REAL*16	INTEGER REAL DOUBLE REAL*16

表 3-1 Fortran 77 算术函数 *（续）*

内函数	定义	参数数量	通用名称	专用名称	参数类型	函数类型
符号传输	|a1| （如果 a2 ≥ 0） -|a1| （如果 a2 < 0）	2	SIGN	ISIGN	INTEGER	INTEGER
				SIGN	REAL	REAL
				DSIGN	DOUBLE	DOUBLE
				QSIGN ✠	REAL*16	REAL*16
正数差异	a1-a2 （如果 a1 > a2） 0 （如果 a1 ≤ a2）	2	DIM	IDIM	INTEGER	INTEGER
				DIM	REAL	REAL
				DDIM	DOUBLE	DOUBLE
				QDIM ✠	REAL*16	REAL*16
两倍和四倍乘积	a1 * a2	2	-	DPROD	REAL	DOUBLE
				QPROD ✠	DOUBLE	REAL*16
选择最大的值	max(a1, a2, …)	≥2	MAX	MAX0	INTEGER	INTEGER
				AMAX1	REAL	REAL
				DMAX1	DOUBLE	DOUBLE
				QMAX1 ✠	REAL*16	REAL*16
			AMAX0	AMAX0	INTEGER	REAL
			MAX1	MAX1	REAL	INTEGER
选择最小的值	min(a1, a2, …)	≥2	MIN	MIN0	INTEGER	INTEGER
				AMIN1	REAL	REAL
				DMIN1	DOUBLE	DOUBLE
				QMIN1 ✠	REAL*16	REAL*16
			AMIN0	AMIN0	INTEGER	REAL
			MIN1	MIN1	REAL	INTEGER

3.1.2 类型转换函数

表 3-2 Fortran 77 类型转换函数

转换为	参数数量	通用名称	专用名称	参数类型	函数类型
INTEGER *参见注释 (1)*。	1	INT	-	INTEGER	INTEGER
			INT	REAL	INTEGER
			IFIX	REAL	INTEGER
			IDINT	DOUBLE	INTEGER
			-	COMPLEX	INTEGER
			-	COMPLEX*16	INTEGER
			-	COMPLEX*32	INTEGER
			IQINT ✠	REAL*16	INTEGER
REAL *参见注释 (2)*。	1	REAL	REAL	INTEGER	REAL
			FLOAT	INTEGER	REAL
			-	REAL	REAL
			SNGL	DOUBLE	REAL
			SNGLQ ✠	REAL*16	REAL
			-	COMPLEX	REAL
			-	COMPLEX*16	REAL
			-	COMPLEX*32	REAL
			FLOATK	INTEGER*8	REAL*4
DOUBLE *参见注释 (3)*。	1	DBLE	DBLE	INTEGER	DOUBLE PRECISION
			DFLOAT	INTEGER	DOUBLE PRECISION
			DFLOATK	INTEGER*8	DOUBLE PRECISION
			DREAL ✠	REAL	DOUBLE PRECISION
			-	DOUBLE	DOUBLE PRECISION
			-	COMPLEX	DOUBLE PRECISION
			-	COMPLEX*16	DOUBLE PRECISION
			-	REAL*16	DOUBLE PRECISION
			-	COMPLEX*32	DOUBLE PRECISION
			DBLEQ ✠	REAL*16	DOUBLE PRECISION
			-	COMPLEX*32	DOUBLE PRECISION

表 3-2 Fortran 77 类型转换函数 *（续）*

转换为	参数数量	通用名称	专用名称	参数类型	函数类型
REAL*16 *参见注释(3')。*	1	QREAL✠ QEXT ✠	QREAL ✠ QFLOAT ✠ - QEXT ✠ QEXTD ✠ - - - -	INTEGER INTEGER REAL INTEGER DOUBLE REAL*16 COMPLEX COMPLEX*16 COMPLEX*32	REAL*16 REAL*16 REAL*16 REAL*16 REAL*16 REAL*16 REAL*16 REAL*16 REAL*16
COMPLEX *参见注释 (4) 和 (8)。*	1 个或 2 个	CMPLX	- - - - - - -	INTEGER REAL DOUBLE REAL*16 COMPLEX COMPLEX*16 COMPLEX*32	COMPLEX COMPLEX COMPLEX COMPLEX COMPLEX COMPLEX COMPLEX
DOUBLE COMPLEX *参见注释 (8)。*	1 个或 2 个	DCMPLX ✠	- - - - - - -	INTEGER REAL DOUBLE REAL*16 COMPLEX COMPLEX*16 COMPLEX*32	DOUBLE COMPLEX DOUBLE COMPLEX DOUBLE COMPLEX DOUBLE COMPLEX DOUBLE COMPLEX DOUBLE COMPLEX DOUBLE COMPLEX
COMPLEX*32 *参见注释 (8)。*	1 个或 2 个	QCMPLX ✠	- - - - - - -	INTEGER REAL DOUBLE REAL*16 COMPLEX COMPLEX*16 COMPLEX*32	COMPLEX*32 COMPLEX*32 COMPLEX*32 COMPLEX*32 COMPLEX*32 COMPLEX*32 COMPLEX*32
INTEGER *参见注释 (5)。*	1	- -	ICHAR IACHAR ✠	字符	INTEGER
字符 *参见注释 (5)。*	1	- -	CHAR ACHAR ✠	INTEGER	字符

在 ASCII 平台上，包括 Sun 系统：

- ACHAR 是 CHAR 的非标准同义词。
- IACHAR 是 ICHAR 的非标准同义词。

在非 ASCII 平台上， ACHAR 和 IACHAR 是为了提供一种直接处理 ASCII 的方式。

3.1.3 三角函数

表 3-3 Fortran 77 三角函数

内函数	定义	参数数量	通用名称	专用名称	参数类型	函数类型
正弦 *参见注释 (7)。*	sin(a)	1	SIN	SIN DSIN QSIN ✠ CSIN ZSIN ✠ CDSIN ✠ CQSIN ✠	REAL DOUBLE REAL*16 COMPLEX DOUBLE COMPLEX DOUBLE COMPLEX COMPLEX*32	REAL DOUBLE REAL*16 COMPLEX DOUBLE COMPLEX DOUBLE COMPLEX COMPLEX*32
正弦 （度数） *参见注释 (7)。*	sin(a)	1	SIND ✠	SIND ✠ DSIND ✠ QSIND ✠	REAL DOUBLE REAL*16	REAL DOUBLE REAL*16
余弦 *参见注释 (7)。*	cos(a)	1	COS	COS DCOS QCOS ✠ CCOS ZCOS ✠ CDCOS ✠ CQCOS ✠	REAL DOUBLE REAL*16 COMPLEX DOUBLE COMPLEX DOUBLE COMPLEX COMPLEX*32	REAL DOUBLE REAL*16 COMPLEX DOUBLE COMPLEX DOUBLE COMPLEX COMPLEX*32
余弦 （度数） *参见注释 (7)。*	cos(a)	1	COSD ✠	COSD ✠ DCOSD ✠ QCOSD ✠	REAL DOUBLE REAL*16	REAL DOUBLE REAL*16
正切 *参见注释 (7)。*	tan(a)	1	TAN	TAN DTAN QTAN ✠	REAL DOUBLE REAL*16	REAL DOUBLE REAL*16
正切 （度数） *参见注释 (7)。*	tan(a)	1	TAND ✠	TAND ✠ DTAND ✠ QTAND ✠	REAL DOUBLE REAL*16	REAL DOUBLE REAL*16

表 **3-3** Fortran 77 三角函数 *（续）*

内函数	定义	参数数量	通用名称	专用名称	参数类型	函数类型
反正弦 *参见注释 (7)*。	arcsin(a)	1	ASIN	ASIN DASIN QASIN ✠	REAL DOUBLE REAL*16	REAL DOUBLE REAL*16
反正弦 （度数） *参见注释 (7)*。	arcsin(a)	1	ASIND ✠	ASIND ✠ DASIND ✠ QASIND ✠	REAL DOUBLE REAL*16	REAL DOUBLE REAL*16
反余弦 *参见注释 (7)*。	arccos(a)	1	ACOS	ACOS DACOS QACOS ✠	REAL DOUBLE REAL*16	REAL DOUBLE REAL*16
反余弦 （度数） *参见注释 (7)*。	arccos(a)	1	ACOSD ✠	ACOSD ✠ DACOSD ✠ QACOSD ✠	REAL DOUBLE REAL*16	REAL DOUBLE REAL*16
反正切 *参见注释 (7)*。	arctan(a)	1	ATAN	ATAN DATAN QATAN ✠	REAL DOUBLE REAL*16	REAL DOUBLE REAL*16
	arctan (a1/a2)	2	ATAN2	ATAN2 DATAN2 QATAN2 ✠	REAL DOUBLE REAL*16	REAL DOUBLE REAL*16
反正切 （度数） *参见注释 (7)*。	arctan(a)	1	ATAND ✠	ATAND ✠ DATAND ✠ QATAND ✠	REAL DOUBLE REAL*16	REAL DOUBLE REAL*16
	arctan (a1/a2)	2	ATAN2D✠	ATAN2D ✠ DATAN2D ✠ QATAN2D ✠	REAL DOUBLE REAL*16	REAL DOUBLE REAL*16
双曲正弦 *参见注释 (7)*。	sinh(a)	1	SINH	SINH DSINH QSINH ✠	REAL DOUBLE REAL*16	REAL DOUBLE REAL*16
双曲余弦 *参见注释 (7)*。	cosh(a)	1	COSH	COSH DCOSH QCOSH ✠	REAL DOUBLE REAL*16	REAL DOUBLE REAL*16
双曲正切 *参见注释 (7)*。	tanh(a)	1	TANH	TANH DTANH QTANH ✠	REAL DOUBLE REAL*16	REAL DOUBLE REAL*16

3.1.4 其它数学函数

表 3-4 其它 Fortran 77 数学函数

内函数	定义	参数数量	通用名称	专用名称	参数类型	函数类型
复数的虚部 *参见注释 (6)。*	ai	1	IMAG	AIMAG DIMAG ✠ QIMAG ✠	COMPLEX DOUBLE COMPLEX COMPLEX*32	REAL DOUBLE REAL*16
共轭复数 *参见注释 (6)。*	(ar, -ai)	1	CONJG	CONJG DCONJG ✠ QCONJG ✠	COMPLEX DOUBLE COMPLEX COMPLEX*32	COMPLEX DOUBLE COMPLEX COMPLEX*32
平方根	a**(1/2)	1	SQRT	SQRT DSQRT QSQRT ✠ CSQRT ZSQRT ✠ CDSQRT ✠ CQSQRT ✠	REAL DOUBLE REAL*16 COMPLEX DOUBLE COMPLEX DOUBLE COMPLEX COMPLEX*32	REAL DOUBLE REAL*16 COMPLEX DOUBLE COMPLEX DOUBLE COMPLEX COMPLEX*32
立方根 *参见注释 (8′)。*	a**(1/3)	1	CBRT	CBRT ✠ DCBRT ✠ QCBRT ✠ CCBRT ✠ ZCBRT ✠ CDCBRT ✠ CQCBRT ✠	REAL DOUBLE REAL*16 COMPLEX DOUBLE COMPLEX DOUBLE COMPLEX COMPLEX*32	REAL DOUBLE REAL*16 COMPLEX DOUBLE COMPLEX DOUBLE COMPLEX COMPLEX*32
指数	e**a	1	EXP	EXP DEXP QEXP ✠ CEXP ZEXP ✠ CDEXP ✠ CQEXP ✠	REAL DOUBLE REAL*16 COMPLEX DOUBLE COMPLEX DOUBLE COMPLEX COMPLEX*32	REAL DOUBLE REAL*16 COMPLEX DOUBLE COMPLEX DOUBLE COMPLEX COMPLEX*32
自然对数	log(a)	1	LOG	ALOG DLOG QLOG ✠ CLOG ZLOG ✠ CDLOG ✠ CQLOG ✠	REAL DOUBLE REAL*16 COMPLEX DOUBLE COMPLEX DOUBLE COMPLEX COMPLEX*32	REAL DOUBLE REAL*16 COMPLEX DOUBLE COMPLEX DOUBLE COMPLEX COMPLEX*32

表 3-4 其它 Fortran 77 数学函数 *（续）*

内函数	定义	参数数量	通用名称	专用名称	参数类型	函数类型
常用对数	log10(a)	1	LOG10	ALOG10 DLOG10 QLOG10 ✠	REAL DOUBLE REAL*16	REAL DOUBLE REAL*16
误差函数 *（参见下面的注释）*	erf(a)	1	ERF	ERF ✠ DERF ✠	REAL DOUBLE	REAL DOUBLE
误差函数	1.0 - erf(a)	1	ERFC	ERFC ✠ DERFC ✠	REAL DOUBLE	REAL DOUBLE

- 误差函数：*exp(-t*t) dt* 的从 *0* 到 *a* 的 *2/sqrt(pi)* x 整数

3.2 字符函数

表 3-5 Fortran 77 字符函数

内函数	定义	参数数量	特定名称	参数类型	函数类型
转换 *参见注释 (5)。*	转换为字符 转换为整数 *参见：* 表 3-2。	1 1	CHAR ACHAR ✠ ICHAR IACHAR ✠	INTEGER 字符	字符 INTEGER
子串的索引	字符串 a1 中子串 a2 的位置 *参见注释 (10)。*	2	INDEX	字符	INTEGER
长度	字符实体的长度 *参见注释 (11)。*	1	LEN	字符	INTEGER
词法上大于或等于	a1 ≥ a2 *参见注释 (12)。*	2	LGE	字符	LOGICAL

表 3-5 Fortran 77 字符函数

内函数	定义	参数数量	特定名称	参数类型	函数类型
词法上大于	a1 > a2 *参见注释 (12)*。	2	LGT	字符	LOGICAL
词法上小于或等于	a1 ≤ a2 *参见注释 (12)*。	2	LLE	字符	LOGICAL
词法上小于	a1 < a2 *参见注释 (12)*。	2	LLT	字符	LOGICAL

在 ASCII 平台上 （包括 Sun 系统）：

- ACHAR 是 CHAR 的非标准同义词。
- IACHAR 是 ICHAR 的非标准同义词。

在非 ASCII 平台上， ACHAR 和 IACHAR 是为了提供一种直接处理 ASCII 的方式。

3.3 杂项函数

其它杂项函数包括按位函数、环境查询函数以及内存分配和释放函数。

3.3.1 位操作 ✠

这些函数都不属于 FORTRAN 77 标准。

表 3-6 Fortran 77 按位函数

按位操作	参数数量	特定名称	参数类型	函数类型
补充	1	NOT	INTEGER	INTEGER
与	2	AND	INTEGER	INTEGER
	2	IAND	INTEGER	INTEGER
包容或	2	OR	INTEGER	INTEGER
	2	IOR	INTEGER	INTEGER
互斥或	2	XOR	INTEGER	INTEGER
	2	IEOR	INTEGER	INTEGER
移位 *参见注释 (14)*。	2	ISHFT	INTEGER	INTEGER

表 3-6 Fortran 77 按位函数 *（续）*

按位操作	参数数量	特定名称	参数类型	函数类型
左移 *参见注释 (14)。*	2	LSHIFT	INTEGER	INTEGER
右移 *参见注释 (14)。*	2	RSHIFT	INTEGER	INTEGER
逻辑右移 *参见注释 (14)。*	2	LRSHFT	INTEGER	INTEGER
循环移位	3	ISHFTC	INTEGER	INTEGER
提取位	3	IBITS	INTEGER	INTEGER
设置位	2	IBSET	INTEGER	INTEGER
测试位	2	BTEST	INTEGER	LOGICAL
清除位	2	IBCLR	INTEGER	INTEGER

以上函数可用作内函数或外函数。参见《*Fortran 库参考*》手册中介绍的库位操作例程。

3.3.2 环境查询函数 ✠

这些函数都不属于 FORTRAN 77 标准。

表 3-7 Fortran 77 环境查询函数

定义	参数数量	通用名称	参数类型	函数类型
编号系统的基数	1	EPBASE	INTEGER	INTEGER
			REAL	INTEGER
			DOUBLE	INTEGER
			REAL*16	INTEGER
有效位数	1	EPPREC	INTEGER	INTEGER
			REAL	INTEGER
			DOUBLE	INTEGER
			REAL*16	INTEGER
最小指数	1	EPEMIN	REAL	INTEGER
			DOUBLE	INTEGER
			REAL*16	INTEGER

表 3-7 Fortran 77 环境查询函数 *（续）*

定义	参数数量	通用名称	参数类型	函数类型
最大指数	1	EPEMAX	REAL	INTEGER
			DOUBLE	INTEGER
			REAL*16	INTEGER
最小的非零数	1	EPTINY	REAL	REAL
			DOUBLE	DOUBLE
			REAL*16	REAL*16
可表示的最大数	1	EPHUGE	INTEGER	INTEGER
			REAL	REAL
			DOUBLE	DOUBLE
			REAL*16	REAL*16
厄普西隆 *参见注释 (16)*。	1	EPMRSP	REAL	REAL
			DOUBLE	DOUBLE
			REAL*16	REAL*16

3.3.3 内存 ✠

这些函数都不属于 FORTRAN 77 标准。

表 3-8 Fortran 77 内存函数

内函数	定义	参数数量	特定名称	参数类型	函数类型
位置	地址 *参见注释 (17)*。	1	LOC	*Any*	INTEGER*4 INTEGER*8
分配	分配内存并返回地址。 *参见注释 (17)*。	1	MALLOC MALLOC64	INTEGER*4 INTEGER*8	INTEGER INTEGER*8
释放	释放 MALLOC *分配的内存。* *参见注释 (17)*。	1	FREE	*任意*	-
大小	返回以字节数表示的参数大小 *参见注释 (18)*。	1	SIZEOF	*任意表达式*	INTEGER

3.4 备注

以下备注适用于本章中的所有内函数表。

- 缩写 DOUBLE 代表 DOUBLE PRECISION。
- 采用 INTEGER 参数的内函数接受 INTEGER*2、 INTEGER*4 或 INTEGER*8。
- 采用 INTEGER 参数的 INTEGER 内函数返回下面确定的 INTEGER 类型的值。注意 -xtypemap 选项可能会更改实际参数的缺省大小：
  - mod sign dim max min and iand or ior xor ieor — 返回的值大小是最大参数的大小。
  - abs ishft lshift rshift lrshft ibset btest ivclr ishftc ibits — 返回的值大小是第一个参数的大小。
  - int epbase epprec — 返回的值大小是缺省 INTEGER 的大小。
  - ephuge — 返回的值大小是缺省 INTEGER 的大小或参数的大小，以两者中最大的值为准。
- 更改缺省数据大小的选项改变了一些内函数的使用方式。例如，在 -dbl 生效时，调用带 DOUBLE COMPLEX 参数的 ZCOS 自动变为调用 CQCOS，这是因为参数已经提升至 COMPLEX*32。以下函数也具有该功能：

  aimag alog amod cabs ccbrt ccos cdabs cdcbrt cdcos cdexp cdlog cdsin cdsqrt cexp clog csin csqrt dabs dacos dacosd dasin dasind datan datand dcbrt dconjg dcos dcosd dcosh ddim derf derfc dexp dimag dint dlog dmod dnint dprod dsign dsin dsind dsinh dsqrt dtan dtand dtanh idnint iidnnt jidnnt zabs zcbrt zcos zexp zlog zsin zsqrt

- 以下函数允许使用整数参数或任意大小的逻辑类型：

  and iand ieor iiand iieor iior inot ior jiand jieor jior jnot lrshft lshift not or rshift xor

- 显示以返回*缺省* REAL、DOUBLE PRECISION、COMPLEX 或 DOUBLE COMPLEX 值的内函数根据某些编译选项返回主要的类型。例如，您使用 -xtypemap=real:64,double:64 选项编译：
  - 调用 REAL 函数返回 REAL*8
  - 调用 DOUBLE PRECISION 函数返回 REAL*8
  - 调用 COMPLEX 函数返回 COMPLEX*16
  - 调用 DOUBLE COMPLEX 函数返回 COMPLEX*16

  更改缺省数据类型的数据大小的其它选项为 -r8 和 -dbl，它们也会从 DOUBLE 提升至 QUAD。 -xtypemap= 选项与其它早期的编译器选项相比更加灵活，因为它是首选的编译器选项。
- 具有通用名称的函数返回与参数相同类型的值 — 类型转换函数、最近的整数函数、复数参数的绝对值以及其它函数除外。如果有多个参数，它们的类型必须相同。
- 如果函数名称用作实际的参数，它必须是专用的名称。
- 如果函数名称用作虚拟的参数，则它不能识别子程序中的内函数，并且根据与变量和数组相同的规则具有数据类型。

3.4.1 有关函数的注释

表和注释 1 至 12 以《*ANSI X3.9-1978 Programming Language FORTRAN*》中的“内函数表”为基础，并增加了 Fortran 扩展。

(1) INT

如果 A 为整数类型，则 INT(A) 为 A。

如果 A 为实数或双精度类型：

如果 |A| < 1，则 INT(A) 为 0。
如果 |A| ≥ 1，则 INT(A) 是最大的整数，但是不超过 A 的幅度，并且它的符号与 A 的符号相同。（这样的数学整数值可能太大，无法符合计算机整数类型的要求。）

如果 A 为复数或双复数类型，则将以上规则应用于 A 的实部。

如果 A 为实数类型，则 IFIX(A) 与 INT(A) 相同。

(2) REAL

如果 A 为实数类型，则 REAL(A) 为 A。

如果 A 为整数或双精度类型，则 REAL(A) 的 A 重要部分的精度与实数据包含的精度差不多。

如果 A 为复数类型，则 REAL(A) 是 A 的实部。

如果 A 为双复数类型，则 REAL(A) 的 A 实部中重要部分的精度与实数据包含的精度差不多。

(3) DBLE

如果 A 为双精度类型，则 DBLE(A) 为 A。

如果 A 为整数或实数类型，则 DBLE(A) 的 A 中重要部分的精度与双精度数据包含的精度差不多。

如果 A 为复数类型，则 DBLE(A) 的 A 实部中重要部分的精度与双精度数据包含的精度差不多。

如果 A 为 COMPLEX*16 类型，则 DBLE(A) 为 A 的实部。

(3′) QREAL

如果 A 为 REAL*16 类型，则 QREAL(A) 为 A。

如果 A 为整数、实数或双精度类型，则 QREAL(A) 的 A 中重要部分的精度与 REAL*16 数据包含的精度差不多。

如果 A 为复数或双复数类型，则 QREAL(A) 的 A 实部中重要部分的精度与 REAL*16 数据包含的精度差不多。

如果 A 为 COMPLEX*16 或 COMPLEX*32 类型，则 QREAL(A) 为 A 的实部。

(4) CMPLX

如果 A 为复数类型，则 CMPLX(A) 为 A。

如果 A 为整数、实数或双精度类型，则 CMPLX(A) 为 REAL(A) + 0i。

如果 A1 和 A2 为整数、实数或双精度类型，则 CMPLX(A1,A2) 为 REAL(A1) + REAL(A2)*i。

如果 A 为双复数类型，则 CMPLX(A) 为 REAL(DBLE(A)) + i*REAL(DIMAG(A))。

如果 CMPLX 有两个参数，则它们必须属于同一个类型，而且必须是某个整数、实数或双精度。

如果 CMPLX 有一个参数，则它必须是某个整数、实数、双精度、复数、 COMPLEX*16 或 COMPLEX*32。

(4′) DCMPLX

如果 A 为 COMPLEX*16 类型，则 DCMPLX(A) 为 A。

如果 A 为整数、实数或双精度类型，则 DCMPLX(A) 为 DBLE(A) + 0i。

如果 A1 和 A2 为整数、实数或双精度类型，则 DCMPLX(A1,A2) 为 DBLE(A1) + DBLE(A2)*i。

如果 DCMPLX 有两个参数，则它们必须属于同一个类型，而且必须是某个整数、实数或双精度。

如果 DCMPLX 有一个参数，则它必须是某个整数、实数、双精度、复数、 COMPLEX*16 或 COMPLEX*32。

(5) ICHAR

ICHAR(A) 为 A 在整理序列中的位置。

第一个位置为 0，最后一个位置为 N-1， 0≤ ICHAR(A)≤ N-1，其中 N 是整理序列中的字符数， A 属于长度为 1 的字符类型。

CHAR 和 ICHAR 从以下几方面来看正好相反：

- ICHAR(CHAR(I)) = I，适用于 0≤ I≤ N-1
- CHAR(ICHAR(C)) = C，适用于能够在处理器中表示的任何字符 C

(6) COMPLEX

COMPLEX 值表示为排好的一对实数， (ar, ai)，其中 ar 为实部，而 ai 为虚部。

(7) 弧度

所有的角度以弧度表示，除非 “内函数”列包含 *“(度数)”* 说明。

(8) COMPLEX 函数

COMPLEX 类型的函数结果是主要值。

(8′) CBRT

如果 a 属于 COMPLEX 类型， CBRT 造成结果 COMPLEX RT1=(A, B)，其中：
A ≥ 0.0，并且 -60 度 ≤ arctan (B/A) < + 60 度。

其它两个结果的计算公式可能如下所示：

- RT2 = RT1 * (-0.5, square_root (0.75))
- RT3 = RT1 * (-0.5, square_root (0.75))

(9) 参数类型

内函数引用中的所有参数必须属于相同的类型。

(10) INDEX

INDEX(X,Y) 指 X 中开始出现 Y 的位置。也就是说，它指字符串 X 中第一次出现字符串 Y 的起始位置。

如果 Y 不在 X 中出现，则 INDEX(X,Y) 为 0。

如果 LEN(X) < LEN(Y)，则 INDEX(X,Y) 为 0。

INDEX 返回缺省的 INTEGER*4 数据。如果为了 64 位环境编译，在结果溢出 INTEGER*4 数据范围时，编译器将发出警告。要在 64 位环境中使用 INDEX，并且字符串超出 INTEGER*4 限制 (2 GB)， INDEX 函数以及接收结果的变量必须声明为 INTEGER*8。

(11) LEN

LEN 返回字符参数变量的声明长度。参数的实际值无关紧要。

LEN 返回缺省的 INTEGER*4 数据。如果为了 64 位环境编译，在结果溢出 INTEGER*4 数据范围时，编译器将发出警告。要在 64 位环境中使用 LEN，并且字符变量超出 INTEGER*4 限制 (2 GB)， LEN 函数以及接收结果的变量必须声明为 INTEGER*8。

(12) 词法比较

如果 X=Y 或者在整理序列中 X 位于 Y 之后， LGE(X, Y) 为 True ；否则为 False。

如果在整理序列中 X 位于 Y 之后， LGT(X, Y) 为 True ；否则为 False。

如果 X=Y 或者在整理序列中 X 位于 Y 之前， LLE(X, Y) 为 True ；否则为 False。

如果在整理序列中 X 位于 Y 之前， LLT(X, Y) 为 True ；否则为 False。

如果 LGE、LGT、LLE 和 LLT 的操作数长度不相等，则会考虑较短的操作数，就好象在右边加上空白。

(13) 位函数

在 VMS Fortran 中还有其它一些按位操作，但是没有实现。

(14) 移位

LSHIFT 将 *a1* 逻辑*左*移 *a2* 个位 （内联代码）。

LRSHFT 将 *a1* 逻辑*右*移 *a2* 个位 （内联代码）。

RSHIFT 将 *a1* 算术*右*移 *a2* 个位。

如果 *a2* > 0， ISHFT 将 *a1* 逻辑*左*移；如果 *a2* < 0，则*向右*移动。

LSHIFT 和 RSHIFT 是 Fortran 语言中类似 C 语言的 << 与 >> 运算符的函数。和在 C 语言中一样，它们的语义取决于硬件。

在移位计数超出范围时，移位函数的性能与硬件有关，并且一般无法预见。在该版本中，移位计数大于 31 将导致性能与硬件有关。

(15) 环境查询

仅对变量类型有意义。

(16) 厄普西隆

厄普西隆是最小的 e，例如 1.0 + e ≠ 1.0。

(17) LOC、 MALLOC 和 FREE

LOC 函数返回变量或外部过程的地址。函数调用 MALLOC(*n*) 会分配至少 *n* 个字节的块，并且返回该块的地址。

LOC 在 32 位环境中返回缺省的 INTEGER*4，在 64 位环境中返回缺省的 INTEGER*8。

MALLOC 是 FORTRAN 77 中的一个库函数，而不是一个内函数。它在 32 位环境中也返回缺省的 INTEGER*4，在 64 位环境中返回缺省的 INTEGER*8。但在 64 位环境中进行编译时，必须将 MALLOC 显式声明为 INTEGER*8。

在 64 位环境中， LOC 或 MALLOC 返回的值应该存储在类型为 POINTER、 INTEGER*4 或 INTEGER*8 的变量中。 FREE 的参数必须是上一次调用 MALLOC 返回的值，因此，它的数据类型必须是 POINTER、 INTEGER*4 或 INTEGER*8。

MALLOC64 始终采用 INTEGER*8 参数 （以字节数表示的内存请求大小）并且始终返回 INTEGER*8 值。在编译必须在 32 位环境和 64 位环境中都能运行的程序时，应使用该例程，而不要使用 MALLOC。接收变量必须声明为 POINTER 或 INTEGER*8。

(18) SIZEOF

SIZEOF 内函数不能应用于假定大小的数组、超长的字符或子例程调用或名称。SIZEOF 返回缺省的 INTEGER*4 数据。如果为了 64 位环境编译，在结果溢出 INTEGER*4 数据范围时，编译器将发出警告。要在 64 位环境中使用 SIZEOF，并且数组超出 INTEGER*4 限制 (2 GB)， SIZEOF 函数以及接收结果的变量必须声明为 INTEGER*8。

3.5 VMS 内函数

本小节列出了 f95 可以识别的 VMS FORTRAN 内例程。当然，它们不是标准的例程。✠

3.5.1 VMS 双精度复数

表 3-9 VMS 双精度复数函数

通用名称	专用名称	函数	参数类型	结果类型
	CDABS	绝对值	COMPLEX*16	REAL*8
	CDEXP	指数，e**a	COMPLEX*16	COMPLEX*16
	CDLOG	自然对数	COMPLEX*16	COMPLEX*16
	CDSQRT	平方根	COMPLEX*16	COMPLEX*16
	CDSIN	正弦	COMPLEX*16	COMPLEX*16
	CDCOS	余弦	COMPLEX*16	COMPLEX*16
DCMPLX		转换为 DOUBLE COMPLEX	Any numeric	COMPLEX*16
	DCONJG	复数共轭	COMPLEX*16	COMPLEX*16
	DIMAG	复数的虚部	COMPLEX*16	REAL*8
	DREAL	复数的实部	COMPLEX*16	REAL*8

3.5.2 VMS 基于度数的三角函数

表 3-10 VMS 基于度数的三角函数

通用名称	专用名称	函数	参数类型	结果类型
SIND		正弦	-	-
	SIND		REAL*4	REAL*4
	DSIND		REAL*8	REAL*8
	QSIND		REAL*16	REAL*16
COSD		余弦	-	-
	COSD		REAL*4	REAL*4
	DCOSD		REAL*8	REAL*8
	QCOSD		REAL*16	REAL*16
TAND		正切	-	-
	TAND		REAL*4	REAL*4
	DTAND		REAL*8	REAL*8
	QTAND		REAL*16	REAL*16
ASIND		反正弦	-	-
	ASIND		REAL*4	REAL*4
	DASIND		REAL*8	REAL*8
	QASIND		REAL*16	REAL*16
ACOSD		反余弦	-	-
	ACOSD		REAL*4	REAL*4
	DACOSD		REAL*8	REAL*8
	QACOSD		REAL*16	REAL*16
ATAND		反正切	-	-
	ATAND		REAL*4	REAL*4
	DATAND		REAL*8	REAL*8
	QATAND		REAL*16	REAL*16
ATAN2D		a1/a2 的反正切	-	-
	ATAN2D		REAL*4	REAL*4
	DATAN2D		REAL*8	REAL*8
	QATAN2D		REAL*16	REAL*16

3.5.3 VMS 位操作

表 3-11 VMS 位操作函数

通用名称	专用名称	函数	参数类型	结果类型
IBITS		根据 a1，初始位 a2，提取 a3 个位	-	-
	IIBITS		INTEGER*2	INTEGER*2
	JIBITS		INTEGER*4	INTEGER*4
	KIBITS		INTEGER*8	INTEGER*8
ISHFT		将 a1 逻辑移动 a2 个位；如果 a2 是正数，向左移动；如果 a2 是负数，向右移动。	-	-
	IISHFT		INTEGER*2	INTEGER*2
	JISHFT		INTEGER*4	INTEGER*4
	KISHFT		INTEGER*8	INTEGER*8
ISHFTC		在 a1 中，将右边的 a3 个位循环移动 a2 个站点。	-	-
	IISHFTC		INTEGER*2	INTEGER*2
	JISHFTC		INTEGER*4	INTEGER*4
IAND		a1， a2 的按位 AND	-	-
	IIAND		INTEGER*2	INTEGER*2
	JIAND		INTEGER*4	INTEGER*4
IOR		a1， a2 的按位 OR	-	-
	IIOR		INTEGER*2	INTEGER*2
	JIOR		INTEGER*4	INTEGER*4
	KIOR		INTEGER*8	INTEGER*8
IEOR		a1， a2 的按位互斥 OR	-	-
	IIEOR		INTEGER*2	INTEGER*2
	JIEOR		INTEGER*4	INTEGER*4
	KIEOR		INTEGER*8	INTEGER*8
NOT		按位补充	-	-
	INOT		INTEGER*2	INTEGER*2
	JNOT		INTEGER*4	INTEGER*4
	KNOT		INTEGER*8	INTEGER*8

表 3-11 VMS 位操作函数 *（续）*

通用名称	专用名称	函数	参数类型	结果类型
IBSET		在 a1 中，将位 a2 设为 1；返回新 a1	-	-
	IIBSET		INTEGER*2	INTEGER*2
	JIBSET		INTEGER*4	INTEGER*4
	KIBSET		INTEGER*8	INTEGER*8
BTEST		如果 a1 的位 a2 为 1，返回 TRUE。	-	-
	BITEST		INTEGER*2	INTEGER*2
	BJTEST		INTEGER*4	INTEGER*4
	BKTEST		INTEGER*8	INTEGER*8
IBCLR		在 a1 中，将位 a2 设为 0；返回新 a1	-	-
	IIBCLR		INTEGER*2	INTEGER*2
	JIBCLR		INTEGER*4	INTEGER*4
	KIBCLR		INTEGER*8	INTEGER*8

3.5.4 VMS 多个整数类型

Fortran 标准没有解决可能出现的多个整数类型问题。编译器将专用的 INTEGER 对 INTEGER 函数名称（IABS 等）作为特殊的一般种类处理，处理出现的多个整数类型。参数类型用于选择相应的运行时例程名称，而程序员无法访问该名称。

VMS Fortran 采用了类似的方法，但是可以使用特定名称。

表 3-12 VMS 整数函数

特定名称	函数	参数类型	结果类型
IIABS	绝对值	INTEGER*2	INTEGER*2
JIABS		INTEGER*4	INTEGER*4
KIABS		INTEGER*8	INTEGER*8
IMAX0	最大值 [1]	INTEGER*2	INTEGER*2
JMAX0		INTEGER*4	INTEGER*4
IMIN0	最小值 [1]	INTEGER*2	INTEGER*2
JMIN0		INTEGER*4	INTEGER*4

表 3-12 VMS 整数函数 *（续）*

特定名称	函数	参数类型	结果类型
IIDIM	正数差异[2]	INTEGER*2	INTEGER*2
JIDIM		INTEGER*4	INTEGER*4
KIDIM		INTEGER*8	INTEGER*8
IMOD	a1/a2 的余数	INTEGER*2	INTEGER*2
JMOD		INTEGER*4	INTEGER*4
IISIGN	符号传输， |a1|* sign(a2)	INTEGER*2	INTEGER*2
JISIGN		INTEGER*4	INTEGER*4
KISIGN		INTEGER*8	INTEGER*8

1 必须至少有两个参数。

2 正数差异为：a1- min(a1,a2)

索引

A

B

C

D

F

G

H

J

K

L

M

N

Q

R

S

T

W

X

Y

Z